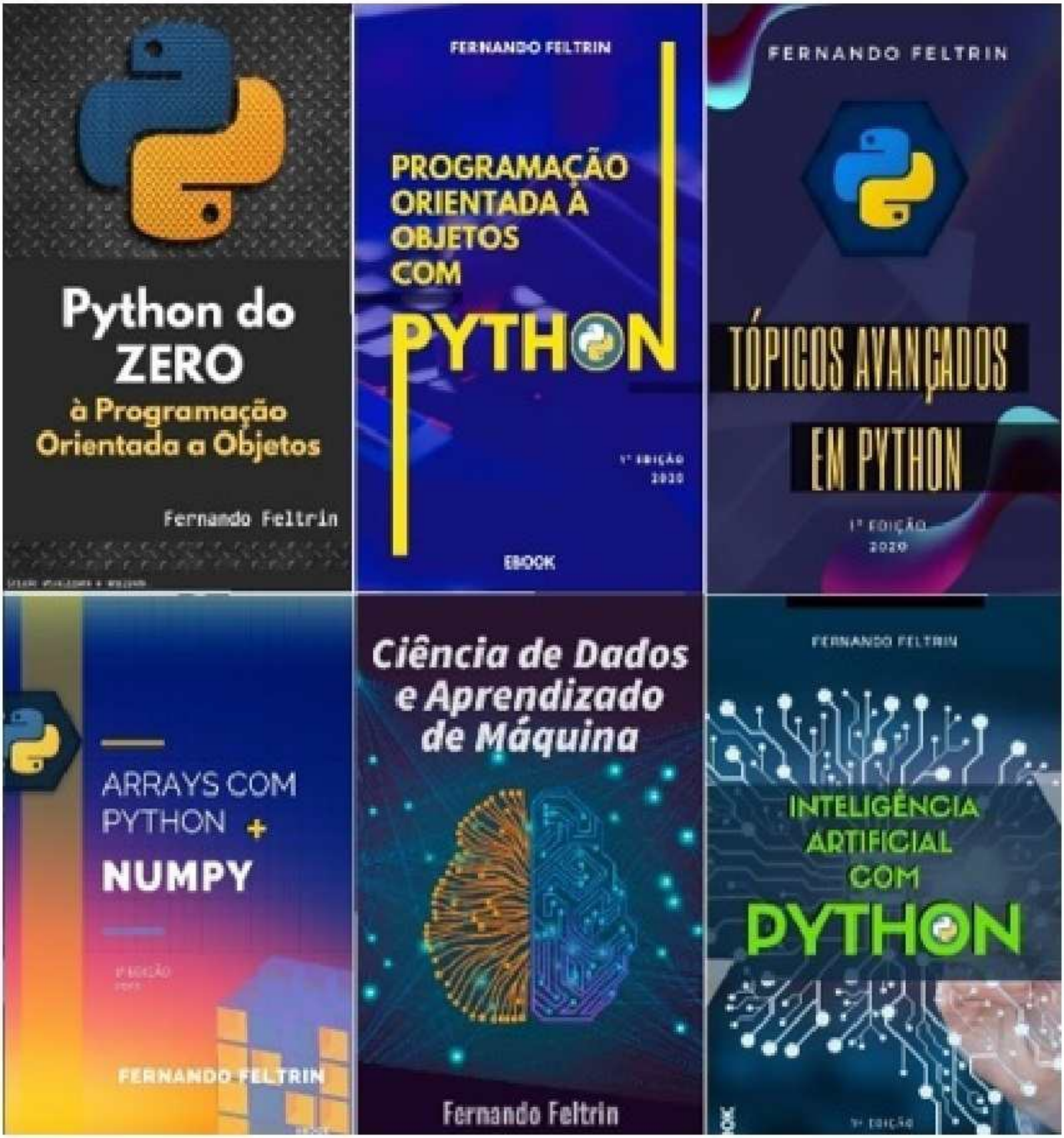
COLETÂNEA PYTHON DO ZERO ÀS REDES NEURAIS ARTIFICIAIS



6 LIVROS 1158 PÁGINAS

COLETÂNEA PYTHON DO ZERO ÀS REDES NEURAIS ARTIFICIAIS



6 LIVROS 1170 PÁGINAS

COLETÂNEA PYTHON DO ZERO ÀS REDES NEURAIS ARTIFICIAIS

FERNANDO FELTRIN

AVISOS

Este livro é um compilado dos seguintes títulos:

- Python do ZERO à Programação Orientada a Objetos
- Programação Orientada a Objetos
- Tópicos Avançados em Python
- Arrays + Numpy Com Python
- Ciência de Dados e Aprendizado de Máquina
- Inteligência Artificial com Python

Todo conteúdo foi reordenado de forma a não haver divisões entre os conteúdos de um livro e outro, cada tópico está rearranjado em posição contínua, formando assim um único livro. Quando necessário, alguns capítulos sofreram pequenas reutilizações e alterações em seu conteúdo a fim de deixar o conteúdo melhor contextualizado.

Este livro conta com mecanismo anti pirataria Amazon Kindle Protect DRM. Cada cópia possui um identificador próprio rastreável, a distribuição ilegal deste conteúdo resultará nas medidas legais cabíveis.

É permitido o uso de trechos do conteúdo para uso como fonte desde que dados os devidos créditos ao autor.

SUMÁRIO

CAPA

AVISOS
PREFÁCIO
Por quê programar? E por quê em Python?
Metodologia
INTRODUÇÃO
Por quê Python?
Um pouco de história
Guido Van Rossum
A filosofia do Python
Empresas que usam Python
O futuro da linguagem
Python será o limite?
AMBIENTE DE PROGRAMAÇÃO
Linguagens de alto, baixo nível e de máquina
Ambientes de desenvolvimento integrado
Principais IDEs
LÓGICA DE PROGRAMAÇÃO
Algoritmos
Sintaxe em Python
Palavras reservadas
Análise léxica
Indentação
ESTRUTURA BÁSICA DE UM PROGRAMA
TIPOS DE DADOS
COMENTÁRIOS

VARIÁVEIS / OBJETOS
Declarando uma variável
Declarando múltiplas variáveis
Declarando múltiplas variáveis (de mesmo tipo)
FUNÇÕES BÁSICAS
Função print( )
Função input( )
Explorando a função print( )
print( ) básico
print( ) básico
print( ) intermediário
print( ) avançado
Interação entre variáveis
Conversão de tipos de dados
OPERADORES
Operadores de Atribuição
Atribuições especiais
Atribuição Aditiva
Atribuição Subtrativa
Atribuição Multiplicativa
Atribuição Divisiva
Módulo
Exponenciação
Divisão Inteira
Operadores aritméticos

Soma
Subtração
Multiplicação
Divisão
Operações com mais de 2 operandos
Operações dentro de operações
Exponenciação
Operadores Lógicos
Tabela verdade
Tabela verdade AND
Tabela Verdade OR (OU)
Tabela Verdade XOR (OU Exclusivo/um ou outro)
Tabela de Operador de Negação (unário)
Bit-a-bit
Operadores de membro
Operadores relacionais
Operadores usando variáveis
Operadores usando condicionais
Operadores de identidade
ESTRUTURAS CONDICIONAIS
Ifs, elifs e elses
And e Or dentro de condicionais
Condicionais dentro de condicionais
Simulando switch/case
ESTRUTURAS DE REPETIÇÃO

While
For
STRINGS
Trabalhando com strings
Formatando uma string
Convertendo uma string para minúsculo
Convertendo uma string para maiúsculo
Buscando dados dentro de uma string
Desmembrando uma string
Alterando a cor de um texto
Alterando a posição de exibição de um texto
Formatando a apresentação de números em uma string
LISTAS
Adicionando dados manualmente
Removendo dados manualmente
Removendo dados via índice
Verificando a posição de um elemento
Verificando se um elemento consta na lista
Formatando dados de uma lista
Listas dentro de listas
Trabalhando com Tuplas
Trabalhando com Pilhas
Adicionando um elemento ao topo de pilha
Removendo um elemento do topo da pilha
Consultando o tamanho da pilha

DICIONÁRIOS
Consultando chaves/valores de um dicionário
Consultando as chaves de um dicionário
Consultando os valores de um dicionário
Mostrando todas chaves e valores de um dicionário
Manipulando dados de um dicionário
Adicionando novos dados a um dicionário
CONJUNTOS NUMÉRICOS
União de conjuntos
Interseção de conjuntos
Verificando se um conjunto pertence ao outro
Diferença entre conjuntos
INTERPOLAÇÃO
Avançando com interpolações
FUNÇÕES
Funções predefinidas
Funções personalizadas
Função simples, sem parâmetros
Função composta, com parâmetros
Função composta, com *args e **kwargs
dir( ) e help( )
BUILTINS
Importando bibliotecas
MÓDULOS E PACOTES
Modularização

Importando de módulos
Importando de pacotes
PROGRAMAÇÃO ORIENTADA A OBJETOS
Classes
Definindo uma classe
Alterando dados/valores de uma instância
Aplicando recursividade
Herança
Polimorfismo
Encapsulamento
TRACEBACKS / EXCEÇÕES
Comandos try, except e finally
AVANÇANDO EM POO
Objetos e Classes
Variáveis vs Objetos
Criando uma classe vazia
Atributos de classe
Manipulando atributos de uma classe
Métodos de classe
Método construtor de uma classe
Escopo / Indentação de uma classe
Classe com parâmetros opcionais
Múltiplos métodos de classe
Interação entre métodos de classe
Estruturas condicionais em métodos de classe

Métodos de classe estáticos e dinâmicos
Getters e Setters
Encapsulamento
Associação de classes
Agregação e composição de classes
Herança Simples
Cadeia de heranças
Herança Múltipla
Sobreposição de membros
Classes abstratas
Polimorfismo
Sobrecarga de operadores
Tratando exceções
AVANÇANDO COM FUNÇÕES
Parâmetros por Justaposição ou Nomeados
*args e **kwargs
Empacotamento e Desempacotamento
Expressões Lambda
RECURSIVIDADE
EXPRESSÕES REGULARES
r’Strings e Metacaracteres
Metacaracteres
Metacaracteres Quantificadores
Métodos de Expressões Regulares
LIST COMPREHENSION

Desmembrando strings via List Comprehension
Manipulando partes de uma string via List Comprehension
DICTIONARY COMPREHENSION
EXPRESSÕES TERNÁRIAS
GERADORES E ITERADORES
Geradores e Memória
ZIP
Zip longest
COUNT
MAP
Map iterando sobre uma lista
Map iterando sobre um dicionário
FILTER
REDUCE
COMBINANDO MAP, FILTER E REDUCE
Extraindo dados via Map + Função Lambda
Extraindo dados via List Comprehension
Extraindo dados via Filter
Combinando Filter e List Comprehension
Combinando Map e Filter
Extraindo dados via Reduce
Reduce + List Comprehension
Reduce (Método Otimizado)
TRY, EXCEPT
Finally

Raise
BIBLIOTECAS, MÓDULOS E PACOTES
Referenciando uma biblioteca/módulo
Criando módulos/pacotes
Testando e validando um módulo
Encapsulamento
ITERTOOLS
combinations( )
permutations( )
product( )
TÓPICOS AVANÇADOS EM POO
Métodos de classe
Métodos estáticos
@property, @getters e @setters
Associação de Classes
Agregação e Composição de Classes
Herança
Cadeia de Heranças
Herança Múltipla
Sobreposição de Membros
super( )
Classes Abstratas
Polimorfismo
Sobrecarga de Operadores
Filas (Queues) e Nodes

Gerenciamento de filas via Deque
PEPS
REPOSITÓROS
PYTHON + NUMPY
Sobre a biblioteca Numpy
Instalação e Importação das Dependências
Trabalhando com Arrays
Criando uma array numpy
Criando uma array gerada com números ordenados
Array gerada com números do tipo float
Array gerada com números do tipo int
Array gerada com números zero
Array gerada com números um
Array gerada com espaços vazios
Criando uma array de números aleatórios, mas com tamanho predefinido em variável
Criando arrays de dimensões específicas
Array unidimensional
Array bidimensional
Array tridimensional
Convertendo uma array multidimensional para unidimensional
Verificando o tamanho e formato de uma array
Verificando o tamanho em bytes de um item e de toda a array
Verificando o elemento de maior valor de uma array
Consultando um elemento por meio de seu índice

Consultando elementos dentro de um intervalo
Modificando manualmente um dado/valor de um elemento por meio de seu índice.
Criando uma array com números igualmente distribuídos
Redefinindo o formato de uma array
Usando de operadores lógicos em arrays
Usando de operadores aritméticos em arrays
Criando uma matriz diagonal
Criando padrões duplicados
Somando um valor a cada elemento da array
Realizando soma de arrays
Subtração entre arrays
Multiplicação e divisão entre arrays
Realizando operações lógicas entre arrays
Transposição de arrays
Salvando uma array no disco local
Carregando uma array do disco local
CIÊNCIA DE DADOS E APRENDIZADO DE MÁQUINA
Nota do Autor
O que é Ciência de Dados e Aprendizado de Máquina?
O que são Redes Neurais Artificiais
O que é Aprendizado de Máquina
Como se trabalha com Ciência de Dados? Quais ferramentas utilizaremos?
Quando e como utilizaremos tais ferramentas?
Qual a Abordagem que Será Utilizada?

PREPARAÇÃO DO AMBIENTE DE TRABALHO
Instalação da Linguagem Python
Instalação da Suite Anaconda
Ambiente de Desenvolvimento
Instalação das Dependências
TEORIA E PRÁTICA EM CIÊNCIA DE DADOS
O que são Redes Neurais Artificiais
Teoria Sobre Redes Neurais Artificiais
O que é Aprendizado de Máquina
Aprendizado de Máquina
Tipos de Aprendizado de Máquina
Aprendizado por Reforço em Detalhes
Características Específicas do Aprendizado por Reforço
Principais Algoritmos para Inteligência Artificial
Perceptrons
Perceptron Multicamada
Deep Learning
Q-Learning e Deep Q-Learning
Modelo de Rede Neural Artificial Intuitiva
Rotinas de uma Rede Neural Artificial
APRENDIZADO DE MÁQUINA
Perceptron de Uma Camada – Modelo Simples
Perceptron de Uma Camada – Tabela AND
Perceptron Multicamada – Tabela XOR
REDES NEURAIS ARTIFICIAIS

Processamento de Linguagem Natural – Minerando Emoções
Classificação Multiclasse – Aprendizado de Máquina Simples – Iris Dataset
Classificação Multiclasse via Rede Neural Artificial – Iris Dataset
Classificação Binária via Rede Neural Artificial – Breast Cancer Dataset
Regressão de Dados de Planilhas Excel – Autos Dataset
Regressão com Múltiplas Saídas – VGDB Dataset
Previsão Quantitativa – Publi Dataset
REDES NEURAIS ARTIFICIAIS CONVOLUCIONAIS
Reconhecimento de Caracteres – Digits Dataset
Classificação de Dígitos Manuscritos – MNIST Dataset
Classificação de Imagens de Animais – Bichos Dataset
Classificação a Partir de Imagens – TensorFlow – Fashion Dataset
REDES NEURAIS ARTIFICIAIS RECORRENTES
Previsão de Séries Temporais – Bolsa Dataset
OUTROS MODELOS DE REDES NEURAIS ARTIFICIAIS
Mapas Auto Organizáveis – Kmeans – Vinhos Dataset
Sistemas de Recomendação – Boltzmann Machines
REDES NEURAIS ARTIFICIAIS INTUITIVAS
BOT Para mercado de Ações
Cão Robô que aprende sobre seu corpo e sobre o ambiente via ARS
Agente Autônomo que joga BreakOut (Atari)
PARÂMETROS ADICIONAIS
CAPÍTULO RESUMO
CONSIDERAÇÕES FINAIS

BÔNUS

Revisão e Exemplos

Artigo - Classificador Multiclasse via TensorFlow e Keras

PREFÁCIO

Por quê programar? E por quê em Python?

No âmbito da tecnologia da informação, basicamente começamos a dividir seus nichos entre a parte física (hardware) e sua parte lógica (software), e dentro de cada uma delas existe uma infinidade de subdivisões, cada uma com suas particularidades e usabilidades diferentes.

O aspirante a profissional de T.I. pode escolher entre várias muitas áreas de atuação, e mesmo escolhendo um nicho bastante específico ainda assim há um mundo de conhecimento a ser explorado. Dentro da parte lógica um dos diferenciais é a área da programação, tanto pela sua complexidade quanto por sua vasta gama de possibilidades.

Sendo assim um dos diferenciais mais importantes do profissional de tecnologia moderno é o mesmo ter certa bagagem de conhecimento de programação. No âmbito acadêmico existem diversos cursos, que vão de análise e desenvolvimento de sistemas até engenharia da computação, e da maneira como esses cursos são organizados você irá reparar que sempre haverá em sua grade curricular uma carga horária dedicada a programação.

No Brasil a linguagem de programação mais popularmente utilizada nos cursos de tecnologia é C ou uma de suas vertentes, isso se dá pelo fato de C ser uma linguagem ainda hoje bastante popular e que pode servir de base para tantas outras.

Quando estamos falando especificamente da área da programação existe uma infinidade de linguagens de programação que foram sendo desenvolvidas ao longo do tempo para suprir a necessidade de criação de softwares que atendessem uma determinada demanda. Poderíamos dedicar um capítulo inteiro mostrando apenas as principais e suas características, mas ao invés disso vamos nos focar logo em Python.

Hoje com a chamada internet das coisas, data science, machine learning, além é claro da criação de softwares, jogos e sistemas, mais do que nunca foi preciso profissionais da área que soubessem programação.

Python é uma linguagem idealizada e criada na década de 80, mas que se mostra hoje uma das mais modernas e promissoras, devido sua

facilidade de aprendizado e sua capacidade de se adaptar a qualquer situação. Se você buscar qualquer comparativo de Python em relação a outras linguagens de programação garanto que em 95% dos casos Python sairá em vantagem.

Python pode ser a sua linguagem de programação definitiva, ou abrir muitas portas para aprender outras mais, já que aqui não existe uma real concorrência, a melhor linguagem sempre será aquela que irá se adaptar melhor ao programador e ao projeto a ser desenvolvido.

Sendo assim, independentemente se você já é programador de outra linguagem ou se você está começando do zero, espero que o conteúdo deste pequeno livro seja de grande valia para seu aprendizado dentro dessa área incrível.

Metodologia

Este material foi elaborado com uma metodologia autodidata, de forma que cada conceito será explicado de forma progressiva, sucinta e exemplificado em seguida.

Cada tópico terá seus exemplos e devida explicação, assim como sempre que necessário você terá o código na íntegra e em seguida sua ‘engenharia reversa’, explicando ponto a ponto o que está sendo feito e os porquês de cada argumento dentro do código.

Cada tópico terá ao menos um exemplo, cada termo dedicado a programação terá destaque para que você o diferencie em meio ao texto ou explicações.

Desde já tenha em mente que aprender a programar requer atenção e mais do que isso, muita prática. Sendo assim, recomendo que sempre que possível pratique recriando os códigos de exemplo e não tenha medo que testar diferentes possibilidades em cima deles.

Dadas as considerações iniciais, mãos à obra!!!

INTRODUÇÃO

Quando estamos iniciando nossa jornada de aprendizado de uma linguagem de programação é importante que tenhamos alguns conceitos bem claros já logo de início. O primeiro deles é que aprender uma linguagem de programação não é muito diferente do que aprender outro idioma falado, haverá uma sintaxe e uma sequência lógica de argumentos a se respeitar a fim de que seus comandos façam sentido e funcionem.

Com certeza quando você começou a aprender inglês na escola você começou pelo verbo to be e posteriormente foi incrementando com novos conceitos e vocabulário, até ter certo domínio sobre o mesmo.

Em uma linguagem de programação não é muito diferente, há uma sintaxe, que é a maneira com que o interpretador irá reconhecer seus comandos, e há também uma lógica de programação a ser seguida uma vez que queremos através de linhas/blocos de código dar instruções ao computador e chegar a algum resultado.

Se você pesquisar, apenas por curiosidade, sobre as linguagens de programação você verá que outrora elas eram extremamente complexas, com uma curva de aprendizado longo e que por fim eram pouco eficientes ou de uso bastante restrito. As linguagens de alto nível, como Python, foram se modernizando de forma a que hoje é possível fazer muito com pouco, de forma descomplicada e com poucas linhas de código já estaremos criando programas e/ou fazendo a manutenção dos mesmos.

Se você realmente tem interesse por essa área arrisco dizer que você irá adorar Python e programação em geral. Então chega de enrolação e vamos começar... do começo.

Por quê Python?

Como já mencionei anteriormente, um dos grandes diferenciais do Python, que normalmente é o chamativo inicial por quem busca uma linguagem de programação, é sua facilidade, seu potencial de fazer mais com menos. Não desmerecendo outras linguagens, mas é fato que Python foi desenvolvida para ser descomplicada. E por quê fazer as coisas da forma mais difícil se existem ferramentas para torná-las mais fáceis?

Existe uma piada interna do pessoal que programa em Python que diz:

“- A vida é curta demais para programar em outra linguagem senão em Python.“

Como exemplo veja três situações, um simples programa que exibe em tela a mensagem “Olá Mundo!!!” escrito em C, em JAVA, e por fim em Python.

Hello World!!! em C

```c
#include <stdio.h>

int main(void) {
  printf("Hello World\n");
  return 0;
}

```

Hello World!!! em JAVA

```java
class Main {
  public static void main(String[] args) {
    System.out.println("Hello world!");
  }
}

```

Hello World!!! em Python

```
1   print('Hello World!')
2
```

Em todos os exemplos acima o retorno será uma mensagem exibida em tela para o usuário dizendo: Hello World!!! (*Garanto que agora a piada fez sentido...)

De forma geral o primeiro grande destaque do Python frente a outras linguagens é sua capacidade de fazer mais com menos, e em todas as situações que conheço este padrão se repetirá, será muito mais fácil, mais rápido, com menos linhas de código, programar em Python.

Outro grande diferencial do Python em relação a outras linguagens de programação é o fato dela ser uma linguagem interpretada, em outras palavras, o ambiente de programação que você irá trabalhar tem capacidade de rodar o código em tempo real e de forma nativa, diferente de outras linguagens que tem que estar emulando uma série de parâmetros do sistema ou até mesmo compilando o código para que aí sim seja possível testá-lo.

Outro diferencial é a sua simplicidade sintática, posteriormente você aprenderá sobre a sintaxe Python, mas por hora apenas imagine que em programação (outras linguagens) muitas vezes temos uma grande ideia e ao codificá-la acabamos nos frustrando porque o código simplesmente não funciona, e em boa parte das vezes é por conta de um ponto ou vírgula que ficou sobrando ou faltando no código.

Python por ser uma linguagem interpretada deveria sofrer ainda mais com esses problemas, mas o que ocorre é ao contrário, o interpretador foca nos comandos e seus parâmetros, os pequenos erros de sintaxe não farão com que o código não funcione...

Por fim, outro grande diferencial do Python se comparado a outras linguagens de programação é que ela nativamente possui um núcleo capaz de trabalhar com uma quantidade enorme de dados de forma bastante tranquila, o que a fez virar a queridinha de quem trabalha com data science, machine learning, blockchain, e outras tecnologias que trabalham e processam

volumes enormes de dados.

Na verdade, eu poderia escrever outro livro só comparando Python com outras linguagens e mostrando suas vantagens, mas acho que por hora já é o suficiente para lhe deixar empolgado, e ao longo desse curso você irá descobrir um potencial muito grande em Python.

Um pouco de história

Em 1989, através do Instituto de Pesquisa Nacional para Matemática e Ciência da Computação, Guido Van Rossum publicava a primeira versão do Python. Derivada do C, a construção do Python se deu inicialmente para ser uma alternativa mais simples e produtiva do que o próprio C.

Por fim em 1991 a linguagem Python ganhava sua “versão estável” e já funcional, começando a gerar também uma comunidade dedicada a aprimorá-la. Somente em 1994 foi lançada sua versão 1.0, ou seja, sua primeira versão oficial e não mais de testes, e de lá para cá houveram gigantescas melhorias na linguagem em si, seja em estrutura quanto em bibliotecas e plugins criadas pela comunidade para implementar novos recursos a mesma e a tornar ainda mais robusta.

Atualmente o Python está integrado em praticamente todas novas tecnologias, assim como é muito fácil implementá-la em sistemas “obsoletos”. Grande parte das distribuições Linux possuem Python nativamente e seu reconhecimento já fez com que, por exemplo, virasse a linguagem padrão do curso de ciências da computação do MIT desde 2009.

Guido Van Rossum

Não posso deixar de falar, ao menos um pouco, sobre a mente por trás da criação do Python, este cara chamado Guido Van Rossum. Guido é um premiado matemático e programador que hoje se dedica ao seu emprego atual na Dropbox. Guido já trabalhou em grandes empresas no passado e tem um grande reconhecimento por estudantes e profissionais da computação.

Dentre outros projetos, Guido em 1991 lançou a primeira versão de sua própria linguagem de programação, o Python, que desde lá sofreu inúmeras melhorias tanto pelos seus desenvolvedores quanto pela comunidade e ainda hoje ele continua supervisionando o desenvolvimento da linguagem Python, tomando as decisões quando necessário.

A filosofia do Python

Python tem uma filosofia própria, ou seja, uma série de porquês que são responsáveis por Python ter sido criada e por não ser “só mais uma linguagem de programação”.

Por Tim Peters, um influente programador de Python

1. Bonito é melhor que feio.

2. Explícito é melhor que implícito.

3. Simples é melhor que complexo.

4. Complexo é melhor que complicado.

5. Plano é melhor que aglomerado.

6. Escasso é melhor que denso.

7. O que conta é a legibilidade.

8. Casos especiais não são especiais o bastante para quebrar as regras.

9. A natureza prática derruba a teórica.

10. Erros nunca deveriam passar silenciosamente.

11. a menos que explicitamente silenciasse.

12. Diante da ambiguidade, recuse a tentação de adivinhar.

13. Deveria haver um -- e preferivelmente só um -- modo óbvio para fazer as coisas.

14. Embora aquele modo possa não ser óbvio a menos que você seja holandês.

15. Agora é melhor que nunca.

16. Embora nunca é frequentemente melhor que *agora mesmo*.

17. Se a implementação é difícil para explicar, isto é uma ideia ruim.

18. Se a implementação é fácil para explicar, pode ser uma ideia boa.

19. Namespaces são uma grande ideia -- façamos mais desses!

Essa filosofia é o que fez com que se agregasse uma comunidade enorme disposta a investir seu tempo em Python. Em suma, programar em Python deve ser simples, de fácil aprendizado, com código de fácil leitura, enxuto, mas inteligível, capaz de se adaptar a qualquer necessidade.

Empresas que usam Python

Quando falamos de linguagens de programação, você já deve ter reparado que existem inúmeras delas, mas basicamente podemos dividi-las em duas grandes categorias: Linguagens específicas e/ou Linguagens Generalistas. Uma linguagem específica, como o próprio nome sugere, é aquela linguagem que foi projetada para atender a um determinado propósito fixo, como exemplo podemos citar o PHP e o HTML, que são linguagens específicas para web. Já linguagens generalistas são aquelas que tem sua aplicabilidade em todo e qualquer propósito, e nem por isso ser inferior às específicas.

No caso do Python, ela é uma linguagem generalista bastante moderna, é possível criar qualquer tipo de sistema para qualquer propósito e plataforma a partir dela. Só para citar alguns exemplos, em Python é possível programar para qualquer sistema operacional, web, mobile, data science, machine learning, blockchain, etc... coisa que outras linguagens de forma nativa não são suficientes ou práticas para o programador, necessitando uma série de gambiarras para realizar sua codificação, tornando o processo mais difícil.

Como exemplo de aplicações que usam parcial ou totalmente Python podemos citar YouTube, Google, Instagram, Dropbox, Quora, Pinterest, Spotify, Reddit, Blender 3D, BitTorrent, etc...

Apenas como curiosidade, em computação gráfica dependendo sua aplicação uma engine pode trabalhar com volumosos dados de informação, processamento e renderização, a Light and Magic, empresa subsidiária da Disney, que produz de maneira absurda animações e filmes com muita computação gráfica, usa de motores gráficos escritos e processados em Python devido sua performance.

O futuro da linguagem

De acordo com as estatísticas de sites especializados, Python é uma das linguagens com maior crescimento em relação às demais no mesmo período, isto se deve pela popularização que a linguagem recebeu após justamente grandes empresas declarar que a adotaram e comunidades gigantescas se formarem para explorar seu potencial.

Em países mais desenvolvidos tecnologicamente até mesmo escolas de ensino fundamental estão adotando o ensino de programação em sua grade de disciplinas, boa parte delas, ensinando nativamente Python.

Por fim, podemos esperar para o futuro que a linguagem Python cresça exponencialmente, uma vez que novas áreas de atuação como data science e machine learning se popularizem ainda mais.

Estudos indicam que para os próximos 10 anos cerca de um milhão de novas vagas surgirão demandando profissionais de tecnologia da área da programação, pode ter certeza que grande parcela desse público serão programadores com domínio em Python.

Python será o limite?

Esta é uma pergunta interessante de se fazer porque precisamos parar uns instantes e pensar no futuro. Raciocine que temos hoje um crescimento exponencial do uso de machine learning, data science, internet das coisas, logo, para o futuro podemos esperar uma demanda cada vez maior de processamento de dados, o que não necessariamente signifique que será mais complexo desenvolver ferramentas para suprir tal demanda.

A versão 3 do Python é bastante robusta e consegue de forma natural já trabalhar com tais tecnologias. Devido a comunidade enorme que desenvolve para Python, podemos esperar que para o futuro haverão novas versões implementando novos recursos de forma natural. Será muito difícil vermos surgir outra linguagem “do zero” ou que tenha usabilidade parecida com Python.

Se você olhar para trás verá uma série de linguagens que foram descontinuadas com o tempo, mesmo seus desenvolvedores insistindo e injetando tempo em dinheiro em seu desenvolvimento elas não eram modernas o suficiente. Do meu ponto de vista não consigo ver um cenário do futuro ao qual Python não consiga se adaptar.

Entenda que na verdade não é uma questão de uma linguagem concorrer contra outra, na verdade independente de qual linguagem formos usar, precisamos de uma que seja capaz de se adaptar ao seu tempo e as nossas necessidades.

Num futuro próximo nosso diferencial como profissionais da área de tecnologia será ter conhecimento sobre C#, Java ou Python. Garanto que você já sabe em qual delas estou apostando minhas fichas...

AMBIENTE DE PROGRAMAÇÃO

Na linguagem Python, e, não diferente das outras linguagens de programação, quando partimos do campo das ideias para a prática, para codificação/programação não basta que tenhamos um computador rodando seu sistema operacional nativo.

É importante começarmos a raciocinar que, a partir do momento que estamos entrando na programação de um sistema, estamos trabalhando com seu backend, ou seja, com o que está por trás das cortinas, com aquilo que o usuário final não tem acesso.

Para isto, existe uma gama enorme de softwares e ferramentas que nos irão auxiliar a criar nossos programas e levá-los ao frontend.

Linguagens de alto, baixo nível e de máquina

Seguindo o raciocínio lógico do tópico anterior, agora entramos nos conceitos de linguagens de alto e baixo nível e posteriormente a linguagem de máquina.

Quando estamos falando em linguagens de alto e baixo nível, estamos falando sobre o quão distante está a sintaxe do usuário. Para ficar mais claro, uma linguagem de alto nível é aquela mais próximo do usuário, que usa termos e conceitos normalmente vindos do inglês e que o usuário pode pegar qualquer bloco de código e o mesmo será legível e fácil de compreender.

Em oposição ao conceito anterior, uma linguagem de baixo nível é aquela mais próxima da máquina, com instruções que fazem mais sentido ao interpretador do que ao usuário.

Quando estamos programando em Python estamos num ambiente de linguagem de alto nível, onde usaremos expressões em inglês e uma sintaxe fácil para por fim dar nossas instruções ao computador. Esta linguagem posteriormente será convertida em linguagem de baixo nível e por fim se tornará sequências de instruções para registradores e portas lógicas.

Imagine que o comando que você digita para exibir um determinado texto em tela é convertido para um segundo código que o interpretador irá ler como bytecode, convertendo ele para uma linguagem de mais baixo nível chamada Assembly, que irá pegar tais instruções e converter para binário, para que por fim tais instruções virem sequências de chaveamento para portas lógicas do processador.

Nos primórdios da computação se convertiam algoritmos em sequências de cartões perfurados que iriam programar sequências de chaveamento em máquinas ainda valvuladas, com o surgimento dos transistores entramos na era da eletrônica digital onde foi possível miniaturizar os registradores, trabalhar com milhares deles de forma a realizar centenas de milhares de cálculos por segundo.

Desenvolvemos linguagens de programação de alto nível para que

justamente facilitássemos a leitura e escrita de nossos códigos, porém a linguagem de máquina ainda é, e por muito tempo será, binário, ou seja, sequências de informações de zeros e uns que se convertem em pulsos ou ausência de pulsos elétricos nos transistores do processador.

Ambientes de desenvolvimento integrado

Entendidos os conceitos de linguagens de alto e baixo nível e de máquina, por fim vamos falar sobre os IDEs, sigla para ambiente de desenvolvimento integrado. Já rodando nosso sistema operacional temos a frontend do sistema, ou seja, a capa ao qual o usuário tem acesso aos recursos do mesmo.

Para que possamos ter acesso aos bastidores do sistema e por fim programar em cima dele, temos softwares específicos para isto, os chamados IDE’s. Nestes ambientes temos as ferramentas necessárias para tanto trabalhar a nível de código quanto para testar o funcionamento de nossos programas no sistema operacional.

As IDEs vieram para integrar todos softwares necessários para esse processo de programação em um único ambiente, já houveram épocas onde se programava separado de onde se compilava, separado de onde se debugava e por fim separado de onde se rodava o programa, entenda que as ides unificam todas camadas necessárias para que possamos nos concentrar em escrever nossos códigos e rodá-los ao final do processo.

Entenda que é possível programar em qualquer editor de texto, como o próprio bloco de notas ou em um terminal, o uso de IDEs se dá pela facilidade de possuir todo um espectro de ferramentas dedicadas em um mesmo lugar.

Principais IDEs

Como mencionado no tópico anterior, as IDEs buscam unificar as ferramentas necessárias para facilitar a vida do programador, claro que é possível programar a partir de qualquer bloco de notas, mas visando ter um ambiente completo onde se possa usar diversas ferramentas, testar e compilar seu código, o uso de uma IDE é altamente recomendado.

Existem diversas IDEs disponíveis no mercado, mas basicamente aqui irei recomendar duas delas.

Pycharm – A IDE Pycharm desenvolvida e mantida pela JetBrains, é uma excelente opção no sentido de que possui versão completamente gratuita, é bastante intuitiva e fácil de aprender a usar seus recursos e por fim ela oferece um ambiente com suporte em tempo real ao uso de console/terminal próprio, sendo possível a qualquer momento executar testes nos seus blocos de código.

Disponível em: https://www.jetbrains.com/pycharm/

Anaconda – Já a suíte Anaconda, também gratuita, conta com uma série de ferramentas que se destacam por suas aplicabilidades. Python é uma linguagem muito usada para data science, machine learning, e a suíte anaconda oferece softwares onde é possível trabalhar dentro dessas modalidades com ferramentas dedicadas a ela, além, é claro, do próprio ambiente de programação comum, que neste caso é o VisualStudioCode, e o Jupyter que é um terminal que roda via browser.

Disponível em: https://www.anaconda.com/download/

Importante salientar também que, é perfeitamente normal e possível programar direto em terminal, seja em Linux, seja em terminais oferecidos pelas próprias IDEs, a escolha de usar um ou outro é de particularidade do programador. É possível inclusive usar um terminal junto ao próprio editor da IDE.

Como dito anteriormente, existem diversas IDEs e ferramentas que facilitarão sua vida como programador, a verdade é que o interessante mesmo você dedicar um tempinho a testar as duas e ver qual você se adapta melhor.

Não existe uma IDE melhor que a outra, o que existem são ferramentas que se adaptam melhor as suas necessidades enquanto programador.

LÓGICA DE PROGRAMAÇÃO

Como já mencionei em um tópico anterior, uma linguagem de programação é uma linguagem como qualquer outra em essência, o diferencial é que na programação são meios que criamos para conseguir passar comandos a serem executados em um computador.

Quando falamos sobre um determinado assunto, automaticamente em nossa língua falada nativa geramos uma sentença que possui uma certa sintaxe e lógica de argumento para que a outra pessoa entenda o que estamos querendo transmitir. Se não nos expressarmos de forma clara podemos não ser entendidos ou o pior, ser mal-entendidos.

Um computador é um mecanismo exato, ele não tem (ainda) a capacidade de discernimento e abstração que possuímos, logo, precisamos passar suas instruções de forma literal e ordenada, para que a execução de um processo seja correta.

Quando estudamos lógica de programação, estudamos métodos de criar sequências lógicas de instrução, para que possamos usar tais sequências para programar qualquer dispositivo para que realize uma determinada função. Essa sequência lógica recebe o nome de algoritmo.

Algoritmos

Todo estudante de computação no início de seu curso recebe essa aula, algoritmos. Algoritmos em suma nada mais é do que uma sequência de passos onde iremos passar uma determinada instrução a ser realizada.

Imagine uma receita de bolo, onde há informação de quais ingredientes serão usados e um passo a passo de que ordem cada ingrediente será adicionado e misturado para que no final do processo o bolo dê certo, seja comestível e saboroso.

Um algoritmo é exatamente a mesma coisa, conforme temos que programar uma determinada operação, temos que passar as instruções passo a passo para que o interpretador consiga as executar e chegar ao fim de um processo.

Quando estudamos algoritmos basicamente temos três tipos básicos de algoritmo. Os que possuem uma entrada e geram uma saída, os que não possuem entrada, mas geram saída e os que possuem entrada, mas não geram saída.

Parece confuso neste primeiro momento, mas calma, imagine que na sua receita de bolo você precise pegar ingrediente por ingrediente e misturar, para que o bolo fique pronto depois de assado (neste caso temos entradas que geram uma saída.).

Em outro cenário imagine que você tem um sachê com a mistura já pronta de ingredientes, bastando apenas adicionar leite e colocar para assar (neste caso, não temos as entradas, mas temos a saída).

Por fim imagine que você é a empresa que produz o sachê com os ingredientes já misturados e pré-prontos, você tem os ingredientes e produz a mistura, porém não é você que fará o bolo (neste caso temos as entradas e nenhuma saída).

Em nossos programas haverão situações onde iremos criar scripts que serão executados ou não de acordo com as entradas necessárias e as saídas esperadas para resolver algum tipo de problema computacional.

Sintaxe em Python

Você provavelmente não se lembra do seu processo de alfabetização, quando aprendeu do zero a escrever suas primeiras letras, sílabas, palavras até o momento em que interpretou sentenças inteiras de acordo com a lógica formal de escrita, e muito provavelmente hoje o faz de forma automática apenas transliterando seus pensamentos.

Costumo dizer que aprender uma linguagem de programação é muito parecido, você literalmente estará aprendendo do zero um meio de "se comunicar" com o computador de forma a lhe passar instruções a serem executadas dentro de uma ordem e de uma lógica.

Toda linguagem de programação tem sua maneira de transliterar a lógica de um determinado algoritmo em uma linguagem característica que será interpretada "pelo computador", isto chamamos de sintaxe.

Toda linguagem de programação tem sua sintaxe característica, o programador deve respeitá-la até porque o interpretador é uma camada de software o qual é "programado" para entender apenas o que está sob a sintaxe correta. Outro ponto importante é que uma vez que você domina a sintaxe da linguagem ao qual está trabalhando, ainda assim haverão erros em seus códigos devido a erros de lógica.

Não se preocupe porque a coisa mais comum, até mesmo para programadores experientes é a questão de vez em quando cometer algum pequeno erro de sintaxe, ou não conseguir transliterar seu algoritmo numa lógica correta.

A sintaxe na verdade será entendida ao longo deste livro, até porque cada tópico que será abordado, será algum tipo de instrução que estaremos aprendendo e haverá uma forma correta de codificar tal instrução para que finalmente seja interpretada.

Já erros de lógica são bastante comuns nesse meio e, felizmente ou infelizmente, uma coisa depende da outra. Raciocine que você tem uma ideia, por exemplo, de criar uma simples calculadora.

De nada adianta você saber programar e não entender o

funcionamento de uma calculadora, ou o contrário, você saber toda lógica de funcionamento de uma calculadora e não ter ideia de como transformar isto em código.

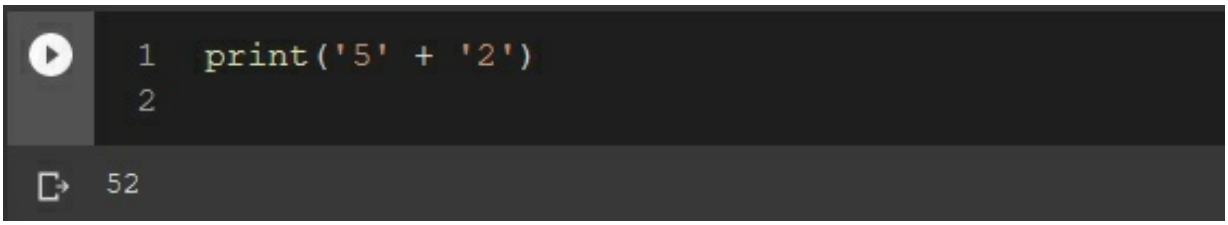
Portanto, iremos bater muito nessa tecla ao longo do livro, tentando sempre entender qual a lógica por trás do código e a codificação de seu algoritmo.

Para facilitar, nada melhor do que usarmos um exemplo prático de erro de sintaxe e de lógica respectivamente.

Supondo que queiramos exibir em tela a soma de dois números. 5 e 2, respectivamente.

*Esses exemplos usam de comandos que serão entendidos detalhadamente em capítulos posteriores, por hora, entenda que existe uma maneira certa de se passar informações para “a máquina”, e essas informações precisam obrigatoriamente ser em uma linguagem que faça sentido para o interpretador da mesma.

Ex 1:



O resultado será 52, porque de acordo com a sintaxe do Python 3, tudo o que digitamos entre ‘aspas’ é uma string (um texto), sendo assim o interpretador pegou o texto ‘5’ e o texto ‘2’ e, como são dois textos, os concatenou de acordo com o símbolo de soma.

Em outras palavras, aqui o nosso objetivo de somar 5 com 2 não foi realizado com sucesso porque passamos a informação para “a máquina” de maneira errada. Um erro típico de lógica.

Ex 2:

```
1  print(5 + 2)
2
```

```
7
```

O resultado é 7, já que o interpretador pegou os dois valores inteiros 5 e 2 e os somou de acordo com o símbolo +, neste caso um operador matemático de soma. Sendo assim, codificando da maneira correta o interpretador consegue realizar a operação necessária.

Ex 3:

```
1  Print(5) + 2 =
2
```

```
  File "<ipython-input-3-3834e23c81ab>", line 1
    Print(5) + 2 =
                 ^
SyntaxError: invalid syntax
```

O interpretador irá acusar um erro de sintaxe nesta linha do código pois ele não reconhece Print (iniciado em maiúsculo) e também não entende quais são os dados a serem usados e seus operadores.

Um erro típico de sintaxe, uma vez que não estamos passando as informações de forma que o interpretador espera para poder interpretá-las adequadamente.

Em suma, como dito anteriormente, temos que ter bem definida a ideia do que queremos criar, e programar de forma que o interpretador consiga receber e trabalhar com essas informações.

Sempre que nos depararmos com a situação de escrever alguma linha ou bloco de código e nosso interpretador acusar algum erro, devemos revisar o código em busca de que tipo de erro foi esse (lógica ou sintaxe) e procurar corrigir no próprio código.

Palavras reservadas

Na linguagem Python existem uma série de palavras reservadas para o sistema, ou seja, são palavras chave que o interpretador busca e usa para receber instruções a partir delas.

Para ficar mais claro vamos pegar como exemplo a palavra print, em Python print é um comando que serve para exibir em tela ou em console um determinado dado ou valor, sendo assim, é impossível criarmos uma variável com nome print, pois esta é uma palavra reservada ao sistema.

Ao todo são 31 palavras reservadas na sintaxe.

and del from not while

as elif global or with

assert else if pass yield

break except import print

class exec in raise

continue finally is return

def for lambda try

Repare que todas palavras utilizadas são termos em inglês, como Python é uma linguagem de alto nível, ela usa uma linguagem bastante próxima do usuário, com conhecimento básico de inglês é possível traduzir e interpretar o que cada palavra reservada faz.

À medida que formos progredindo você irá automaticamente associar que determinadas “palavras” são comandos ou instruções internas a linguagem. Você precisa falar uma língua que a máquina possa reconhecer, para que no final das contas suas instruções sejam entendidas, interpretadas e executadas.

Análise léxica

Quando estamos programando estamos colocando em prática o que planejamos anteriormente sob a ideia de um algoritmo. Algoritmos por si só são sequências de instruções que iremos codificar para serem interpretadas e executar uma determinada função. Uma das etapas internas do processo de programação é a chamada análise léxica.

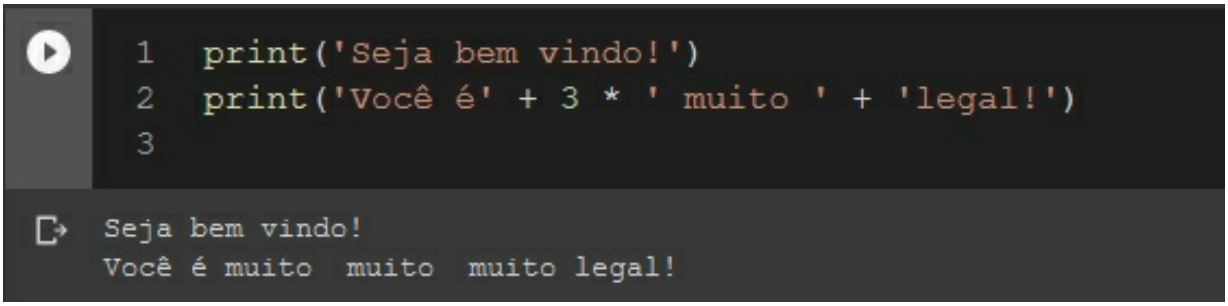
Raciocine que todo e qualquer código deve seguir uma sequência lógica, um passo-a-passo a ser seguido em uma ordem definida, basicamente quando estamos programando dividimos cada passo de nosso código em uma linha nova/diferente.

Sendo assim, temos que respeitar a ordem e a sequência lógica dos fatos para que eles sempre ocorram na ordem certa.

O interpretador segue fielmente cada linha e seu conteúdo, uma após a outra, uma vez que “ele” é uma camada de programa sem capacidade de raciocínio e interpretação como nós humanos. Sendo assim devemos criar cada linha ou bloco de código em uma sequência passo-a-passo que faça sentido para o interpretador.

Por fim, é importante ter em mente que o interpretador sempre irá ler e executar, de forma sequencial, linha após linha e o conteúdo de cada linha sempre da esquerda para direita.

Por exemplo:



Como mencionamos anteriormente, o interpretador sempre fará a leitura das linhas de cima para baixo. Ex: linha 1, linha 2, linha 3, etc... e o interpretador sempre irá ler o conteúdo da linha da esquerda para direita.

Nesse caso o retorno que o usuário terá é:

Seja bem vindo!

Você é muito muito muito legal.

Repare que pela sintaxe a linha 2 do código tem algumas particularidades, existem 3 strings e um operador mandando repetir 3 vezes uma delas, no caso 3 vezes ‘muito’, e como são strings, o símbolo de + está servindo apenas para concatená-las.

Embora isso ainda não faça sentido para você, fique tranquilo pois iremos entender cada ponto de cada linha de código posteriormente.

Indentação

Python é uma linguagem de forte indentação, ou seja, para fácil sintaxe e leitura, suas linhas de código não precisam necessariamente de uma pontuação, mas de uma tabulação correta. Quando linhas/blocos de código são filhos de uma determinada função ou parâmetro, devemos organizá-los de forma a que sua tabulação siga um determinado padrão.

Diferente de outras linguagens de programação que o interpretador percorre cada sentença e busca uma pontuação para demarcar seu fim, em Python o interpretador usa uma indentação para conseguir ver a hierarquia das sentenças.

O interpretador de Python irá considerar linhas e blocos de código que estão situados na mesma tabulação (margem) como blocos filhos do mesmo objeto/parâmetro. Para ficar mais claro, vejamos dois exemplos, o primeiro com indentação errada e o segundo com a indentação correta:

Indentação errada:

```
1  variavel1 = input('Digite um número: ')
2  if variavel1 >= str(0):
3  print('Número Positivo.')
4  print(f'O número é: {variavel1}')
5  else:
6  print('Número Negativo')
7  print(f'O número é: {variavel1}')
8
9
```

```
  File "<ipython-input-8-704c0bef8916>", line 3
    print('Número Positivo.')
        ^
IndentationError: expected an indented block
```

Repare que neste bloco de código não sabemos claramente que linhas de código são filhas e nem de quem... e mais importante que isto, com

indentação errada o interpretador não saberá quais linhas ou blocos de código são filhas de quem, não conseguindo executar e gerar o retorno esperado, gerando erro de sintaxe.

Indentação correta:

```
variavel1 = input('Digite um número: ')

if variavel1 >= str(0):
    print('Número Positivo.')
    print(f'O número é: {variavel1}')
else:
    print('Número Negativo')
    print(f'O número é: {variavel1}')

```

```
Digite um número: 15
Número Positivo.
O número é: 15
```

Agora repare no mesmo bloco de código, mas com as indentações corretas, podemos ver de acordo com as margens e espaçamentos quais linhas de código são filhas de quais comandos ou parâmetros.

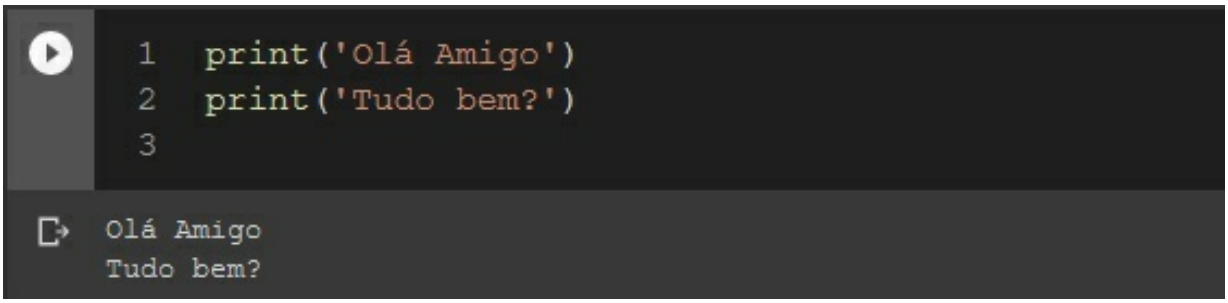
O mesmo ocorre por parte do interpretador, de acordo com a tabulação usada, ele consegue perceber que inicialmente existe uma variável variavel1 declarada que pede que o usuário digite um número, também que existem duas estruturas condicionais if e else, e que dentro delas existem instruções de imprimir em tela o resultado de acordo com o que o usuário digitou anteriormente e foi atribuído a variavel1.

Se você está vindo de outra linguagem de programação como C ou uma de suas derivadas, você deve estar acostumado a sempre encerrar uma sentença com ponto e vírgula ; Em Python não é necessário uma pontuação, mas se por ventura você inserir, ele simplesmente irá reconhecer que ali naquele ponto e vírgula se encerra uma instrução.

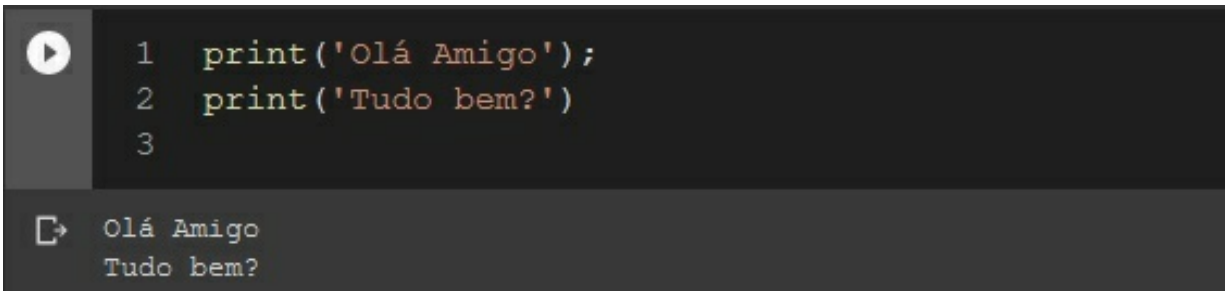
Isto é ótimo porque você verá que você volta e meia cometerá algumas exceções usando vícios de programação de outras linguagens que

em Python ao invés de gerarmos um erro, o interpretador saberá como lidar com essa exceção. Ex:

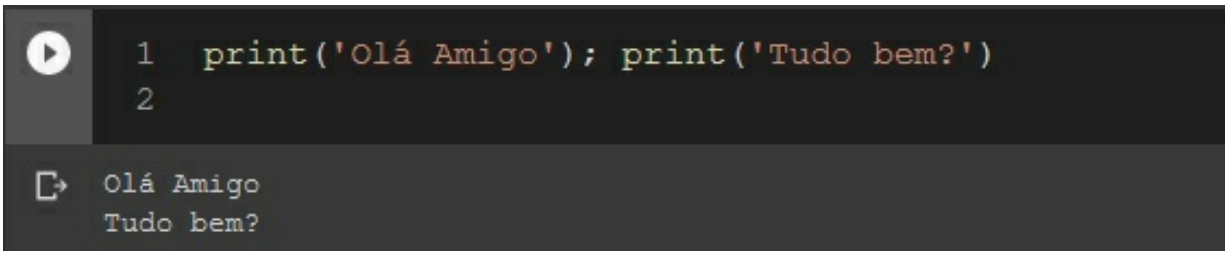
Código correto:



Código com vício de linguagem:



Código extrapolado:



Nos três casos o interpretador irá contornar o que for vício de linguagem e entenderá a sintaxe prevista, executando normalmente os comandos print( ).

O retorno será:

Olá Amigo

Tudo bem?

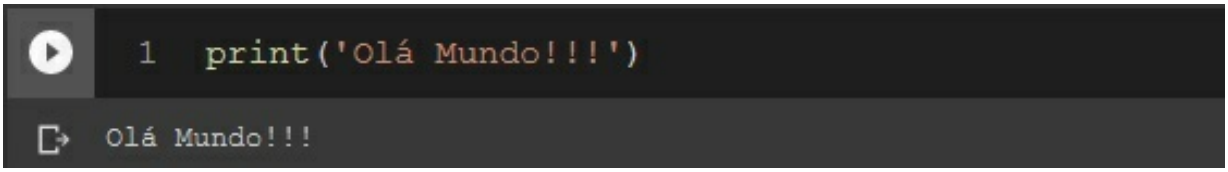
ESTRUTURA BÁSICA DE UM PROGRAMA

Enquanto em outras linguagens de programação você tem de criar toda uma estrutura básica para que realmente você possa começar a programar, Python já nos oferece praticamente tudo o que precisamos pré-carregado de forma que ao abrirmos uma IDE nossa única preocupação inicial é realmente programar.

Python é uma linguagem “batteries included”, termo em inglês para (pilhas inclusas), ou seja, ele já vem com o necessário para seu funcionamento pronto para uso. Posteriormente iremos implementar novas bibliotecas de funcionalidades em nosso código, mas é realmente muito bom você ter um ambiente de programação realmente pronto para uso.

Que tal começarmos pelo programa mais básico do mundo. Escreva um programa que mostre em tela a mensagem “Olá Mundo”. Fácil não?

Vamos ao exemplo:



Sim, por mais simples que pareça, isto é tudo o que você precisará escrever para que de fato, seja exibida a mensagem Olá Mundo!!! na tela do usuário.

Note que existe no início da sentença um print( ), que é uma palavra reservada ao sistema que tem a função de mostrar algo no console ou na tela do usuário, print( ) sempre será seguido de ( ) parênteses, pois dentro deles estará o que chamamos de argumentos/parâmetros dessa função, neste caso, uma string (frase) ‘Olá Mundo!!!’, toda frase, para ser reconhecida como tal, deve estar entre aspas.

Então fazendo o raciocínio lógico desta linha de código, chamamos a função print( ) que recebe como argumento ‘Olá Mundo!!!’.

O retorno será:

Olá Mundo!!!

Em outras linguagens de programação você teria de importar bibliotecas para que fosse reconhecido mouse, teclado, suporte a entradas e saída em tela, definir um escopo, criar um método que iria chamar uma determinada função, etc... etc... etc...

Em Python basta já de início dar o comando que você quer para que ele já possa ser executado. O interpretador já possui pré-carregado todos recursos necessários para identificar uma função e seus métodos, assim como os tipos de dados básicos que usaremos e todos seus operadores.

TIPOS DE DADOS

Independentemente da linguagem de programação que você está aprendendo, na computação em geral trabalhamos com dados, e os classificamos conforme nossa necessidade.

Para ficar mais claro, raciocine que na programação precisamos separar os dados quanto ao seu tipo. Por exemplo uma string, que é o termo reservado para qualquer tipo de dado alfanumérico (qualquer tipo de palavra/texto que contenha letras e números).

Já quando vamos fazer qualquer operação aritmética precisamos tratar os números conforme seu tipo, por exemplo o número 8, que para programação é um int (número inteiro), enquanto o número 8.2 é um float (número com casa decimal).

O ponto que você precisa entender de imediato é que não podemos misturar tipos de dados diferentes em nossas operações, porque o interpretador não irá conseguir distinguir que tipo de operação você quer realizar uma vez que ele faz uma leitura léxica e “literal” dos dados.

Por exemplo: Podemos dizer que “Maria tem 8 anos”, e nesse contexto, para o interpretador, o 8 é uma string, é como qualquer outra palavra dessa mesma sentença. Já quando pegamos dois números para somá-los por exemplo, o interpretador espera que esses números sejam int ou float, mas nunca uma string.

Parece muito confuso de imediato, mas com os exemplos que iremos posteriormente abordar você irá de forma automática diferenciar os dados quanto ao seu tipo e seu uso correto.

Segue um pequeno exemplo dos tipos de dados mais comuns que usaremos em Python:

Tipo	Descrição	Exemplo
Int	Número real inteiro, sem casas decimais	12

Float	Número com casas decimais	12.8
Bool	Booleano / Binário (0 ou 1)	True ou False / o ou 1
String	Texto com qualquer caractere alfanumérico	‘palavra’ “marca d’água”
List	Listas(s)	[2, ‘Pedro’, 15.9]
Dict	Dicionário(s)	{‘nome’:’João da Silva’, ‘idade’: 32}

Repare também que cada tipo de dado possui uma sintaxe própria para que o interpretador os reconheça como tal. Vamos criar algumas variáveis (que veremos no capítulo a seguir) apenas para atribuir diferentes tipos de dados, de forma que possamos finalmente visualizar como cada tipo de dados deve ser representado.

```
variavel_tipo_string = 'Conjunto de caracteres alfanuméricos'

variavel_tipo_int = 12     #número inteiro

variavel_tipo_float = 15.3    #número com casas decimais

variavel_tipo_bool = True     #Booleano (verdadeiro ou falso)

variavel_tipo_lista = [1, 2, 'Paulo', 'Maria', True]

variavel_tipo_dicionario = {'nome':'Fernando', 'idade':31,}

variavel_tipo_tupla = ('Ana', 'Fernando', 3)

variavel_tipo_comentario = """Um comentário atribuído a uma variável deixa de ser apenas um comentário, vira frase!!!"""
```

```python
variavel_tipo_string = 'Conjunto de caracteres alfanuméricos'

variavel_tipo_int = 12      #número inteiro

variavel_tipo_float = 15.3      #número com casas decimais

variavel_tipo_bool = True      #Booleano (verdadeiro ou falso)

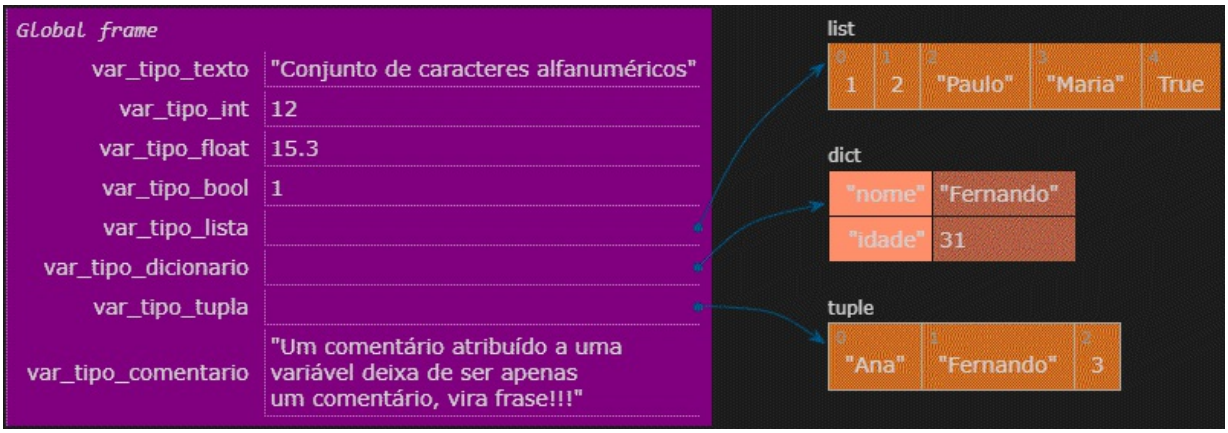
variavel_tipo_lista = [1, 2, 'Paulo', 'Maria', True]

variavel_tipo_dicionario = {'nome':'Fernando', 'idade':31,}

variavel_tipo_tupla = ('Ana', 'Fernando', 3)

variavel_tipo_comentario = """Um comentário atribuído a uma variável deixa
                           de ser apenas um comentário, vira frase!!!"""
```

Representação visual:



COMENTÁRIOS

Desculpe a redundância, mas comentários dentro das linguagens de programação servem realmente para comentar determinados blocos de código, para criar anotações sobre o mesmo.

É uma prática bastante comum à medida que implementamos novas funcionalidades em nosso código ir comentando-o também, para facilitar nosso entendimento quando revisarmos o mesmo.

Essa prática também é bastante comum quando pegamos códigos de domínio público, normalmente o código virá com comentários do seu autor explicando os porquês de determinadas linhas de código.

A sintaxe para comentar nossos códigos em Python é bastante simples, basicamente existem duas maneiras de comentar o código, quando queremos fazer um comentário que não será maior do que uma linha usaremos o símbolo # e em seguida o devido comentário.

Já quando precisamos fazer algum comentário mais elaborado, que irá ter mais de uma linha, usaremos ‘’’aspas triplas antes de nosso comentário e depois dele para o terminar’’’.

A ideia de comentar determinados blocos de código é uma maneira do programador colocar anotações sobre determinadas linhas de código. Tenha em mente que o interpretador não lê o que o usuário determinou como comentário, tudo o que estiver após # ou entre ‘’’ o interpretador irá simplesmente ignorar.

Vamos ao exemplo:

```
1  nome = 'Maria'
2  #Maria é a nova funcionária
3
4  print('Bem vinda Maria!!!')
5  #acima está uma mensagem de boas vindas a ela.
6
```

```
Bem vinda Maria!!!
```

Exemplo 2:

```
1  '''Este programa está sendo escrito
2  para que Maria, a nova funcionária,
3  comece a se integrar com o sistema.'''
4
```

```
'Este programa está sendo escrito\npara que Maria, a nova funcionária,
a se integrar com o sistema.'
```

Neste exemplo acima existem comentários dos dois tipos, o interpretador irá fazer apenas a leitura da variável nome e irá executar o comando print( ), ignorando todo o resto.

Porém se por ventura você quiser fazer com que um comentário passe a ser lido em seu código o mesmo deverá ser associado a uma variável.

```
1  '''Exemplo de comentário
2      este, inclusive, não será
3      lido pelo interpretador'''
4
```

Comentário interno, não lido pelo interpretador e não visível pelo usuário. Apenas comentário para alguma linha/bloco de código.

```
1  comentario1 = '''Exemplo de comentário
2                  e agora sim será lido
3                  pelo interpretador!!!'''
4  print(comentario1)
5
```

```
Exemplo de comentário
e agora sim será lido
pelo interpretador!!!
```

Comentário que será exibido ao usuário, pois está associado a uma variável e o comando print( ) está executando sua exibição.

O retorno será: Exemplo de comentário

e agora sim será lido

pelo interpretador!!!

Uma prática bastante comum é, enquanto testamos nossos blocos de código usar do artifício de comentário para isolar algum elemento, uma vez que este quando comentado passa a ser ignorado pelo interpretador.

```
1  var1 = 2019
2  var2 = 2020
3
4  soma = var1 + var2
5
6  print(soma)
7
```

```
4039
```

Repare que o código acima por mais básico que seja possui duas variáveis var1 e var2, uma operação de soma entre elas e um comando print( ) que irá exibir em tela o resultado de soma.

Se quisermos por exemplo, apenas a fim de testes, ignorar a operação soma declarada no código e realizar esta operação diretamente dentro do comando print( ), isto é perfeitamente possível. Bastando "comentar" a função soma, ela passa a ser ignorada pelo interpretador.

```
1  var1 = 2019
2  var2 = 2020
3  var3 = 2021
4
5  #soma = var1 + var2
6
7  print(var1 + var2)
8
```

```
4039
```

O retorno em ambos os casos será: 4039

Note que no segundo bloco de código, ao inserir o marcador # a frente da função soma, a transformamos em um simples comentário, ignorado pelo interpretador, para traze-la de volta à ativa basta "descomentar" a mesma.

VARIÁVEIS / OBJETOS

Uma variável basicamente é um espaço alocado na memória ao qual iremos armazenar um dado, valor ou informação, simples assim.

Imagine que você tem uma escrivaninha com várias gavetas, uma variável é uma dessas gavetas ao qual podemos guardar dentro dela qualquer coisa (qualquer tipo de dado) e ao mesmo tempo ter acesso fácil a este dado sempre que necessário durante a execução de nosso programa.

Python é uma linguagem dinamicamente tipada, ou seja, quando trabalhamos com variáveis/objetos (itens aos quais iremos atribuir dados ou valores), podemos trabalhar livremente com qualquer tipo de dado e se necessário, também alterar o tipo de uma variável a qualquer momento.

Outra característica importante de salientar neste momento é que Python, assim como outras linguagens, ao longo do tempo foi sofrendo uma série de mudanças que trouxeram melhorias em sua forma de uso. Na data de publicação deste livro estamos usando a versão 3.7 da linguagem Python, onde se comparado com as versões anteriores da mesma, houve uma série de simplificações em relação a maneira como declaramos variáveis, definimos funções, iteramos dados.

Por fim apenas raciocine que não iremos estar nos focando em sintaxe antiga ou que está por ser descontinuada, não há sentido em nos atermos a isto, todo e qualquer código que será usado neste livro respeitará a sintaxe mais atual.

Declarando uma variável

A declaração básica de uma variável/objeto sempre seguirá uma estrutura lógica onde, toda variável deve ter um nome (desde que não seja uma palavra reservada ao sistema) e algo atribuído a ela (qualquer tipo de dado ou valor).

Partindo diretamente para prática, vamos ver um exemplo de declaração de uma variável:

```
1  variavel1 = 11
2
```

Neste caso, inicialmente declaramos uma variável de nome variavel1 que por sua vez está recebendo o valor 11 como valor (o símbolo de = aqui é utilizado para atribuir um valor a variável).

No contexto de declaração de variáveis, o símbolo de igualdade não está comparando ou igualando os lados, mas está sendo usado para atribuir um dado ou valor a uma variável.

```
1  variavel1 = 11
2
3  print(variavel1)
4
```

Aqui, inicialmente apenas para fins de exemplo, na segunda linha do código usamos o comando print( ) que dentro de seus “parênteses” está instanciando a variável que acabamos de criar, a execução dessa linha de código irá exibir em tela para o usuário o dado/valor que for conteúdo, que estiver atribuído a variável variavel1.

Neste caso o retorno será: 11

Conforme você progredir em seus estudos de programação você irá notar que é comum possuirmos várias variáveis, na verdade, quantas forem necessárias em nosso programa.

Não existe um limite, você pode usar à vontade quantas variáveis forem necessárias desde que respeite a sua sintaxe e que elas tenham um propósito no código.

Outro ponto importante é que quando atribuímos qualquer dado ou valor a uma variável o tipo de dado é implícito, ou seja, se você por exemplo atribuir a uma variável simplesmente um número 6, o interpretador automaticamente identifica esse dado como um int (dado numérico do tipo inteiro).

Se você inserir um ponto '.' seguido de outro número, 5 por exemplo, tornando esse número agora 6.5, o interpretador automaticamente irá reconhecer esse mesmo dado agora como float (número de ponto flutuante).

O mesmo ocorre quando você abre aspas ' ' para digitar algum dado a ser atribuído a uma variável, automaticamente o interpretador passará a trabalhar com aquele dado o tratando como do tipo string (palavra, texto ou qualquer combinação alfanumérica de caracteres).

Posteriormente iremos ver que também é possível declarar explicitamente o tipo de um dado e até convertê-lo de um tipo para outro.

Python também é uma linguagem case sensitive, ou seja, ela diferencia caracteres maiúsculos de minúsculos, logo, existem certas maneiras de declarar variáveis que são permitidas enquanto outras não, gerando conflito com o interpretador. A grosso modo podemos declarar variáveis usando qualquer letra minúscula e o símbolo "_" underline simples no lugar do espaço.

Exemplo 1: Declarando variáveis corretamente.

```
1  variavel1 = 'Ana'
2  variavel2 = 'Pedro'
3
4  variavel_1 = 'Ana'
5  variavel_2 = 'Pedro'
6
```

Ambos os modelos apresentam sintaxe correta, cabe ao usuário escolher qual modo ele considera mais fácil de identificar essa variável pelo nome.

Exemplo 2: Simulando erro, declarando uma variável de forma não permitida.

```
 1  variavel 1 = 'String'
 2  14 = 'Numero'
 3

 File "<ipython-input-5-47d7a087c1af>", line 1
   variavel 1 = 'String'
            ^
SyntaxError: invalid syntax
```

Nem variavel 1 nem a 14 serão reconhecidas pelo interpretador como variáveis porque estão escritas de forma não reconhecida por sua sintaxe.

No primeiro exemplo o espaço entre variável e 1 gera conflito com o interpretador. No segundo exemplo, um número nunca pode ser usado como nome para uma variável.

Embora permitido, não é recomendável usar o nome de uma variável todo em letras maiúsculas, por exemplo:

```
1  NOME = 'Fernando'
2
```

*Se você está vindo de outras linguagens de programação para o Python, você deve conhecer o conceito de que quando se declara uma variável com

letras maiúsculas ela se torna uma constante, uma variável imutável.

Esse conceito não existe para o Python porque nele as variáveis são tratadas como objetos e sempre serão dinâmicas. Por fim, fique tranquilo que usar essa sintaxe é permitida em Python, não muito recomendável, mas você não terá problemas em seu código em função disso.

Outro ponto importante de destacar é que em função do padrão case sensitive, ao declarar duas variáveis iguais, uma com caracteres maiúsculos e outra com caracteres minúsculos serão interpretadas como duas variáveis diferentes. Ex:

```
1  NOME = 'Fernando'
2  nome = 'Rafael'
3
```

Também é permitido, mas não é recomendado o uso de palavras com acentos: Ex:

```
1  variável = 'Maria'
2  cômodos = 3
3
```

Por fim, raciocine que o interessante é você criar o hábito de criar seus códigos usando as formas mais comuns de se declarar variáveis, deixando apenas para casos especiais essas exceções.

Exemplos de nomenclatura de variáveis permitidos:

```
variavel = 'Ana'

variavel1 = 'Ana'

variavel_1 = 'Ana'

var_num_1 = 'Ana'

minhavariavel = 'Ana'

minha_variavel = 'Ana'

minhaVariavel = 'Ana'

```

Permitidos, mas não recomendados:

```
variável = 'Ana'

VARIAVEL = 'Ana'

Variavel = 'Ana'

```

Não permitidos, por poder gerar conflitos com o interpretador de sua IDE:

```
1991 = 'Ana'

minha variavel = 'Ana'

1variavel = 'Ana'

```

*Existirão situações onde um objeto será declarado com letra maiúscula inicial, mas neste caso ele não é uma variável qualquer e seu modo de uso será totalmente diferente. Abordaremos esse tema nos capítulos finais do livro.

Vale lembrar também que por convenção é recomendado criar nomes

pequenos e que tenham alguma lógica para o programador, se você vai declarar uma variável para armazenar o valor de um número, faz mais sentido declarar uma variável numero1 do que algo tipo ag_23421_m_meuNumero...

Por fim, vale lembrar que, como visto em um capítulo anterior, existem palavras que são reservadas ao sistema, dessa forma você não conseguirá as usar como nome de uma variável.

Por exemplo while, que é uma chamada para uma estrutura condicional, sendo assim, é impossível usar “while” como nome de uma variável.

Declarando múltiplas variáveis

É possível, para deixar o código mais enxuto, declarar várias variáveis em uma linha, independentemente do tipo de dado a ser recebido, desde que se respeite a sintaxe correta de acordo com o tipo de dado e sua justaposição.

Por exemplo, o código abaixo:

```
1  nome = 'Maria'
2  idade = 32
3  sexo = 'F'
4  altura = 1.89
5
```

Pode ser agrupado em uma linha, da seguinte forma:

```
1  nome, idade, sexo, altura = 'Maria', 32, 'F', 1.89
2
```

A ordem será respeitada e serão atribuídos os valores na ordem aos quais foram citados. (primeira variável com primeiro atributo, segunda variável com segundo atributo, etc...)

Declarando múltiplas variáveis (de mesmo tipo)

Quando formos definir várias variáveis, mas que possuem o mesmo valor ou tipo de dado em comum, podemos fazer essa declaração de maneira resumida. Ex:

Método convencional:

```
1  num1 = 10
2  x = 10
3  a1 = 10
4
```

Método reduzido:

```
1  num1 = x = a1 = 10
2
```

Repare que neste caso, em uma linha de código foram declaradas 3 variáveis, todas associadas com o valor 10, o interpretador as reconhecerá como variáveis independentes, podendo o usuário fazer o uso de qualquer uma delas isoladamente a qualquer momento.

FUNÇÕES BÁSICAS

Funções, falando de forma bastante básica, são linhas ou blocos de códigos aos quais executarão uma determinada ação a partir de nosso código, seguindo suas instruções.

Inicialmente trabalharemos com as funções mais básicas que existem, responsáveis por exibir em tela uma determinada resposta e também responsáveis por interagir com o usuário.

Funções podem receber parâmetros de execução (ou não), dependendo a necessidade, uma vez que esses parâmetros nada mais são do que as informações de como os dados deverão interagir internamente para realizar uma determinada função.

Pela sintaxe Python, “chamamos” uma função pelo seu nome, logo em seguida, entre ( ) parênteses podemos definir seus parâmetros, instanciar variáveis ou escrever linhas de código à vontade, desde que não nos esqueçamos que este bloco de código fará parte (será de propriedade) desta função...

Posteriormente trataremos das funções propriamente ditas, como criamos funções personalizadas que realizam uma determinada ação, por hora, para conseguirmos dar prosseguimento em nossos estudos, precisamos entender que existem uma série de funções pré-programadas, prontas para uso, com parâmetros internos que podem ser implícitos ou explícitos ao usuário.

As mais comuns delas, print( ) e input( ) respectivamente nos permitirão exibir em tela o resultado de uma determinada ação de um bloco de código, e interagir com o usuário de forma que ele consiga por meio do teclado inserir dados em nosso programa.

Função print( )

Quando estamos criando nossos programas, é comum que de acordo com as instruções que programamos, recebamos alguma saída, seja ela alguma mensagem ou até mesmo a realização de uma nova tarefa.

Uma das saídas mais comuns é exibirmos, seja na tela para o usuário ou em console (em programas que não possuem uma interface gráfica), uma mensagem, para isto, na linguagem python usamos a função print( ).

Na verdade anteriormente já usamos ela enquanto estávamos exibindo em tela o conteúdo de uma variável, mas naquele capítulo esse conceito de fazer o uso de uma função estava lá apenas como exemplo e para obtermos algum retorno de nossos primeiros códigos, agora iremos de fato entender o mecanismo de funcionamento deste tipo de função.

Por exemplo:

```
1  print('Seja bem vindo!!!')
2
```

```
Seja bem vindo!!!
```

Repare na sintaxe: a função print( ) tem como parâmetro (o que está dentro de parênteses) uma string com a mensagem Seja bem vindo!!!

Todo parâmetro é delimitado por ( ) parênteses e toda string é demarcada por ‘ ‘ aspas para que o interpretador reconheça esse tipo de dado como tal.

O retorno dessa linha de código será: Seja bem vindo!!!

Função input( )

Em todo e qualquer programa é natural que haja interação do usuário com o mesmo, de modo que com algum dispositivo de entrada o usuário dê instruções ou adicione dados.

Começando pelo básico, em nossos programas a maneira mais rudimentar de captar os dados do usuário será por intermédio da função input( ), por meio dela podemos pedir, por exemplo que o usuário digite um dado ou valor, que internamente será atribuído a uma variável.

Ex:

```
1  nome = input('Digite o seu nome: ')
2  print('Bem Vindo', nome)
3
```

```
Digite o seu nome: Fernando
Bem Vindo Fernando
```

Inicialmente declaramos uma variável de nome nome que recebe como atributo a função input( ) que por sua vez dentro tem uma mensagem para o usuário.

Assim que o usuário digitar alguma coisa e pressionar a tecla ENTER, esse dado será atribuído a variável nome.

Em seguida, a função print( ) exibe em tela uma mensagem definida concatenada ao nome digitado pelo usuário e atribuído a variável nome.

O retorno será: Bem Vindo Fernando

*Supondo, é claro, que o usuário digitou Fernando.

Apenas um adendo, como parâmetro de nossa função print( ) podemos instanciar múltiplos tipos de dados, inclusive um tipo de dado mais de uma vez.

Apenas como exemplo, aprimorando o código anterior, podemos

adicionar mais de uma string ao mesmo exemplo, desde que se respeite a justaposição das mesmas. Ex:

```
1  nome = input('Digite o seu nome: ')
2  print('Bem Vindo', nome,'!!!')
3
```

```
Digite o seu nome: Fernando
Bem Vindo Fernando !!!
```

O retorno será: Bem Vindo Fernando !!!

Explorando a função print( )

Como mencionado anteriormente, existem muitas formas permitidas de se “escrever” a mesma coisa, e também existe a questão de que a linguagem Python ao longo dos anos foi sofrendo alterações e atualizações, mudando aos poucos sua sintaxe.

O importante de se lembrar é que, entre versões se aprimoraram certos pontos da sintaxe visando facilitar a vida do programador, porém, para programadores que de longa data já usavam pontos de uma sintaxe antiga, ela não necessariamente deixou de funcionar a medida que a linguagem foi sendo atualizada.

Raciocine que muitos dos códigos que você verá internet a fora estarão no padrão Python 2, e eles funcionam perfeitamente no Python, então, se você está aprendendo realmente do zero por meio deste livro, fique tranquilo que você está aprendendo com base na versão mais atualizada da linguagem.

Se você programava em Python 2 e agora está buscando se atualizar, fique tranquilo também porque a sintaxe que você usava não foi descontinuada, ela gradualmente deixará de ser usada pela comunidade até o ponto de poder ser removida do núcleo do Python, por hora, ambas sintaxes funcionam perfeitamente.

Vamos ver isso na prática:

print( ) básico – Apenas exibindo o conteúdo de uma variável:

```
1  nome1 = 'Maria'
2  print(nome1)
3
```

```
Maria
```

Inicialmente declaramos uma variável de nome nome1 que recebe como atributo ‘Maria’, uma string. Em seguida exibimos em tela o conteúdo

atribuído a nome1.

O retorno será: Maria

print( ) básico – Pedindo ao usuário que dê entrada de algum dado:

```
1  nome1 = input('Digite o seu nome: ')
2
3  print(nome1)
4
```

```
Digite o seu nome: Fernando
Fernando
```

Declaramos uma variável de nome nome1 que recebe como atributo a função input( ) que por sua vez pede ao usuário que digite alguma coisa.

Quando o usuário digitar o que for solicitado e pressionar a tecla ENTER, este dado/valor será atribuído a nome1. Da mesma forma que antes, por meio da função print( ) exibimos em tela o conteúdo de num1.

O retorno será: Fernando

*supondo é claro, que o usuário digitou Fernando.

print( ) intermediário – Usando máscaras de substituição (Sintaxe antiga):

```
1  nome1 = input('Digite o seu nome: ')
2  print('Seja bem vindo(a) %s' %(nome1))
3
```

```
Digite o seu nome: Fernando
Seja bem vindo(a) Fernando
```

Da mesma forma que fizemos no exemplo anterior, declaramos uma variável nome1 e por meio da função input( ) pedidos uma informação ao usuário. Agora, como parâmetro de nossa função print( ) temos uma mensagem (string) que reserva dentro de si, por meio do marcador %s, um

espaço a ser substituído pelo valor existente em nome1. Supondo que o usuário digitou Fernando.

O retorno será: Seja bem vindo(a) Fernando

Porém existem formas mais sofisticadas de realizar esse processo de interação, e isso se dá por o que chamamos de máscaras de substituição.

Máscaras de substituição foram inseridas na sintaxe Python com o intuito de quebrar a limitação que existia de poder instanciar apenas um dado em nossa string parâmetro de nossa função print( ).

Com o uso de máscaras de substituição e da função .format( ) podemos inserir um ou mais de um dado/valor a ser substituído dentro de nossos parâmetros de nossa função print( ).

```
1  nome1 = input('Digite o seu nome: ')
2
3  print('Seja bem vindo(a) {} !!!'.format(nome1))
4
```

```
Digite o seu nome: Fernando
Seja bem vindo(a) Fernando !!!
```

A máscara { } reserva um espaço dentro da string a ser substituída pelo dado/valor atribuído a nome1.

O retorno será: Seja bem vindo(a) Fernando !!!

Fazendo o uso de máscaras de substituição, como dito anteriormente, podemos instanciar mais de um dado/valor/variável dentro dos parâmetros de nossa função print( ).

```
1  nome1 = input('Digite o seu nome: ')
2  msg1 = 'Por favor entre'
3  msg2 = 'Você é o primeiro a chegar.'
4
5  print('Seja bem vindo {}, {}, {}'.format(nome1, msg1, msg2))
6
```

```
Digite o seu nome: Fernando
Seja bem vindo Fernando, Por favor entre, Você é o primeiro a chegar.
```

Repare que dessa vez temos mais duas variáveis msg1 e msg2 respectivamente que possuem strings como atributos.

Em nossa função print( ) criamos uma string de mensagem de boas-vindas e criamos 3 máscaras de substituição, de acordo com a ordem definida em .format( ) substituiremos a primeira pelo nome digitado pelo usuário, a segunda e a terceira máscara pelas frases atribuídas a msg1 e msg2.

O retorno será: Seja bem vindo Fernando, Por favor entre, Você é o primeiro a chegar.

print( ) avançado – Usando de f’strings (Sintaxe nova/atual):

Apenas dando um passo adiante, uma maneira mais moderna de se trabalhar com máscaras de substituição se dá por meio de f’strings, a partir da versão 3 do Python se permite usar um simples parâmetro “f” antes de uma string para que assim o interpretador subentenda que ali haverão máscaras de substituição a serem trabalhadas, independentemente do tipo de dado.

Dessa forma se facilitou ainda mais o uso de máscaras dentro de nossa função print( ) e outras em geral.

Basicamente declaramos um parâmetro “f” antes de qualquer outro e podemos instanciar o que quisermos diretamente dentro das máscaras de substituição, desde o conteúdo de uma variável até mesmo operações e funções dentro de funções, mas começando pelo básico, veja o exemplo:

```
1  nome = input('Digite o seu nome: ')
2  ap = input('Digite o número do seu apartamento: ')
3
4  print(f'Seja bem vindo(a) {nome}, morador(a) do ap n° {ap}')
5
```

```
Digite o seu nome: Fernando
Digite o número do seu apartamento: 134
Seja bem vindo(a) Fernando, morador(a) do ap n° 134
```

Supondo que o usuário digitou respectivamente Fernando e 134...

O retorno será: Seja bem vinda Maria, moradora do ap nº 33

Quando for necessário exibir uma mensagem muito grande, de mais de uma linha, uma forma de simplificar nosso código reduzindo o número de prints a serem executados é usar a quebra de linha dentro de um print( ), adicionando um \n frente ao texto que deverá estar posicionado em uma nova linha. Ex:

```
1  nome = 'João'
2  dia_vencimento = 10
3  valor_fatura = 149.90
4
5  print(f'Olá, caro {nome},\n Sua fatura vence dia {dia_vencimento} de janeiro')
6  print(f'\n O valor é de R${valor_fatura}.\n Favor pagar até o prazo para evitar multas.')
7
```

```
Olá, caro João,
 Sua fatura vence dia 10 de janeiro

 O valor é de R$149.9.
 Favor pagar até o prazo para evitar multas.
```

Repare que existe um comando print( ) com uma sentença enorme, que irá gerar 4 linhas de texto de retorno, combinando o uso de máscaras para sua composição.

O retorno será:

Olá, caro João,

Sua fatura vence dia 10 de janeiro

O valor é de R$149.9.

Favor pagar até o prazo para evitar multas.

O que você deve ter em mente, e começar a praticar, é que em nossos programas, mesmo os mais básicos, sempre haverá meios de interagir com o usuário, seja exibindo certo conteúdo para o mesmo, seja interagindo diretamente com ele de forma que ele forneça ou manipule os dados do programa.

Lembre-se que um algoritmo pode ter entradas, para execução de uma ou mais funções e gerar uma ou mais saídas. Estas entradas pode ser um dado importado, um link ou um arquivo instanciado, ou qualquer coisa que esteja sendo inserida por um dispositivo de entrada do computador, como o teclado que o usuário digita ou o mouse interagindo com alguma interface gráfica.

Interação entre variáveis

Agora entendidos os conceitos básicos de como fazer o uso de variáveis, como exibir seu conteúdo em tela por meio da função print( ) e até mesmo interagir com o usuário via função input( ), hora de voltarmos a trabalhar os conceitos de variáveis, aprofundando um pouco mais sobre o que pode ser possível fazer a partir delas.

A partir do momento que declaramos variáveis e atribuímos valores a elas, podemos fazer a interação entre elas (interação entre seus atributos), por exemplo:

```
1  num1 = 10
2  num2 = 5.2
3  soma = num1 + num2
4
5  print(soma)
6
7  print(f'O resultado é {soma}')
8
```

```
15.2
O resultado é 15.2
```

Inicialmente criamos uma variável num1 que recebe como atributo 10 e uma segunda variável num2 que recebe como atributo 5.2.

Na sequência criamos uma variável soma que faz a soma entre num1 e num2 realizando a operação instanciando as próprias variáveis. O resultado da soma será guardado em soma.

A partir daí podemos simplesmente exibir em tela via função print( ) o valor de soma, assim como podemos criar uma mensagem mais elaborada usando máscara de substituição. Dessa forma...

O retorno será: 15.2

Seguindo com o que aprendemos no capítulo anterior, podemos melhorar ainda mais esse código realizando este tipo de operação básica diretamente dentro da máscara de substituição. Ex:

```
1  num1 = 10
2  num2 = 5.2
3
4  print(f'O resultado é {num1 + num2}')
5  O retorno será: 15.2
6
```

```
15.2
O resultado é 15.2
```

Como mencionado anteriormente, o fato da linguagem Python ser dinamicamente tipada nos permite a qualquer momento alterar o valor e o tipo de uma variável.

Outro exemplo, a variável a que antes tinha o valor 10 (int) podemos a qualquer momento alterar para 'Maria' (string).

```
1  a = 10
2
3  print(a)
4
```

```
10
```

O resultado será: 10

```
1  a = 10
2  a = 'Maria'
3
4  print(a)
5
```

```
Maria
```

O resultado será: Maria.

A ordem de leitura por parte do interpretador faz com que o último dado/valor atribuído a variável a seja ‘Maria’.

Como explicado nos capítulos iniciais, a leitura léxica desse código respeita a ordem das linhas de código, ao alterarmos o dado/valor de uma variável, o interpretador irá considerar a última linha de código a qual se fazia referência a essa variável e seu último dado/valor atribuído. Sendo assim, devemos tomar cuidado quanto a sobrescrever o dado/valor de uma variável.

Por fim, quando estamos usando variáveis dentro de alguns tipos de operadores podemos temporariamente convertê-los para um tipo de dado, ou deixar mais explícito para o interpretador que tipo de dado estamos trabalhando para que não haja conflito.

Por exemplo:

```
1  num1 = 5
2  num2 = 8.2
3  soma = int(num1) + int(num2)
4
5  print(soma)
6
```

```
13
```

O resultado será: 13 (sem casas decimais, porque definimos na expressão de soma que num1 e num2 serão tratados como int, número inteiro, sem casas decimais).

Existe a possibilidade também de já deixar especificado de que tipo de dado estamos falando quando o declaramos em uma variável. Por exemplo:

```
1  num1 = int(5)
2  num2 = float(8.2)
3  soma = num1 + num2
4
5  print(soma)
6
```

```
13.2
```

A regra geral diz que qualquer operação entre um int e um float resultará em float.

O retorno será 13.2

Como você deve estar reparando, a sintaxe em Python é flexível, no sentido de que haverão várias maneiras de codificar a mesma coisa, deixando a escolha por parte do usuário. Apenas aproveitando o exemplo acima, duas maneiras de realizar a mesma operação de soma dos valores atribuídos as respectivas variáveis.

```
1  num1 = 5
2  num2 = 8.2
3  soma = int(num1) + int(num2)
4
5  # Mesmo que:
6  num1 = int(5)
7  num2 = float(8.2)
8  soma = num1 + num2
9
10 print(soma)
11
```

```
13.2
```

Por fim, é possível “transformar” de um tipo numérico para outro apenas alterando a sua declaração. Por exemplo:

```
1  num1 = int(5)
2  num2 = float(5)
3
4  print(num1)
5  print(num2)
6
```

```
5
5.0
```

O retorno será: 5

5.0

Note que no segundo retorno, o valor 5 foi declarado e atribuído a num2 como do tipo float, sendo assim, ao usar esse valor, mesmo inicialmente ele não ter sido declarado com sua casa decimal, a mesma aparecerá nas operações e resultados.

```
1  num1 = int(5)
2  num2 = int(8.2)
3  soma = num1 + num2
4
5  print(soma)
6
```

```
13
```

Mesmo exemplo do anterior, mas agora já especificamos que o valor de num2, apesar de ser um número com casa decimal, deve ser tratado como inteiro, e sendo assim:

O Retorno será: 13

E apenas concluindo o raciocínio, podemos aprimorar nosso código realizando as operações de forma mais eficiente, por meio de f’strings. Ex:

```
1   num1 = int(5)
2   num2 = int(8.2)
3
4   print(f'O resultado da soma é: {num1 + num2}')
5
```

```
O resultado da soma é: 13
```

O Retorno será: 13

Conversão de tipos de dados

É importante entendermos que alguns tipos de dados podem ser “misturados” enquanto outros não, quando os atribuímos a nossas variáveis e tentamos realizar interações entre as mesmas.

Porém existem recursos para elucidar tanto ao usuário quanto ao interpretador que tipo de dado é em questão, assim como podemos, conforme nossa necessidade, convertê-los de um tipo para outro.

A forma mais básica de se verificar o tipo de um dado é por meio da função type( ). Ex:

```
1  numero = 5
2
3  print(type(numero))
4
```

```
<class 'int'>
```

O resultado será: <class 'int'>

O que será exibido em tela é o tipo de dado, neste caso, um int.

Declarando a mesma variável, mas agora atribuindo 5 entre aspas, pela sintaxe, 5, mesmo sendo um número, será lido e interpretado pelo interpretador como uma string. Ex:

```
1  numero = '5'
2
3  print(type(numero))
4
```

```
<class 'str'>
```

O resultado será: <class 'str'>

Repare que agora, respeitando a sintaxe, ‘5’ passa a ser uma string, um “texto”.

Da mesma forma, sempre respeitando a sintaxe, podemos verificar o tipo de qualquer dado para nos certificarmos de seu tipo e presumir que funções podemos exercer sobre ele. Ex:

```
1  numero = [5]
2
3  print(type(numero))
4
```

```
<class 'list'>
```

O retorno será: <class 'list'>

```
1  numero = {5}
2
3  print(type(numero))
4
```

```
<class 'set'>
```

O retorno será: <class 'set'>

*Lembre-se que a conotação de chaves { } para um tipo de dado normalmente faz dele um dicionário, nesse exemplo acima o tipo de dado retornado foi ‘set’ e não ‘dict’ em função de que a notação do dado não é, como esperado, uma chave : valor.

Seguindo esta lógica e, respeitando o tipo de dado, podemos evitar erros de interpretação fazendo com que todo dado ou valor atribuído a uma variável já seja identificado como tal. Ex:

```
1  frase1 = str('Raquel tem 15 anos')
2
3  print(type(frase1))
4
```

```
<class 'str'>
```

Neste caso antes mesmo de atribuir um dado ou valor a variável frase1 já especificamos que todo dado contido nela é do tipo string.

Executando o comando print(type(frase)) o retorno será: <class 'str'>

De acordo com o tipo de dados certas operações serão diferentes quanto ao seu contexto, por exemplo tendo duas frases atribuídas às suas respectivas variáveis, podemos usar o operador + para concatená-las (como são textos, soma de textos não existe, mas sim a junção entre eles). Ex:

```
1  frase1 = str('Raquel tem 15 anos, ')
2  frase2 = str('de verdade')
3
4  print(frase1 + frase2)
5
```

```
Raquel tem 15 anos, de verdade
```

O resultado será: Raquel tem 15 anos, de verdade.

Já o mesmo não irá ocorrer se misturarmos os tipos de dados, por exemplo:

```
1  frase1 = str('Raquel tem ')
2  frase2 = int(15)
3  frase3 = 'de verdade'
4
5  print(frase1 + frase2 + frase3)
6
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call
<ipython-input-37-3e32395acec3> in <module>()
      3 frase3 = 'de verdade'
      4
----> 5 print(frase1 + frase2 + frase3)

TypeError: must be str, not int
```

O retorno será um erro de sintaxe, pois estamos tentando juntar diferentes tipos de dados.

Corrigindo o exemplo anterior, usando as 3 variáveis como de mesmo tipo o comando print( ) será executado normalmente.

```
1  frase1 = str('Raquel tem ')
2  frase2 = str(15)
3  frase3 = ' de verdade'
4
5  print(frase1 + frase2 + frase3)
6
```

```
Raquel tem 15 de verdade
```

O retorno será: Raquel tem 15 de verdade.

Apenas por curiosidade, repare que o código apresentado nesse último exemplo não necessariamente está usando f’strings porque a maneira mais prática de o executar é instanciando diretamente as variáveis como parâmetro em nossa função print( ).

Já que podemos optar por diferentes opções de sintaxe, podemos

perfeitamente fazer o uso da qual considerarmos mais prática.

```
1  print(frase1 + frase2 + frase3)
2
3  # Mesmo que:
4  print(f'{frase1 + frase2 + frase3}')
5
```

```
Raquel tem 15 de verdade
Raquel tem 15 de verdade
```

Nesse exemplo em particular o uso de f’strings está aumentando nosso código em alguns caracteres desnecessariamente.

Em suma, sempre preste muita atenção quanto ao tipo de dado e sua respectiva sintaxe, 5 é um int enquanto ‘5’ é uma string.

Se necessário, converta-os para o tipo de dado correto para evitar erros de interpretação de seu código.

OPERADORES

Operadores de Atribuição

Em programação trabalharemos com variáveis/objetos que nada mais são do que espaços alocados na memória onde iremos armazenar dados, para durante a execução de nosso programa, fazer o uso deles.

Esses dados, independentemente do tipo, podem receber uma nomenclatura personalizada e particular que nos permitirá ao longo do código os referenciar ou incorporar dependendo a situação.

Para atribuir um determinado dado/valor a uma variável teremos um operador que fará esse processo.

A atribuição padrão de um dado para uma variável, pela sintaxe do Python, é feita através do operador =, repare que o uso do símbolo de igual “ = ” usado uma vez tem a função de atribuidor, já quando realmente queremos usar o símbolo de igual para igualar operandos, usaremos ele duplicado “ == “.

Por exemplo:

```
1  salario = 955
2
```

Nesta linha de código temos declaramos a variável salario que recebe como valor 955, esse valor nesse caso é fixo, e sempre que referenciarmos a variável salario o interpretador usará seu valor atribuído, 955.

Uma vez que temos um valor atribuído a uma variável podemos também realizar operações que a referenciem, por exemplo:

```
1  salario = 955
2  aumento1 = 27
3
4  print(salario + aumento1)
5

982
```

O resultado será 982, porque o interpretador pegou os valores das variáveis salario e aumento1 e os somou.

Por fim, também é possível fazer a atualização do valor de uma variável, por exemplo:

```
1  mensalidade = 229
2  mensalidade = 229 + 10
3
4  print(mensalidade)
5

239
```

O resultado será 239, pois o último valor atribuído a variavel mensalidade, atualizando a mesma, era 229 + 10.

Aproveitando o tópico, outra possibilidade que temos, já vimos anteriormente, e na verdade trabalharemos muito com ela, é a de solicitar que o usuário digite algum dado ou algum valor que será atribuído a variável, podendo assim, por meio do operador de atribuição, atualizar o dado ou valor declarado inicialmente, por exemplo:

```
1  nome = 'sem nome'
2  idade = 0
3
4  nome = input('Por favor, digite o seu nome: ')
5  idade = input('Digite a sua idade: ')
6
7  print(nome, idade)
8

Por favor, digite o seu nome: Fernando
Digite a sua idade: 33
Fernando 33
```

Repare que inicialmente as variáveis nome e idade tinham valores padrão pré-definidos, ao executar esse programa será solicitado que o usuário digite esses dados.

Supondo que o usuário digitou Fernando, a partir deste momento a variável nome passa a ter como valor Fernando.

Na sequência o usuário quando questionado sobre sua idade irá digitar números, supondo que digitou 33, a variável idade a partir deste momento passa a ter como atribuição 33.

Internamente ocorre a atualização dessa variável para esses novos dados/valores atribuídos.

O retorno será: Fernando 33

Atribuições especiais

Atribuição Aditiva:

```
1  variavel1 = 4
2  variavel1 = variavel1 + 5
3
4  # Mesmo que:
5  variavel1 += 5
6
7  print(variavel1)
8
```

```
14
```

Com esse comando o usuário está acrescentando 5 ao valor de variavel1 que inicialmente era 4. Sendo 4 + 5:

O resultado será 9.

Atribuição Subtrativa:

```
variavel1 = 4
resultado = variavel1 - 3

print(resultado)

# Mesmo que:
variavel2 = 4
variavel2 -= 3

print(variavel2)

```

```
1
1
```

Nesse caso, o usuário está subtraindo 3 de variavel1. Sendo 4 – 3:

O resultado será 1.

Atribuição Multiplicativa:

```
variavel1 = 4
resultado = variavel1 * 2

print(resultado)

# Mesmo que:
variavel2 = 4
variavel2 *= 2

print(variavel2)

```

```
8
8
```

Nesse caso, o usuário está multiplicando o valor de variavel1 por 2. Logo 4 x 2:

O resultado será: 8

Atribuição Divisiva:

```
variavel1 = 4
resultado = variavel1 / 4

print(resultado)

# Mesmo que:
variavel2 = 4
variavel2 /= 4

print(variavel2)

```

```
1.0
1.0
```

Nesse caso, o usuário está dividindo o valor de variavel1 por 4. Sendo 4 / 4.

O resultado será: 1.0

Módulo de (ou resto da divisão de):

```
 1  variavel1 = 4
 2  resultado = variavel1 % 4
 3
 4  print(resultado)
 5
 6  # Mesmo que:
 7  variavel2 = 4
 8  variavel2 %= 4
 9
10  print(variavel2)
11
```

```
0
0
```

Será mostrado apenas o resto da divisão de variavel1 por 4.

O resultado será: 0

Exponenciação:

```
 1  variavel1 = 4
 2  resultado = variavel1 ** 8
 3
 4  print(resultado)
 5
 6  # Mesmo que:
 7  variavel2 = 4
 8  variavel2 **= 8
 9
10  print(variavel2)
11
```

```
65536
65536
```

Nesse caso, o valor de a será multiplicado 8 vezes por ele mesmo.

Como a valia 4 inicialmente, a exponenciação será (4*4*4*4*4*4*4*4).

O resultado será: 65536

Divisão Inteira:

```
variavel1 = 512
resultado = variavel1 // 256

print(resultado)

# Mesmo que:
variavel2 = 512
variavel2 //= 256

print(variavel2)
```

```
2
2
```

Neste caso a divisão retornará um número inteiro (ou arredondado). Ex: 512/256.

O resultado será: 2

Operadores aritméticos

Operadores aritméticos, como o nome sugere, são aqueles que usaremos para realizar operações matemáticas em meio a nossos blocos de código.

Python por padrão já vem com bibliotecas pré-alocadas que nos permitem a qualquer momento fazer operações matemáticas simples como soma, subtração, multiplicação e divisão.

Para operações de maior complexidade também é possível importar bibliotecas externas que irão implementar tais funções. Por hora, vamos começar do início, entendendo quais são os operadores que usaremos para realizarmos pequenas operações matemáticas.

Operador	Função
+	Realiza a soma de dois números
-	Realiza a subtração de dois números
*	Realiza a multiplicação de dois números
/	Realiza a divisão de dois números

Como mencionado anteriormente, a biblioteca que nos permite realizar tais operações já vem carregada quando iniciamos nosso IDE, nos permitindo a qualquer momento realizar os cálculos básicos que forem necessários. Por exemplo:

Soma

```
1  print(5 + 7)
2

12
```

O resultado será: 12

Subtração

```
1  print(12 - 3)
2
```

```
9
```

O resultado será: 9

Multiplicação

```
1  print(5 * 7)
2
```

```
35
```

O resultado será: 35

Divisão

```
1  print(120 / 6)
2
```

```
20.0
```

O resultado será: 20

Operações com mais de 2 operandos

```
1  print(5 + 2 * 7)
2
```

```
19
```

O resultado será 19 porque pela regra matemática primeiro se fazem as multiplicações e divisões para depois efetuar as somas ou subtrações, logo 2 x 7 são 14 que somados a 5 se tornam 19.

Operações dentro de operações:

```
1  print((5 + 2) * 7)
2
```

```
49
```

O resultado será 49 porque inicialmente é realizada a operação dentro dos parênteses (5 + 2) que resulta 7 e aí sim este valor é multiplicado por 7 fora dos parênteses.

Exponenciação

```
1  print(3 ** 5)
2
```

```
243
```

O resultado será 243, ou seja, 3 x 3 x 3 x 3 x 3.

Outra operação possível e a de fazer uma divisão que retorne um número inteiro, “arredondado”, através do operador //. Ex:

```
1  print(9.4 // 3)
2
```

```
3.0
```

O resultado será 3.0, um valor arredondado.

Por fim também é possível obter somente o resto de uma divisão fazendo o uso do operador %.

```
print(10 % 3)

1
```

O resultado será 1, porque 10 divididos por 3 são 9 e seu resto é 1.

Apenas como exemplo, para encerrar este tópico, é importante você raciocinar que os exemplos que dei acima poderiam ser executados diretamente no console/terminal de sua IDE, mas claro que podemos usar tais operadores dentro de nossos blocos de código, inclusive atribuindo valores numéricos a variáveis e realizando operações entre elas. Por exemplo:

```
numero1 = 12
numero2 = 3

print(numero1 + numero2)

# Mesmo que:
print('O resultado da soma é:', numero1 + numero2)

# Que pode ser aprimorado para:
print(f'O resultado da soma é: {numero1 + numero2}')

15
O resultado da soma é: 15
O resultado da soma é: 15
```

O resultado será 15, uma vez que numero1 tem como valor atribuído 12, e numero2 tem como valor atribuído 3. Somando as duas variáveis chegamos ao valor 15.

Exemplos com os demais operadores:

```
 1  x = 5
 2  y = 8
 3  z = 13.2
 4
 5  print(x + y)
 6  print(x - y)
 7  print(x ** z)
 8  print(z // y)
 9  print(z / y)
10
```

```
13
-3
1684240309.400895
1.0
1.65
```

Os resultados serão: 13

-3

1684240309.400895

1.0

1.65

Operadores Lógicos

Operadores lógicos seguem a mesma base lógica dos operadores relacionais, inclusive nos retornando Truee ou False, mas com o diferencial de suportar expressões lógicas de maior complexidade.

Por exemplo, se executarmos diretamente no console a expressão 7 != 3 teremos o valor True (7 diferente de 3, verdadeiro), mas se executarmos por exemplo 7 != 3 and 2 > 3 teremos como retorno False (7 é diferente de 3, mas 2 não é maior que 3, e pela tabela verdade isso já caracteriza False).

```
1  print(7 != 3)
2
```

```
True
```

O retorno será: True

Afinal, 7 é diferente de 3.

```
1  print(7 != 3 and 2 > 3)
2
```

```
False
```

O retorno será: False

7 é diferente de 3 (Verdadeiro) e (and) 2 é maior que 3 (Falso).

Tabela verdade

Tabela verdade AND (E)

Independentemente de quais expressões forem usadas, se os resultados forem:

V E V = V
V E F = F
F E V = F
F E F = F

No caso do exemplo anterior, 7 era diferente de 3 (V) mas 2 não era maior que 3 (F), o retorno foi False.

Neste tipo de tabela verdade, bastando uma das proposições ser Falsa para que invalide todas as outras Verdadeiras. Ex:

V e V e V e V e V e V = V
V e F e V e V e V e V = F

Em python o operador "and" é um operador lógico (assim como os aritméticos) e pela sequência lógica em que o interpretador trabalha, a expressão é lida da seguinte forma:

```
7 != 3 and 2 > 3
True and False
False
# V e F = F
```

Analisando estas estruturas lógicas estamos tentando relacionar se 7 é diferente de 3 (Verdadeiro) e se 2 é maior que 3 (Falso), logo pela tabela verdade Verdadeiro e Falso resultará False.

Mesma lógica para operações mais complexas, e sempre respeitando a tabela verdade.

7 != 3 and 3 > 1 and 6 == 6 and 8 >= 9
True and True and True and False
False

7 diferente de 3 (V) E 3 maior que 1 (V) E 6 igual a 6 (V) E 8 maior ou igual a 9 (F).

É o mesmo que True and True and True and False.

Retornando False porque uma operação False já invalida todas as outras verdadeiras.

Tabela Verdade OR (OU)

Neste tipo de tabela verdade, mesmo tendo uma proposição Falsa, ela não invalida a Verdadeira. Ex:

V e V = V
V e F = V
F e V = V
F e F = F

Independentemente do número de proposições, bastando ter uma delas verdadeira já valida a expressão inteira.

V e V e V e V e V e V e V e V = V
F e F e F e F e F e F e F e F = F
F e F e F e F e F E F e V e F = V
F e F e F e F e F e F e F e F = F

Tabela Verdade XOR (OU Exclusivo/um ou outro)

Os dois do mesmo tipo de proposição são falsos, e nenhum é falso também.

V e V = F
V e F = V
F e V = V
F e F = F

Tabela de Operador de Negação (unário)

not True = F

O mesmo que dizer: Se não é verdadeiro então é falso.

not False = V

O mesmo que dizer: Se não é falso então é verdadeiro.

Também visto como:

not 0 = True

O mesmo que dizer: Se não for zero / Se é diferente de zero então é verdadeiro.

not 1 = False

O mesmo que dizer: Se não for um (ou qualquer valor) então é falso.

Bit-a-bit

O interpretador também pode fazer uma comparação bit-a-bit da seguinte forma:

AND Bit-a-bit

3 = 11 (3 em binário)

2 = 10 (2 em binário)

_ = 10

OR bit-a-bit

3 = 11

2 = 10

_ = 11

XOR bit-a-bit

3 = 11

2 = 10

_ = 01

Por fim, vamos ver um exemplo prático do uso de operadores lógicos, para que faça mais sentido.

```
1  saldo = 1000
2  salario = 4000
3  despesas = 2967
4
5  meta = saldo > 0 and salario - despesas >= 0.2 * salario
6
7  print(meta)
8
```

```
True
```

Analisando a variável meta: Ela verifica se saldo é maior que zero e se salario menos despesas é maior ou igual a 20% do salário.

O retorno será True porque o saldo era maior que zero e o valor de salario menos as despesas era maior ou igual a 20% do salário. (todas proposições foram verdadeiras).

Operadores de membro

Ainda dentro de operadores podemos fazer consulta dentro de uma lista obtendo a confirmação (True) ou a negação (False) quanto a um determinado elemento constar ou não dentro da mesma. Por Exemplo:

```
1  lista = [1, 2, 3, 'Ana', 'Maria']
2
3  print(2 in lista)
4
```

```
True
```

O retorno será: True

Lembrando que uma lista é definida por [ ] e seus valores podem ser de qualquer tipo, desde que separados por vírgula.

Ao executar o comando 2 in lista, você está perguntando ao interpretador: "2" é membro desta lista? Se for (em qualquer posição) o retorno será True.

Também é possível fazer a negação lógica, por exemplo:

```
1  lista = [1, 2, 3, 'Ana', 'Maria']
2
3  print('Maria' not in lista)
4
```

```
False
```

O Retorno será False. ‘Maria’ não está na lista? A resposta foi False porque ‘Maria’ está na lista.

Operadores relacionais

Operadores relacionais basicamente são aqueles que fazem a comparação de dois ou mais operandos, possuem uma sintaxe própria que também deve ser respeitada para que não haja conflito com o interpretador.

>	-	Maior que
>=	-	Maior ou igual a
<	-	Menor que
<=	-	Menor ou igual a
==	-	Igual a
!=	-	Diferente de

O retorno obtido no uso desses operadores será Verdadeiro (True) ou Falso (False).

Usando como referência o console, operando diretamente nele, ou por meio de nossa função print( ), sem declarar variáveis, podemos fazer alguns experimentos.

```
1  print(3 > 4)
2  #(3 é maior que 4?)
3
```

```
False
```

O resultado será False, pois 3 não é maior que 4.

```
1  print(7 >= 3)
2  #(7 é maior ou igual a 3?)
3
```

```
True
```

O resultado será True. 7 é maior ou igual a 3, neste caso, maior que 3.

```
1  print(3 >= 3)
2  #(3 é maior ou igual a 3?)
3
```

```
True
```

O resultado será True. 3 não é maior, mas é igual a 3.

Operadores usando variáveis

```
1  x = 2
2  y = 7
3  z = 5
4
5  print(x > z)
6
```

```
False
```

O retorno será False. Porque x (2) não é maior que z (5).

```
1  x = 2
2  y = 7
3  z = 5
4
5  print(z <= y)
6
```

```
True
```

O retorno será True. Porque z (5) não é igual, mas menor que y (7).

```
1  x = 2
2  y = 7
3  z = 5
4
5  print(y != x)
6
```

```
True
```

O retorno será True porque y (7) e diferente de x (2).

Operadores usando condicionais

```
1  if (2 > 1):
2      print('2 é maior que 1')
3
```

```
2 é maior que 1
```

Repare que esse bloco de código se iniciou com um if, ou seja, com uma condicional, se dois for maior do que um, então seria executado a linha de código abaixo, que exibe uma mensagem para o usuário: 2 é maior que 1.

Mesmo exemplo usando valores atribuídos a variáveis e aplicando estruturas condicionais (que veremos em detalhe no capítulo seguinte):

```
1  num1 = 2
2  num2 = 1
3
4  if num1 > num2:
5      print('2 é maior que 1')
6
```

```
2 é maior que 1
```

O retorno será: 2 é maior que 1

Operadores de identidade

Seguindo a mesma lógica dos outros operadores, podemos confirmar se diferentes objetos tem o mesmo dado ou valor atribuído. Por exemplo:

```
1  aluguel = 250
2  energia = 250
3  agua = 65
4
5  print(aluguel is energia)
6
```

```
True
```

O retorno será True porque os valores atribuídos são os mesmos (nesse caso, 250).

ESTRUTURAS CONDICIONAIS

Quando aprendemos sobre lógica de programação e algoritmos, era fundamental entendermos que toda ação tem uma reação (mesmo que apenas interna ao sistema), dessa forma, conforme abstraíamos ideias para código, a coisa mais comum era nos depararmos com tomadas de decisão, que iriam influenciar os rumos da execução de nosso programa.

Muitos tipos de programas se baseiam em metodologias de estruturas condicionais, são programadas todas possíveis tomadas de decisão que o usuário pode ter e o programa executa e retorna certos aspectos conforme o usuário vai aderindo a certas opções.

Lembre-se das suas aulas de algoritmos, digamos que, apenas por exemplo o algoritmo ir_ate_o_mercado está sendo executado, e em determinada seção do mesmo existam as opções: SE estiver chovendo vá pela rua nº1, SE NÃO estiver chovendo, continue na rua nº2.

Esta é uma tomada de decisão onde o usuário irá aderir a um rumo ou outro, mudando as vezes totalmente a execução do programa, desde que essas possibilidades estejam programadas dentro do mesmo.

Não existe como o usuário tomar uma decisão que não está condicionada ao código, logo, todas possíveis tomadas de decisão dever ser programadas de forma lógica e responsiva, prevendo todas as possíveis alternativas.

Ifs, elifs e elses

Uma das partes mais legais de programação, sem sombra de dúvidas, é quando finalmente começamos a lidar com estruturas condicionais. Uma coisa é você ter um programa linear, que apenas executa uma tarefa após a outra, sem grandes interações e desfecho sempre linear, como um script passo-a-passo sequencial.

Já outra coisa é você colocar condições, onde de acordo com as variáveis o programa pode tomar um rumo ou outro.

Como sempre, começando pelo básico, em Python a sintaxe para trabalhar com condicionais é bastante simples se comparado a outras linguagens de programação, basicamente temos os comandos if (se), elif (o mesmo que else if / mas se) e else (se não) e os usaremos de acordo com nosso algoritmo demandar tomadas de decisão.

A lógica de execução sempre se dará dessa forma, o interpretador estará executando o código linha por linha até que ele encontrará uma das palavras reservadas mencionadas anteriormente que sinaliza que naquele ponto existe uma tomada de decisão, de acordo com a decisão que o usuário indicar, ou de acordo com a validação de algum parâmetro, o código executará uma instrução, ou não executará nada, ignorando esta condição e pulando para o bloco de código seguinte.

Partindo pra prática:

```
1  a = 33
2  b = 34
3  c = 35
4
5  if b > a:
6      print('b é MAIOR que a')
7
```

```
b é MAIOR que a
```

Declaradas três variáveis a, b e c com seus respectivos valores já

atribuídos na linha abaixo existe a expressão if, uma tomada de decisão, sempre um if será seguido de uma instrução, que se for verdadeira, irá executar um bloco de instrução indentado a ela.

Neste caso, se b for maior que a será executado o comando print( ).

O retorno será: b é MAIOR que a

*Caso o valor atribuído a b fosse menor que a, o interpretador simplesmente iria pular para o próximo bloco de código.

```
1  a = 33
2  b = 33
3  c = 35
4
5  if b > a:
6      print('b é MAIOR que a')
7  elif b == a:
8      print('b é IGUAL a a')
9

b é IGUAL a a
```

Repare que agora além da condicional if existe uma nova condicional declarada, o elif. Seguindo o método do interpretador, primeiro ele irá verificar se a condição de if é verdadeira, como não é, ele irá pular para esta segunda condicional.

Por convenção da segunda condicional em diante se usa elif, porém se você usar if repetidas vezes, não há problema algum.

Seguindo com o código, a segunda condicional coloca como instrução que se b for igual a a, e repare que nesse caso é, será executado o comando print.

O retorno será: b é IGUAL a a

```
 1  a = 33
 2  b = 1
 3  c = 608
 4
 5  if b > a:
 6      print('b é MAIOR que a')
 7  elif b == a:
 8      print('b é IGUAL a a')
 9  else:
10      print('b é MENOR que a')
11
```

```
b é MENOR que a
```

Por fim, o comando else funciona como última condicional, ou uma condicional que é acionada quando nenhuma das condições anteriores do código for verdadeira, else pode ter um bloco de código próprio porém note que ele não precisa de nenhuma instrução, já que seu propósito é justamente apenas mostrar que nenhuma condicional (e sua instrução) anterior foi válida. Ex:

```
 1  var1 = 18
 2  var2 = 2
 3  var3 = 'Maria'
 4  var4 = 4
 5
 6  if var2 > var1:
 7      print('A segunda variável é maior que a primeira')
 8  elif var2 == 500:
 9      print('A segunda variável vale 500')
10  elif var3 == var2:
11      print('A variavel 3 tem o mesmo valor da variavel 2')
12  elif var4 is str('4'):
13      print('A variavel 4 não é do tipo string')
14  else:
15      print('Nenhuma condição é verdadeira')
16
```

```
Nenhuma condição é verdadeira
```

*Como comentamos rapidamente lá no início do livro, sobre algoritmos, estes podem ter uma saída ou não, aqui a saída é mostrar ao usuário esta mensagem de erro, porém se este fosse um código interno não haveria a necessidade dessa mensagem como retorno, supondo que essas condições simplesmente não fossem válidas o interpretador iria pular para o próximo bloco de código executando o que viesse a seguir.

Por fim, revisando os códigos anteriores, declaramos várias variáveis, e partir delas colocamos suas respectivas condições, onde de acordo com a validação destas condições, será impresso na tela uma mensagem.

Importante entendermos também que o interpretador lê a instrução de uma condicional e se esta for verdadeira, ele irá executar o bloco de código e encerrar seu processo ali, pulando a verificação das outras condicionais.

Em suma, o interpretador irá verificar a primeira condicional, se esta for falsa, irá verificar a segunda e assim por diante até encontrar uma verdadeira, ali será executado o bloco de código indentado a ela e se encerrará o processo.

O legal é que como python é uma linguagem de programação dinamicamente tipada, você pode brincar à vontade alterando o valor das variáveis para verificar que tipos de mensagem aparece no seu terminal.

Também é possível criar cadeias de tomada de decisão com inúmeras alternativas, mas sempre lembrando que pela sequência lógica em que o interpretador faz sua leitura é linha-a-linha, quando uma condição for verdadeira o algoritmo encerra a tomada de decisão naquele ponto.

And e Or dentro de condicionais

Também é possível combinar o uso de operadores and e or para elaborar condicionais mais complexas (de duas ou mais condições válidas). Por exemplo:

```
1  a = 34
2  b = 33
3  c = 35
4
5  if a > b and c > a:
6      print('a é maior que b e c é maior que a')
7
```

```
a é maior que b e c é maior que a
```

Se a for maior que b e c for maior que a:

O retorno será: a é maior que b e c é maior que a

```
1  a = 35
2  b = 34
3  c = 35
4
5  if a > b or a > c:
6      print('a é a variavel maior')
7
```

```
a é a variavel maior
```

Se a for maior que b ou a for maior que c.

O retorno será: a é a variavel maior

Outro exemplo:

```
1  nota = int(input('Informe a nota: '))
2
3  if nota >= 9:
4      print('Parabéns, quadro de honra')
5  elif nota >= 7:
6      print('Aprovado')
7  elif nota >= 5:
8      print('Recuperação')
9  else:
10     print('Reprovado')
11
```

```
Informe a nota: 8
Aprovado
```

Primeiro foi declarada uma variável de nome nota, onde será solicitado ao usuário que a atribua um valor, após o usuário digitar uma nota e apertar ENTER, o interpretador fará a leitura do mesmo e de acordo com as condições irá imprimir a mensagem adequada a situação.

Se nota for maior ou igual a 9:

O retorno será: Parabéns, quadro de honra

Se nota for maior ou igual a 7:

O retorno será: Aprovado

Se nota for maior ou igual a 5:

O retorno será: Recuperação

Se nota não corresponder a nenhuma das proposições anteriores:

O retorno será: Reprovado

Condicionais dentro de condicionais

Outra prática comum é criar cadeias de estruturas condicionais, ou seja, blocos de código com condicionais dentro de condicionais. Python permite este uso e tudo funcionará perfeitamente desde que usada a sintaxe e indentação correta. Ex:

```
1  var1 = 0
2  var2 = int(input('Digite um número: '))
3
4  if var2 > var1:
5      print('Numero maior que ZERO')
6      if var2 == 1:
7          print('O número digitado foi 1')
8      elif var2 == 2:
9          print('O número digitado foi 2')
10     elif var2 == 3:
11         print('O número digitado foi 3')
12     else:
13         print('O número digitado é maior que 3')
14
15 else:
16     print('Número inválido')
17
```

```
Digite um número: 2
Numero maior que ZERO
O número digitado foi 2
```

Repare que foram criadas 2 variáveis var1 e var2, a primeira já com o valor atribuído 0 e a segunda será um valor que o usuário digitar conforme solicitado pela mensagem, convertido para int.

Em seguida foi colocada uma estrutura condicional onde se o valor de var2 for maior do que var1, será executado o comando print e em seguida, dentro dessa condicional que já foi validada, existe uma segunda estrutura condicional, que com seus respectivos ifs, elifs e elses irá verificar que

número é o valor de var2 e assim irá apresentar a respectiva mensagem.

Supondo que o usuário digitou 2:

O retorno será: Numero maior que ZERO

O número digitado foi 2

Fora dessa cadeia de condicionais (repare na indentação), ainda existe uma condicional else para caso o usuário digite um número inválido (um número negativo ou um caractere que não é um número).

Simulando switch/case

Quem está familiarizado com outras linguagens de programação está acostumado a usar a função Swich para que, de acordo com a necessidade, sejam tomadas certas decisões.

Em Python nativamente não temos a função switch, porém temos como simular sua funcionalidade através de uma função onde definiremos uma variável com dados em forma de dicionário (que veremos em detalhes nos capítulos subsequentes), e como um dicionário trabalha com a lógica de chave:valor, podemos simular de acordo com uma opção:umaopção.

Para ficar mais claro vamos ao código:

```python
class Cor:
    vermelho = 1
    verde = 2
    azul = 3
    branco = 4
    preto = 5

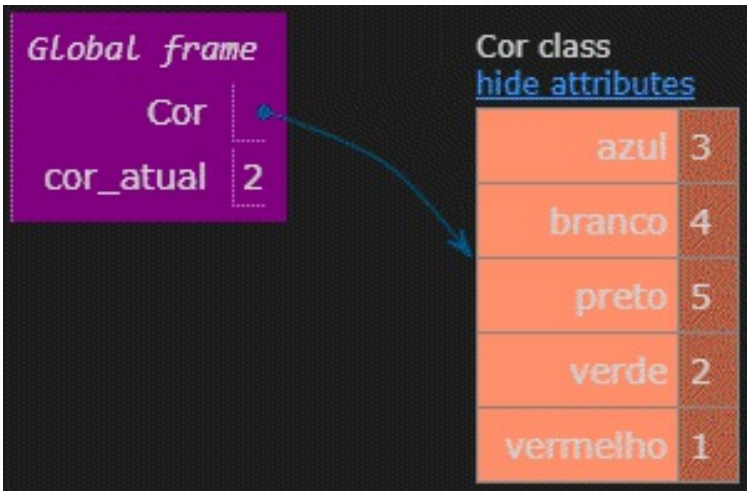
# Mude a cor para testar
cor_atual = 2

if cor_atual == Cor.vermelho:
    print("Vermelho")
elif cor_atual == Cor.verde:
    print("Verde")
elif cor_atual == Cor.azul:
    print("Azul")
elif cor_atual == Cor.branco:
    print("Branco")
elif cor_atual == Cor.preto:
    print("Preto")
else:
    print("Desconhecido")

```

```
Verde
```

Representação visual:

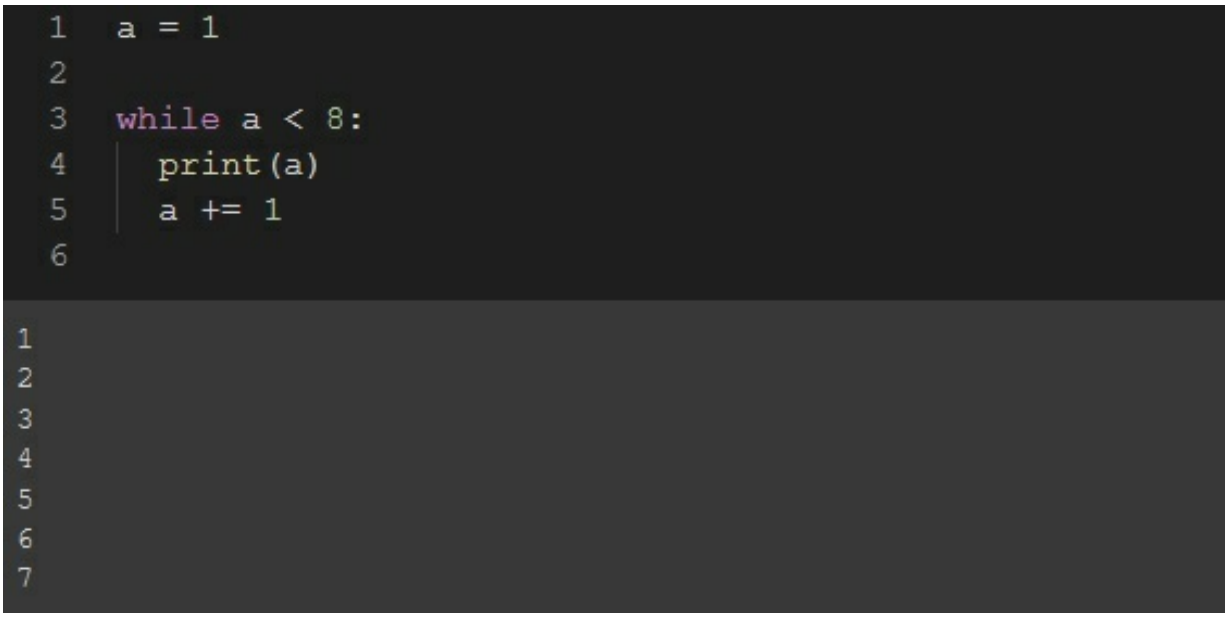


ESTRUTURAS DE REPETIÇÃO

While

Python tem dois tipos de comandos para executar comandos em loop (executar repetidas vezes uma mesma instrução) o while e o for.

Inicialmente vamos entender como funciona o while. While do inglês, em tradução livre significa enquanto, ou seja, enquanto uma determinada condição for válida, a ação continuará sendo repetida. Por exemplo:



```
a = 1

while a < 8:
    print(a)
    a += 1

```

```
1
2
3
4
5
6
7
```

Declarada a variável a, de valor inicial 1 (pode ser qualquer valor, inclusive zero) colocamos a condição de que, enquanto o valor de a for menor que 8, imprime o valor de a e acrescente (some) 1, repetidamente.

O retorno será: 1

2

3

4

5

6

Repare que isto é um loop, ou seja, a cada ação o bloco de código salva seu último estado e repete a instrução, até atingir a condição proposta.

Outra possibilidade é de que durante uma execução de while, podemos programar um break (comando que para a execução de um determinado bloco de código ou instrução) que acontece se determinada condição for atingida.

Normalmente o uso de break se dá quando colocamos mais de uma condição que, se a instrução do código atingir qualquer uma dessas condições (uma delas) ele para sua execução para que não entre em um loop infinito de repetições. Por exemplo:

```
1  a = 1
2
3  while a < 10:
4      print(a)
5      a += 1
6      if a == 4:
7          break
8
```

```
1
2
3
```

Enquanto a variável a for menor que 10, continue imprimindo-a e acrescentando 1 ao seu valor. Mas se em algum momento ela for igual a 4, pare a repetição.

Como explicado anteriormente, existem duas condições, se a execução do código chegar em uma delas, ele já dá por encerrada sua execução.

Neste caso, se em algum momento o valor de a for 4 ou for um número maior que 10 ele para sua execução.

O resultado será:1

2

3

4

For

O comando for será muito utilizado quando quisermos trabalhar com um laço de repetição onde conhecemos os seus limites, ou seja, quando temos um objeto em forma de lista ou dicionário e queremos que uma variável percorra cada elemento dessa lista/dicionário interagindo com o mesmo. Ex:

```
1  compras = ['Arroz', 'Feijão', 'Carne', 'Pão']
2
3  for i in compras:
4      print(i)
5
```

```
Arroz
Feijão
Carne
Pão
```

Note que inicialmente declaramos uma variável em forma de lista de nome compras, ele recebe 4 elementos do tipo string em sua composição.

Em seguida declaramos o laço for e uma variável temporária de nome i (você pode usar o nome que quiser, e essa será uma variável temporária, apenas instanciada nesse laço / nesse bloco de código) que para cada execução dentro de compras, irá imprimir o próprio valor.

Em outras palavras, a primeira vez que i entra nessa lista ela faz a leitura do elemento indexado na posição 0 e o imprime, encerrado o laço essa variável i agora entra novamente nessa lista e faz a leitura e exibição do elemento da posição 1 e assim por diante, até o último elemento encontrado nessa lista.

Outro uso bastante comum do for é quando sabemos o tamanho de um determinado intervalo, o número de elementos de uma lista, etc... e usamos seu método in range para que seja explorado todo esse intervalo. Ex:

```
1  for x in range(0, 6):
2      print(f'Número {x}')
3
```

```
Número 0
Número 1
Número 2
Número 3
Número 4
Número 5
```

Repare que já de início existe o comando for, seguido de uma variável temporária x, logo em seguida está o comando in range, que basicamente define um intervalo a percorrer (de 0 até 6).

Por fim, por meio de nossa função print( ) podemos ver cada valor atribuído a variável x a cada repetição.

O retorno será: Número 0

Número 1

Número 2

Número 3

Número 4

Número 5

*Note que a contagem dos elementos deste intervalo foi de 1 a 5, 6 já está fora do range, serve apenas como orientação para o interpretador de que ali é o fim deste intervalo. Em Python não é feita a leitura deste último dígito indexado, o interpretador irá identificar que o limite máximo desse intervalo é 6, sendo 5 seu último elemento.

Quando estamos trabalhando com um intervalo há a possibilidade de declararmos apenas um valor como parâmetro, o interpretador o usará como orientação para o fim de um intervalo.

Outro exemplo comum é quando já temos uma lista de elementos e

queremos a percorrer e exibir seu conteúdo.

```
1  nomes = ['Pedro', 255, 'Leticia']
2
3  for n in nomes:
4      print(n)
5
```

```
Pedro
255
Leticia
```

Repare que existe uma lista inicial, com 3 dados inclusos, já quando executamos o comando for ele internamente irá percorrer todos valores contidos na lista nomes e incluir os mesmos na variável n, independentemente do tipo de dado que cada elemento da lista.

Por fim o comando print foi dado em cima da variável n para que seja exibido ao usuário cada elemento dessa lista.

O resultado será: Pedro

255

Letícia

Em Python o laço for pode nativamente trabalhar como uma espécie de condicional, sendo assim podemos usar o comando else para incluir novas instruções no mesmo. Ex:

```
1  nomes = ['Pedro', 'João', 'Leticia',]
2
3  for laco in nomes:
4      print(laco)
5  else:
6      print('---Fim da lista!!!---')
7
```

```
Pedro
João
Leticia
---Fim da lista!!!---
```

O resultado será: Pedro

João

Leticia

---Fim da lista!!!---



Por fim, importante salientar que como for serve como laço de repetição ele suporta operações dentro de si, desde que essas operações requeiram repetidas instruções obviamente. Por exemplo:

```python
for x in range(11):
    for y in range(11):
        print(f'{x} x {y} = {x * y}')

```

```
0 x 5 = 0
0 x 6 = 0
0 x 7 = 0
0 x 8 = 0
0 x 9 = 0
0 x 10 = 0
1 x 0 = 0
1 x 1 = 1
1 x 2 = 2
1 x 3 = 3
1 x 4 = 4
1 x 5 = 5
1 x 6 = 6
1 x 7 = 7
1 x 8 = 8
1 x 9 = 9
1 x 10 = 10
2 x 0 = 0
2 x 1 = 2
2 x 2 = 4
2 x 3 = 6
2 x 4 = 8
2 x 5 = 10
2 x 6 = 12
```

O retorno será toda a tabuada de 0 x 0 até 10 x 10 (que por convenção não irei colocar toda aqui obviamente...).

```
"C:\Users\Fernando\PycharmProjects\Exercíci
0 x 0 = 0  1 x 0 = 0   2 x 0 = 0
0 x 1 = 0  1 x 1 = 1   2 x 1 = 2
0 x 2 = 0  1 x 2 = 2   2 x 2 = 4
0 x 3 = 0  1 x 3 = 3   2 x 3 = 6
0 x 4 = 0  1 x 4 = 4   2 x 4 = 8
0 x 5 = 0  1 x 5 = 5   2 x 5 = 10
0 x 6 = 0  1 x 6 = 6   2 x 6 = 12
0 x 7 = 0  1 x 7 = 7   2 x 7 = 14
0 x 8 = 0  1 x 8 = 8   2 x 8 = 16
0 x 9 = 0  1 x 9 = 9   2 x 9 = 18
0 x 10 = 0 1 x 10 = 10 2 x 10 = 20 etc...
```

STRINGS

Como já vimos algumas vezes ao longo deste livro, um objeto/variável é um espaço alocado na memória onde armazenaremos um tipo de dado ou informação para que o interpretador trabalhe com esses dados na execução do programa.

Uma string é o tipo de dado que usamos quando queremos trabalhar com qualquer tipo de texto, ou conjunto ordenado de caracteres alfanuméricos em geral.

Quando atribuímos um conjunto de caracteres, representando uma palavra/texto, devemos obrigatoriamente seguir a sintaxe correta para que o interpretador leia os dados como tal.

A sintaxe para qualquer tipo de texto é basicamente colocar o conteúdo desse objeto entre aspas ' ', uma vez atribuído dados do tipo string para uma variável, por exemplo nome = 'Maria', devemos lembrar de o referenciar como tal.

Apenas como exemplo, na sintaxe antiga do Python precisávamos usar do marcador %s em máscaras de substituição para que o interpretador de fato esperasse que aquele dado era do tipo String.

Na sintaxe moderna independente do uso o interpretador lê tudo o que estiver entre aspas ‘ ‘ como string.

Trabalhando com strings

No Python 3, se condicionou usar por padrão ' ' aspas simples para que se determine que aquele conteúdo é uma string, porém em nossa língua existem expressões que usam ' como apóstrofe, isso gera um erro de sintaxe ao interpretador.

Por exemplo:

```
1  'marca d'água'
2
```

```
File "<ipython-input-25-a3c48ea86f6a>", line 1
  'marca d'água'
            ^
SyntaxError: invalid syntax
```

O retorno será um erro de sintaxe porque o interpretador quando abre aspas ele espera que feche aspas apenas uma vez, nessa expressão existem 3 aspas, o interpretador fica esperando que você "feche" aspas novamente, como isso não ocorre ele gera um erro.

O legal é que é muito fácil contornar uma situação dessas, uma vez que python suporta, com a mesma função, " "aspas duplas, usando o mesmo exemplo anterior, se você escrever "marca d'água" ele irá ler perfeitamente todos caracteres (incluindo a apóstrofe) como string.

O mesmo ocorre se invertermos a ordem de uso das aspas. Por exemplo:

```
1  frase1 = 'Era um dia "muuuito" frio'
2
3  print(frase1)
4
```

```
Era um dia "muuuito" frio
```

O retorno será: Era um dia "muuuito" frio



Vimos anteriormente, na seção de comentários, que uma forma de comentar o código, quando precisamos fazer um comentário de múltiplas linhas, era as colocando entre """ aspas triplas (pode ser ''' aspas simples ou """ aspas duplas).

Um texto entre aspas triplas é interpretado pelo interpretador como um comentário, ou seja, o seu conteúdo será por padrão ignorado, mas se atribuirmos esse texto de várias linhas a uma variável, ele passa a ser um objeto comum, do tipo string.

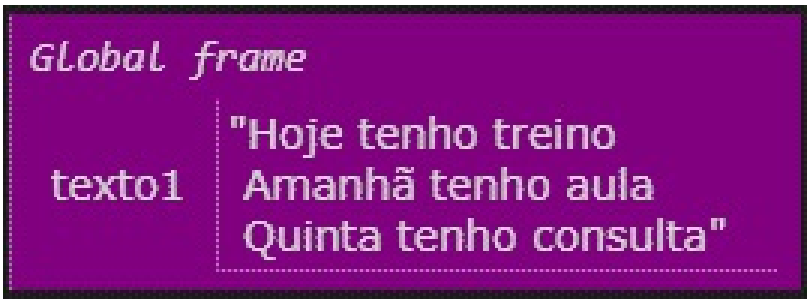
```
1  """Hoje tenho treino
2     Amanhã tenho aula
3     Quinta tenho consulta"""
4
```

*Esse texto nessa forma é apenas um comentário.

```
1  texto1 = """Hoje tenho treino
2                 Amanhã tenho aula
3                 Quinta tenho consulta"""
4
5  print(texto1)
6
```

```
Hoje tenho treino
Amanhã tenho aula
Quinta tenho consulta
```

*Agora esse texto é legível ao interpretador, inclusive você pode printar ele ou usar da forma como quiser, uma vez que agora ele é uma string.


Global frame
texto1
"Hoje tenho treino
Amanhã tenho aula
Quinta tenho consulta"

Formatando uma string

Quando estamos trabalhando com uma string também é bastante comum que por algum motivo precisamos formatá-la de alguma forma, e isso também é facilmente realizado desde que dados os comandos corretos.

Convertendo uma string para minúsculo

```
1  frase1 = 'A linguagem Python é muito fácil de aprender.'
2
3  print(frase1.lower())
4
```

```
a linguagem python é muito fácil de aprender.
```

Repare que na segunda linha o comando print( ) recebe como parâmetro o conteúdo da variável frase1 acrescido do comando .lower( ), convertendo a exibição dessa string para todos caracteres minúsculos.

O retorno será: a linguagem python é muito fácil de aprender.

Convertendo uma string para maiúsculo

Da mesma forma, o comando .upper( ) fará o oposto de .lower( ), convertendo tudo para maiúsculo. Ex:

```
1  frase1 = 'A linguagem Python é muito fácil de aprender.'
2
3  print(frase1.upper())
4
```

```
A LINGUAGEM PYTHON É MUITO FÁCIL DE APRENDER.
```

O retorno será: A LINGUAGEM PYTHON É MUITO FÁCIL DE APRENDER.

Lembrando que isso é uma formatação, ou seja, os valores da string não serão modificados, essa modificação não é permanente, ela é apenas a maneira com que você está pedindo para que o conteúdo da string seja mostrado, tratando o mesmo apenas em sua exibição.

Se você usar esses comandos em cima da variável em questão aí sim a mudança será permanente, por exemplo:

```
1  frase1 = 'A LINGUAGEM PYTHON É MUITO FÁCIL DE APRENDER.'
2  frase1 = frase1.lower()
3
4  print(frase1)
5
```

```
a linguagem python é muito fácil de aprender.
```

O retorno será: a linguagem python é muito fácil de aprender.

Nesse caso você estará alterando permanentemente todos caracteres da string para minúsculo.

Buscando dados dentro de uma string

Você pode buscar um dado dentro do texto convertendo ele para facilitar achar esses dados, por exemplo um nome que você não sabe se lá

dentro está escrito com a inicial maiúscula, todo em maiúsculo ou todo em minúsculo, para não ter que testar as 3 possibilidades, você pode usar comandos como:

```
1  frase1 = 'A LINGUAGEM PYTHON É MUITO FÁCIL DE APRENDER.'
2  frase1 = frase1.lower()
3
4  print('py' in  frase1.lower())
5
```

```
True
```

O retorno será True. Primeiro toda string foi convertida para minúscula, por segundo foi procurado ‘py’ que nesse caso consta dentro da string.

Desmembrando uma string

O utro comando interessante é o .split( ), ele irá desmembrar uma string em palavras separadas, para que você possa fazer a formatação ou o uso de apenas uma delas, por exemplo:

```
1  frase1 = 'A LINGUAGEM PYTHON É MUITO FÁCIL DE APRENDER.'
2  frase1 = frase1.lower()
3
4  print(frase1.split())
5
```

```
['a', 'linguagem', 'python', 'é', 'muito', 'fácil', 'de', 'aprender.']
```

O resultado será: ['a', 'linguagem', 'python', 'é', 'muito', 'fácil', 'de', 'aprender.']

Alterando a cor de um texto

Supondo que você quer imprimir uma mensagem de erro que chame a atenção do usuário, você pode fazer isso associando também uma cor a este texto, por meio de um código de cores, é possível criar variáveis que irão determinar a cor de um texto, por exemplo:

```
1  ERRO = '\033[91m'    #código de cores "vermelho"
2  NORMAL  = '\033[0m'
3
4  print(ERRO, 'Mensagem de erro aqui' , NORMAL)
5

Mensagem de erro aqui
```

O resultado será: Mensagem de erro aqui.

Uma vez que você declarou as cores como variáveis, você pode incorporar elas nos próprios parâmetros do código (note também que aqui como é uma variável criada para um caso especial, ela inclusive foi criada com nomenclatura toda maiúscula para eventualmente destaca-la de outras variáveis comuns no corpo do código).

```
1  ERRO = '\033[91m'    #código de cores "vermelho"
2  NORMAL  = '\033[0m'
3
4  print(ERRO + 'Mensagem de erro' + NORMAL)
5

Mensagem de erro
```

O resultado será: Mensagem de erro aqui.

Apenas como referência, segue uma tabela com os principais códigos ANSI de cores tanto para a fonte quanto para o fundo.

Cor	Fonte	Fundo
Preto	\033[1;30m	\033[1;40m

Vermelho	\033[1;31m	\033[1;41m
Verde	\033[1;32m	\033[1;42m
Amarelo	\033[1;33m	\033[1;43m
Azul	\033[1;34m	\033[1;44m
Magenta	\033[1;35m	\033[1;45m
Cyan	\033[1;36m	\033[1;46m
Cinza Claro	\033[1;37m	\033[1;47m
Cinza Escuro	\033[1;90m	\033[1;100m
Vermelho Claro	\033[1;91m	\033[1;101m
Verde Claro	\033[1;92m	\033[1;102m
Amarelo Claro	\033[1;93m	\033[1;103m
Azul Claro	\033[1;94m	\033[1;104m
Magenta Claro	\033[1;95m	\033[1;105m
Cyan Claro	\033[1;96m	\033[1;106m
Branco	\033[1;97m	\033[1;107m

Alterando a posição de exibição de um texto

Estamos acostumados a, enquanto trabalhamos dentro de um editor de texto, usar do artifício de botões de atalho que podem fazer com que um determinado texto fique centralizado ou alinhado a um dos lados da página, por exemplo.

Internamente isto é feito pelos comandos center( ) (centralizar), ljust( ) (alinhar à esquerda) e rjust( ) (alinhar à direita).

Exemplo de centralização:

```
1  frase1 = 'Bem Vindo ao Meu Programa!!!'
2
3  print(frase1.center(50))
4
```

```
        Bem Vindo ao Meu Programa!!!
```

Repare que a função print( ) aqui tem como parâmetro a variável frase1 acrescida do comando center(50), ou seja, centralizar dentro do intervalo de 50 caracteres.

A string inteira terá 50 caracteres, como frase1 é menor do que isto, os outros caracteres antes e depois dela serão substituídos por espaços.

O retorno será: ‘ Bem Vindo ao Meu Programa!!!
‘

Exemplo de alinhamento à direita:

```
1  frase1 = 'Bem Vindo ao Meu Programa!!!'
2
3  print(frase1.rjust(50))
4
```

```
                Bem Vindo ao Meu Programa!!!
```

O retorno será: ‘ Bem Vindo ao Meu Programa!!!’

Formatando a apresentação de números em uma string

Existirão situações onde, seja em uma string ou num contexto geral, quando pedirmos ao Python o resultado de uma operação numérica ou um valor pré estabelecido na matemática (como pi por exemplo) ele irá exibir este número com mais casas decimais do que o necessário. Ex:

```
1  from math import pi
2
3  print(f'O número pi é: {pi}')
4
```

```
O número pi é: 3.141592653589793
```

Na primeira linha estamos importando o valor de pi da biblioteca externa math. Em seguida dentro da string estamos usando uma máscara que irá ser substituída pelo valor de pi, mas nesse caso, como padrão.

O retorno será: O número pi é: 3.141592653589793

Num outro cenário, vamos imaginar que temos uma variável com um valor int extenso, mas só queremos exibir suas duas primeiras casas decimais, nesse caso, isto pode ser feito pelo comando .f e uma máscara simples. Ex:

```
1  num1 = 34.295927957329247
2
3  print('O valor da ação fechou em %.2f'%num1)
4
```

```
O valor da ação fechou em 34.30
```

Repare que dentro da string existe uma máscara de substituição % seguida de .2f, estes 2f significam ao interpretador que nós queremos que sejam exibidas apenas duas casas decimais após a vírgula. Neste caso o retorno será: O valor da ação fechou em 34.30

Usando o mesmo exemplo, mas substituindo .2f por .5f, o resultado será: O valor da ação fechou em 34.29593 (foram exibidas 5 casas “após a vírgula”).

LISTAS

Listas em Python são o equivalente a Arrays em outras linguagens de programação, mas calma que se você está começando do zero, e começando com Python, esse tipo de conceito é bastante simples de entender.

Listas são um dos tipos de dados aos quais iremos trabalhar com frequência, uma lista será um objeto que permite guardar diversos dados dentro dele, de forma organizada e indexada.

É como se você pegasse várias variáveis de vários tipos e colocasse em um espaço só da memória do seu computador, dessa maneira, com a indexação correta, o interpretador consegue buscar e ler esses dados de forma muito mais rápida do que trabalhar com eles individualmente. Ex:

```
1  lista = [ ]
2
```

Podemos facilmente criar uma lista com já valores inseridos ou adiciona-los manualmente conforme nossa necessidade, sempre respeitando a sintaxe do tipo de dado. Ex:

```
1  lista2 = [1, 5, 'Maria', 'João']
2
3  print(lista2)
4

[1, 5, 'Maria', 'João']
```

Aqui criamos uma lista já com 4 dados inseridos no seu índice, repare que os tipos de dados podem ser mesclados, temos ints e strings na composição dessa lista, sem problema algum.

Podemos assim deduzir também que uma lista é um tipo de variável onde conseguimos colocar diversos tipos de dados sem causar conflito.


Global frame
lista2
list
0
1
1
5
2
"Maria"
3
"João"

Adicionando dados manualmente

Se queremos adicionar manualmente um dado ou um valor a uma lista, este pode facilmente ser feito pelo comando append( ). Ex:

```
1  lista2 = [1, 5, 'Maria', 'João']
2  lista2.append(4)
3
4  print(lista2)
5
```

```
[1, 5, 'Maria', 'João', 4]
```

Irá adicionar o valor 4 na última posição do índice da lista.

O retorno será: [1, 5, 'Maria', 'João', 4]



Lembrando que por enquanto, esses comandos são sequenciais, ou seja, a cada execução do comando append( ) para inserir um novo elemento na lista, o mesmo será inserido automaticamente na última posição da mesma.

Removendo dados manualmente

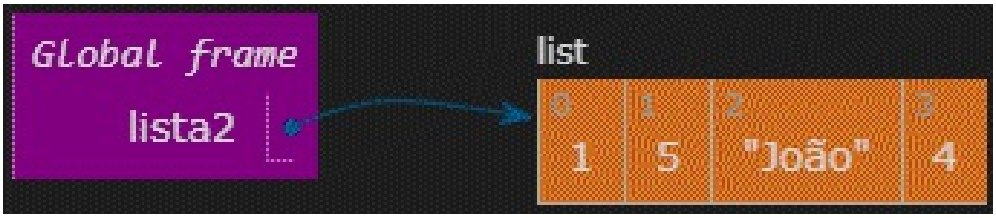
Da mesma forma, podemos executar o comando .remove( ) para remover o conteúdo de algum índice da lista.

```
1  lista2 = [1, 5, 'Maria', 'João']
2  lista2.append(4)
3  lista2.remove('Maria')
4
5  print(lista2)
6
```

```
[1, 5, 'João', 4]
```

Irá remover o dado Maria, que nesse caso estava armazenado na posição 2 do índice.

O retorno será: [1, 5, 'João', 4]



Lembrando que no comando .remove você estará dizendo que conteúdo você quer excluir da lista, supondo que fosse uma lista de nomes [‘Paulo’, ‘Ana’, ‘Maria’] o comando .remove(‘Maria’) irá excluir o dado ‘Maria’, o próprio comando busca dentro da lista onde esse dado específico estava guardado e o remove, independentemente de sua posição.

Removendo dados via índice

Para deletarmos o conteúdo de um índice específico, basta executar o comando del lista[nº do índice]

```
1  lista2 = ['Ana', 'Carlos', 'João', 'Sonia']
2  del lista2[2]
3
4  print(lista2)
5
```

```
['Ana', 'Carlos', 'Sonia']
```

O retorno será: ['Ana', 'Carlos', 'Sonia']



Repare que inicialmente temos uma lista ['Ana', 'Carlos', 'João', 'Sonia'] e executarmos o comando del lista[2] iremos deletar 'João' pois ele está no índice 2 da lista. Nesse caso 'Sonia' que era índice 3 passa a ser índice 2, ele assume a casa anterior, pois não existe “espaço vazio” numa lista.

Verificando a posição de um elemento

Para verificarmos em que posição da lista está um determinado elemento, basta executarmos o comando .index( ) Ex:

```
1  lista2 = [1, 5, 'Maria', 'João']
2  lista2.append(4)
3
4  print(lista2.index('Maria'))
5

2
```

O retorno será 2, porque 'Maria' está guardado no índice 2 da lista.

Representação visual:



Se você perguntar sobre um elemento que não está na lista o interpretador simplesmente retornará uma mensagem de erro dizendo que de fato, aquele elemento não está na nossa lista.

Verificando se um elemento consta na lista

Podemos também trabalhar com operadores aqui para consultar em nossa lista se um determinado elemento consta nela ou não. Por exemplo, executando o código 'João' in lista devemos receber um retorno True ou False.

```
1  lista2 = [1, 5, 'Maria', 'João']
2  lista2.append(4)
3
4  print('João' in lista2)
5
```

```
True
```

O retorno será: True

Formatando dados de uma lista

Assim como usamos comandos para modificar/formatar uma string anteriormente, podemos aplicar os mesmos comandos a uma lista, porém é importante que fique bem claro que aqui as modificações ficam imediatamente alteradas (no caso da string, nossa formatação apenas alterava a forma como seria exibida, mas na variável a string permanecia integra). Ex:

```
1  lista1 = []
2
3  lista1.append('Paulo')
4  lista1.append('Maria')
5
6  print(lista1)
7
```

```
['Paulo', 'Maria']
```

Inicialmente criamos uma lista vazia e por meio do método append( ) inserimos nela duas strings. Por meio da função print( ) podemos exibir em tela seu conteúdo:

O retorno será [Paulo, Maria]

Representação visual:



```
 1  lista1 = []
 2
 3  lista1.append('Paulo')
 4  lista1.append('Maria')
 5
 6  print(lista1)
 7
 8  lista1.reverse()
 9
10  print(lista1)
11
```

```
['Paulo', 'Maria']
['Maria', 'Paulo']
```

Se fizermos o comando lista1.reverse( ) e em seguida dermos o comando print(lista1) o retorno será [Maria, Paulo], e desta vez esta alteração será permanente.

Listas dentro de listas

Também é possível adicionar listas dentro de listas, parece confuso, mas na prática, ao inserir um novo dado dentro de um índice, se você usar a sintaxe de lista [ ] você estará adicionando, ali naquela posição do índice, uma nova lista. Ex:

```
1  lista = [1, 2, 4, 'Paulo']
2
3  print(lista)
4
```

```
[1, 2, 4, 'Paulo']
```

Adicionando uma nova lista no índice 4 da lista atual ficaria:

```
1  lista = [1, 2, 4, 'Paulo']
2
3  print(lista)
4
5  lista = [1, 2, 4, 'Paulo', [2, 5, 'Ana']]
6
7  print(lista)
8
```

```
[1, 2, 4, 'Paulo']
[1, 2, 4, 'Paulo', [2, 5, 'Ana']]
```

O retorno será [1, 2, 4, 'Paulo', [2, 5, 'Ana']]

Representação visual:



Trabalhando com Tuplas

Uma Tupla trabalha assim como uma lista para o interpretador, mas a principal diferença entre elas é que lista é dinâmica (você pode alterá-la à vontade) enquanto uma tupla é estática (elementos atribuídos são fixos) e haverão casos onde será interessante usar uma ou outra.

Um dos principais motivos para se usar uma tupla é o fato de poder incluir um elemento, o nome Paulo por exemplo, várias vezes em várias posições do índice, coisa que não é permitida em uma lista, a partir do momento que uma lista tem um dado ou valor atribuído, ele não pode se repetir.

Segunda diferença é que pela sintaxe uma lista é criada a partir de colchetes, uma tupla a partir de parênteses novamente.

```
1  minhatupla = tuple( )
2
```

Para que o interpretador não se confunda, achando que é um simples objeto com um parâmetro atribuído, pela sintaxe, mesmo que haja só um elemento na tupla, deve haver ao menos uma vírgula como separador. Ex:

```
1  minhatupla = (1, )
2
3  print(minhatupla)
4
```

```
(1,)
```

```
1  minhatupla = tuple( )
2
3  type(minhatupla)
4
```

```
tuple
```

Se você executar type(minhatupla) você verá tuple como retorno, o que está correto, se não houvesse a vírgula o interpretador iria retornar o valor int (já que 1 é um número inteiro).

```
1  minhatupla = (1, )
2
3  dir(minhatupla)
4
```

```
'__init__',
'__init_subclass__',
'__iter__',
'__le__',
'__len__',
'__lt__',
'__mul__',
'__ne__',
'__new__',
'__reduce__',
'__reduce_ex__',
'__repr__',
'__rmul__',
'__setattr__',
'__sizeof__',
'__str__',
'__subclasshook__',
'count',
'index']
```

Se você der um comando dir(minhatupla) você verá que basicamente temos index e count como comandos padrão que podemos executar, ou seja, ver seus dados e contá-los também.

Sendo assim, você pode acessar o índice exatamente como fazia com sua lista. Ex:

```
1  minhatupla = (1, 2, 3)
2  minhatupla[0]
3
```

```
1
```

O retorno será: 1

Outro ponto a ser comentado é que, se sua tupla possuir apenas um elemento dentro de si, é necessário indicar que ela é uma tupla ou dentro de si colocar uma vírgula como separador mesmo que não haja um segundo elemento. Já quando temos 2 ou mais elementos dentro da tupla, não é mais necessário indicar que ela é uma tupla, o interpretador identificará automaticamente. Ex:

```
1  minhatupla = tuple('Ana')
2  minhatupla2 = ('Ana',)
3  minhatupla3 = ('Ana', 'Maria')
4
5  print(minhatupla)
6  print(minhatupla2)
7  print(minhatupla3)
8
```

```
'A', 'n', 'a')
'Ana',)
'Ana', 'Maria')
```

Em minhatupla existe a atribuição de tipo tuple( ), em minhatupla2 existe a vírgula como separador mesmo não havendo um segundo elemento, em minhatupla3 já não é mais necessário isto para que o identifique o tipo de dado como tupla.

Como uma tupla tem valores já predefinidos e imutáveis, é comum haver mais do mesmo elemento em diferentes posições do índice. Ex:

```
1  tuplacores = ('azul', 'branco', 'roxo', 'azul',
2                'vermelho', 'preto', 'azul', 'amarelo')
3
4  tuplacores.count('azul')
5
```

```
3
```

Executando o código tuplacores.count('azul'), o retorno será 3 porque existem dentro da tupla 'azul' 3 vezes.

Representação visual:



Trabalhando com Pilhas

Pilhas nada mais são do que listas em que a inserção e a remoção de elementos acontecem na mesma extremidade.

Para abstrairmos e entendermos de forma mais fácil, imagine uma pilha de papéis, se você colocar mais um papel na pilha, será em cima dos outros, da mesma forma que o primeiro papel a ser removido da pilha será o do topo.

Adicionando um elemento ao topo de pilha

Para adicionarmos um elemento ao topo da pilha podemos usar o nosso já conhecido .append( ) recebendo o novo elemento como parâmetro.

```
1  pilha = [10, 20, 30]
2  pilha.append(50)
3
4  print(pilha)
5
```

```
[10, 20, 30, 50]
```

O retorno será: [10, 20, 30, 50]

Removendo um elemento do topo da pilha

Para removermos um elemento do topo de uma pilha podemos usar o comando .pop( ), e neste caso como está subentendido que será removido o último elemento da pilha (o do topo) não é necessário declarar o mesmo como parâmetro.

```
1  pilha = [10, 20, 30]
2  pilha.pop()
3
4  print(pilha)
5
```

```
[10, 20]
```

O retorno será: [10, 20]

Consultando o tamanho da pilha

Da mesma forma como consultamos o tamanho de uma lista, podemos consultar o tamanho de uma pilha, para termos noção de quantos elementos ela possui e de qual é o topo da pilha. Ex:

```
1  pilha = [10, 20, 30]
2  pilha.append(50)
3
4  print(pilha)
5  print(len(pilha))
6
```

```
[10, 20, 30, 50]
4
```

O retorno será: [10, 20, 30, 50]

4

Representação visual:



Na primeira linha, referente ao primeiro print( ), nos mostra os elementos da pilha (já com a adição do “50” em seu topo. Por fim na segunda linha, referente ao segundo print( ) temos o valor 4, dizendo que esta pilha tem 4 elementos;

DICIONÁRIOS

Enquanto listas e tuplas são estruturas indexadas, um dicionário, além da sintaxe também diferente, se resume em organizar dados em formato chave : valor.

Assim como em um dicionário normal você tem uma palavra e seu respectivo significado, aqui a estrutura lógica será a mesma.

A sintaxe de um dicionário é definida por chaves { }, deve seguir a ordem chave:valor e usar vírgula como separador, para não gerar um erro de sintaxe. Ex:

```
1  fernando = {'nome': 'Fernando Feltrin',
2              'idade': 31,
3              'formacao': ['Eng. da Computação', 'Técnico em TI']}
4
5  print(fernando)
6
```

```
{'nome': 'Fernando Feltrin', 'idade': 31,
 'formacao': ['Engenheiro da Computação', 'Técnico em TI']}
```

Repare que inicialmente o dicionário foi atribuído a um objeto, depois seguindo a sintaxe foram criados 3 campos (nome, idade e formação) e dentro de formação ainda foi criado uma lista, onde foram adicionados dois elementos ao índice.



Uma prática comum tanto em listas, quanto em dicionários, é usar de uma tabulação que torne visualmente o código mais organizado, sempre que estivermos trabalhando com estes tipos de dados teremos uma vírgula

como separador dos elementos, e é perfeitamente normal após uma vírgula fazer uma quebra de linha para que o código fique mais legível.

Esta prática não afeta em nada a leitura léxica do interpretador nem a performance de operação do código.

```
fernando = {'nome': 'Fernando Feltrin',
            'idade': 31,
            'formacao': ['Engenheiro da Computação',
                         'Técnico em TI']}

```

Executando um type(fernando) o retorno será dict, porque o interpretador, tudo o que estiver dentro de { } chaves ele interpretará como chaves e valores de um dicionário.

Assim como existem listas dentro de listas, você pode criar listas dentro de dicionários e até mesmo dicionários dentro de dicionários sem problema algum.

Você pode usar qualquer tipo de dado como chave e valor, o que não pode acontecer é esquecer de declarar um ou outro pois irá gerar erro de sintaxe.

Da mesma forma como fazíamos com listas, é interessante saber o tamanho de um dicionário, e assim como em listas, o comando len( ) agora nos retornará à quantidade de chaves:valores inclusos no dicionário. Ex:

```
fernando = {'nome': 'Fernando Feltrin',
            'idade': 31,
            'formacao': ['Engenheiro da Computação',
                         'Técnico em TI']}

print(len(fernando))

```

```
3
```

O retorno será 3.

Consultando chaves/valores de um dicionário

Você pode consultar o valor de dentro de um dicionário pelo comando .get( ). Isso será feito consultando uma determinada chave para obter como retorno seu respectivo valor. Ex:

```
fernando = {'nome': 'Fernando Feltrin',
            'idade': 31,
            'formacao': ['Engenheiro da Computação',
                         'Técnico em TI']}

print(fernando.get('idade'))

```

```
31
```

O retorno será 31.

Consultando as chaves de um dicionário

É possível consultar quantas e quais são as chaves que estão inclusas dentro de um dicionário pelo comando .keys( ). Ex:

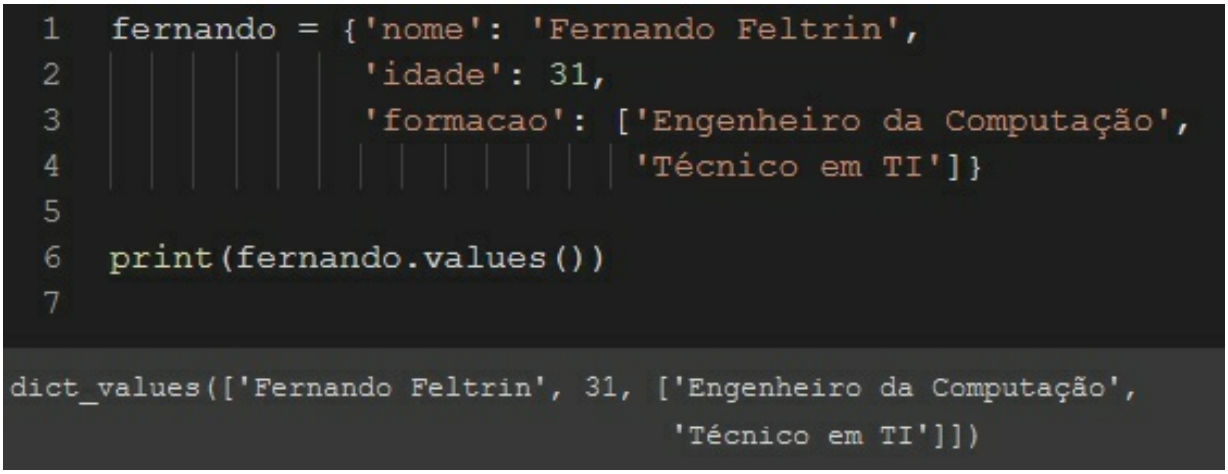
```
1  fernando = {'nome': 'Fernando Feltrin',
2              'idade': 31,
3              'formacao': ['Engenheiro da Computação',
4                           'Técnico em TI']}
5
6  print(fernando.keys())
7
```

```
dict_keys(['nome', 'idade', 'formacao'])
```

O retorno será: dict_keys(['nome', 'idade', 'formacao'])

Consultando os valores de um dicionário

Assim como é possível fazer a leitura somente dos valores, pelo comando .values( ). Ex:



```
fernando = {'nome': 'Fernando Feltrin',
            'idade': 31,
            'formacao': ['Engenheiro da Computação',
                         'Técnico em TI']}

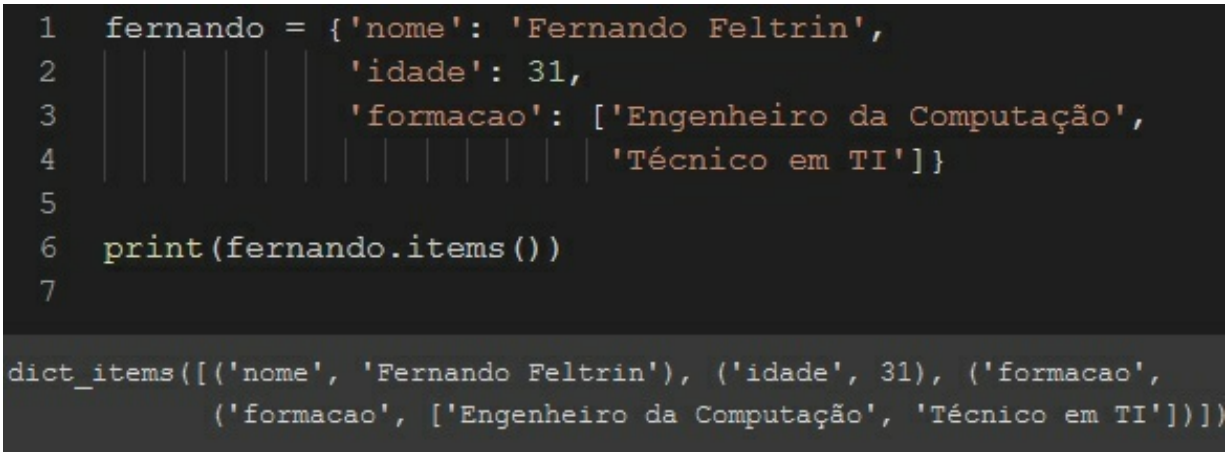
print(fernando.values())
```

```
dict_values(['Fernando Feltrin', 31, ['Engenheiro da Computação',
                                      'Técnico em TI']])
```

O retorno será: dict_values(['Fernando Feltrin', 31, ['Engenheiro da Computação', 'Técnico em TI']])

Mostrando todas chaves e valores de um dicionário

Por fim o comando .items( ) retornará todas as chaves e valores para sua consulta. Ex:



```
fernando = {'nome': 'Fernando Feltrin',
            'idade': 31,
            'formacao': ['Engenheiro da Computação',
                         'Técnico em TI']}

print(fernando.items())

dict_items([('nome', 'Fernando Feltrin'), ('idade', 31), ('formacao',
           ('formacao', ['Engenheiro da Computação', 'Técnico em TI'])])
```

O retorno será: dict_items([('nome': 'Fernando Feltrin', 'idade': 31, 'formacao': ['Engenheiro da Computação', 'Técnico em TI'])])

Manipulando dados de um dicionário

Quanto temos um dicionário já com valores definidos, pré-programados, mas queremos alterá-los, digamos, atualizar os dados dentro de nosso dicionário, podemos fazer isso manualmente através de alguns simples comandos.

Supondo que inicialmente temos um dicionário:

```
1  pessoa = {'nome': 'Alberto Feltrin',
2            'idade': '42',
3            'formação': ['Tec. em Radiologia'],
4            'nacionalidade': 'brasileiro'}
5
6  print(pessoa)
7
```

```
{'nome': 'Alberto Feltrin', 'idade': '42', 'formação': ['Tec. em Radiologia'],
 'nacionalidade': 'brasileiro'}
```

O retorno será: {'nome': 'Alberto Feltrin', 'idade': '42', 'formação': ['Tec. em Radiologia'], 'nacionalidade': 'brasileiro'}

Representação visual:



```
 1  pessoa = {'nome': 'Alberto Feltrin',
 2            'idade': '42',
 3            'formação': ['Tec. em Radiologia'],
 4            'nacionalidade': 'brasileiro'}
 5
 6  print(pessoa)
 7
 8  pessoa['idade'] = 44
 9
10  print(pessoa)
11
```

```
{'nome': 'Alberto Feltrin', 'idade': '42', 'formação': ['Tec. em Radiologia'],
 'nacionalidade': 'brasileiro'}
{'nome': 'Alberto Feltrin', 'idade': 44, 'formação': ['Tec. em Radiologia'],
 'nacionalidade': 'brasileiro'}
```

Executando o comando pessoa['idade'] = 44 estaremos atualizando o valor 'idade' para 44 no nosso dicionário, consultando o dicionário novamente você verá que a idade foi atualizada.

Executando novamente print(pessoa) o retorno será: {'nome': 'Alberto Feltrin', 'idade': 44, 'formação': ['Tec. em Radiologia', 'nacionalidade']: 'brasileiro'}

Adicionando novos dados a um dicionário

Além de substituir um valor por um mais atual, também é possível adicionar manualmente mais dados ao dicionário, assim como fizemos anteriormente com nossas listas, através do comando .append( ). Ex:

```
pessoa = {'nome': 'Alberto Feltrin',
          'idade': '42',
          'formação': ['Tec. em Radiologia'],
          'nacionalidade': 'brasileiro'}

pessoa['idade'] = 44

pessoa['formação'].append('Esp. em Tomografia')

print(pessoa)

```

```
{'nome': 'Alberto Feltrin', 'idade': 44, 'formação': ['Tec. em Radiologia',
 'Esp. em Tomografia'], 'nacionalidade': 'brasileiro'}
```

O retorno será: {'nome': 'Alberto Feltrin', 'idade': 44, 'formação': ['Tec. em Radiologia', 'Esp. em Tomografia'], 'nacionalidade': 'brasileiro'}

Representação visual:



*Importante salientar que para que você possa adicionar mais valores a uma chave, ela deve respeitar a sintaxe de uma lista.

CONJUNTOS NUMÉRICOS

Se você lembrar das aulas do ensino médio certamente lembrará que em alguma etapa lhe foi ensinado sobre conjuntos numéricos, que nada mais era do que uma forma de categorizarmos os números quanto suas características (reais, inteiros, etc...).

Na programação isto se repete de forma parecida, uma vez que quando queremos trabalhar com conjuntos numéricos normalmente estamos organizando números para poder aplicar funções matemáticas sobre os mesmos.

Um conjunto é ainda outra possibilidade de armazenamento de dados que temos, de forma que ele parece uma lista com sintaxe de dicionário, mas não indexável e que não aceita valores repetidos, confuso não?

A questão é que como aqui podemos ter valores repetidos, assim como conjuntos numéricos que usávamos no ensino médio, aqui podemos unir dois conjuntos, fazer a intersecção entre eles, etc...

Vamos aos exemplos:

```
1  a = {1, 2, 3}
2
3  type(a)
4
```

```
set
```

Se executarmos o comando type(a) o retorno será set. O interpretador consegue ver na sintaxe que não é nem uma lista, nem uma tupla e nem um dicionário (apesar da simbologia { }), mas apenas um conjunto de dados alinhados.

Trabalhando com mais de um conjunto, podemos fazer operações entre eles. Por exemplo:

União de conjuntos

A união de dois conjuntos é feita através da função .union( ), de forma que a operação irá na verdade gerar um terceiro conjunto onde constam os valores dos dois anteriores. Ex:

```
1  c1 = {1, 2}
2  c2 = {2, 3}
3
4  c1.union(c2)
5  #c1 união com c2, matematicamente juntar 2 conjuntos.
6
```

```
{1, 2, 3}
```

O retorno será: {1, 2, 3}

Repare que a união dos dois conjuntos foi feita de forma a pegar os valores que não eram comuns aos dois conjuntos e incluir, e os valores que eram comuns aos dois conjuntos simplesmente manter, sem realizar nenhuma operação.

Interseção de conjuntos

Já a interseção entre eles fará o oposto, anulando valores avulsos e mantendo só os que eram comuns aos dois conjuntos. Interseção de dois conjuntos através do operador .intersection( ). Ex:

```
1  c1 = {1, 2}
2  c2 = {2, 3}
3
4  c1.intersection(c2)
5
```

```
{2}
```

O retorno será:{2}, pois este é o único elemento em comum aos dois conjuntos.

Lembrando que quando executamos expressões matemáticas elas apenas alteram o que será mostrado para o usuário, a integridade dos dados iniciais se mantem.

Para alterar os dados iniciais, ou seja, salvar essas alterações, é possível executar o comando .update( ) após as expressões, assim os valores serão alterados de forma permanente. Ex:

```
1  c1 = {1, 2}
2  c2 = {2, 3}
3
4  c1.union(c2)
5  c1.update(c2)
6
7  print(c1)
8
```

```
{1, 2, 3}
```

Agora consultando c1 o retorno será {1, 2, 3}

Verificando se um conjunto pertence ao outro

Também é possível verificar se um conjunto está contido dentro do outro, a través da expressão <=. Ex:

```
1  c1 = {1, 2}
2  c2 = {2, 3}
3
4  c1.union(c2)
5  c1.update(c2)
6
7  c2 <= c1
8
```

```
True
```

O retorno será True porque os valores de c2 (2 e 3) estão contidos dentro de c1 (que agora é 1, 2 e 3).

Diferença entre conjuntos

Outra expressão matemática bastante comum é fazer a diferença entre dois conjuntos. Ex:

```
 1  c1 = {1, 2}
 2  c2 = {2, 3}
 3
 4  c1.union(c2)
 5  c1.update(c2)
 6
 7  c1 = c1 - {2}
 8
 9  print(c1)
10
```

```
{1, 3}
```

O retorno será: {1, 3} #O elemento 2 foi subtraído dos conjuntos.

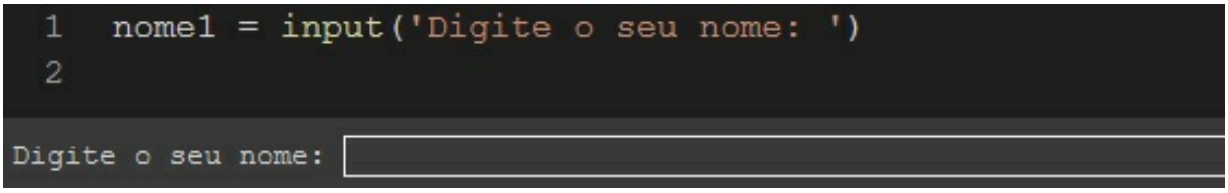
INTERPOLAÇÃO

Quando estamos escrevendo nossos primeiros códigos, costumamos escrever todas expressões de forma literal e, na verdade, não há problema nenhum nisto. Porém quando estamos avançando nos estudos de programação iremos ver aos poucos que é possível fazer a mesma coisa de diferentes formas, e a proficiência em uma linguagem de programação se dá quando conseguirmos criar códigos limpos, legíveis, bem interpretados e enxutos em sua sintaxe.

O ponto que eu quero chegar é que pelas boas práticas de programação, posteriormente usaremos uma simbologia em nossas expressões que deixarão nosso código mais limpo, ao mesmo tempo um menor número de caracteres por linha e um menor número de linhas por código resultam em um programa que é executado com performance superior.

Um dos conceitos que iremos trabalhar é o de máscaras de substituição que como o próprio nome já sugere, se dá a uma série de símbolos reservados pelo sistema que servirão para serem substituídos por um determinado dado ao longo do código.

Vamos ao exemplo:



A partir daí poderíamos simplesmente mandar exibir o nome que o usuário digitou e que foi atribuído a variável nome1. Ex:

```
1  nome1 = input('Digite o seu nome: ')
2
3  print(nome1)
4
```

```
Digite o seu nome: Fernando
Fernando
```

O resultado será o nome que o usuário digitou anteriormente. Porém existe uma forma mais elaborada, onde exibiremos uma mensagem e dentro da mensagem haverá uma máscara que será substituída pelo nome digitado pelo usuário.

Vamos supor que o usuário irá digitar “Fernando”, então teremos um código assim:

```
1  nome1 = input('Digite o seu nome: ')
2
3  print('Seja bem vindo %s'%(nome1))
4
```

```
Digite o seu nome: Fernando
Seja bem vindo Fernando
```

O retorno será: Seja bem vindo Fernando

Repare que agora existe uma mensagem de boas vindas seguida da máscara %s (que neste caso está deixando um marcador, que por sua vez identifica ao interpretador que ali será inserido um dado em formato string) e após a frase existe um operador que irá ler o que está atribuído a variável nome1 e irá substituir pela máscara.

As máscaras de substituição podem suportar qualquer tipo de dado desde que referenciados da forma correta: %s(string) ou %d(número inteiro) por exemplo.

Também é possível usar mais de uma máscara se necessário, desde que se respeite a ordem das máscaras com seus dados a serem substituídos e a sintaxe equivalente, que agora é representada por { } contendo dentro o número da ordem dos índices dos dados a serem substituídos pelo comando

.format( ).

Por exemplo:

```
1  nota1 = '{0} está reprovado, assim como seu colega {1}'
2
3  print(nota1.format('Pedro', 'Francisco'))
4
```

```
Pedro está reprovado, assim como seu colega Francisco
```

O Resultado será: Pedro está reprovado, assim como seu colega Francisco

Este tipo de interpolação também funcionará normalmente sem a necessidade de identificar a ordem das máscaras, podendo inclusive deixar as { } chaves em branco.

Avançando com interpolações

```
1  nome = 'Maria'
2  idade = 30
3
```

Se executarmos o comando print( ) da seguinte forma:

```
1  nome = 'Maria'
2  idade = 30
3
4  print('Nome: %s Idade: %d' % (nome, idade))
5
```

```
Nome: Maria Idade: 30
```

O interpretador na hora de exibir este conteúdo irá substituir %s pela variável do tipo string e o %d pela variável do tipo int declarada anteriormente, sendo assim, o retorno será: Nome: Maria Idade: 30

Python vem sofrendo uma serie de "atualizações" e uma delas de maior impacto diz respeito a nova forma de se fazer as interpolações, alguns programadores consideraram essa mudança uma "mudança para pior", mas é tudo uma questão de se acostumar com a nova sintaxe.

Por quê estou dizendo isto, porque será bastante comum você pegar exemplos de códigos na web seguindo a sintaxe antiga, assim como códigos já escritos em Python 3 que adotaram essa nova sintaxe que mostrarei abaixo.

Importante salientar que as sintaxes antigas não serão (ao menos por enquanto) descontinuadas, ou seja, você pode escrever usando a que quiser, ambas funcionarão, mas será interessante começar a se acostumar com a nova.

Seguindo o mesmo exemplo anterior, agora em uma sintaxe intermediaria:

```
1  nome = 'Maria'
2  idade = 30
3
4  print('Nome: {0} Idade: {1}'.format(nome, idade))
5
```

```
Nome: Maria Idade: 30
```

O resultado será: Nome: Maria Idade: 30

Repare que agora o comando vem seguido de um índice ao qual serão substituídos pelos valores encontrados no .format( ).

E agora por fim seguindo o padrão da nova sintaxe, mais simples e funcional que os anteriores:

```
1  nome = 'Maria'
2  idade = 30
3
4  print(f'Nome: {nome}, Idade: {idade}')
5
```

```
Nome: Maria, Idade: 30
```

Repare que o operador .format( ) foi abreviado para simplesmente f no início da sentença, e dentro de suas máscaras foram referenciadas as variáveis a serem substituídas.

O resultado será: Nome: Maria Idade: 30

O ponto que eu quero que você observe é que à medida que as linguagens de programação vão se modernizando, elas tendem a adotar sintaxes e comandos mais fáceis de serem lidos, codificados e executados.

A nova sintaxe, utilizando de chaves { } para serem interpoladas, foi um grande avanço também no sentido de que entre chaves é possível escrever expressões (por exemplo uma expressão matemática) que ela será lida e interpretada normalmente, por exemplo:

```python
nome = 'Maria'
idade = 30

print(f'Nome: {nome}, Idade: {idade}')

print(f'{nome} tem 3 vezes minha idade, ela tem {3 * idade} anos')

```

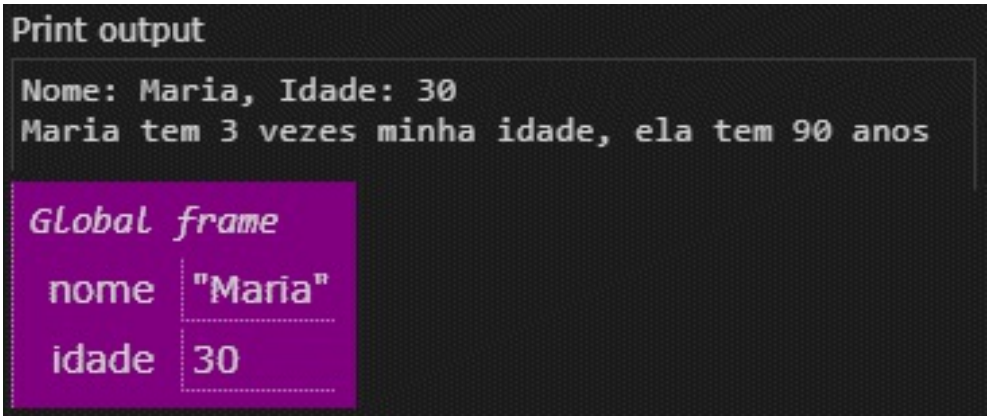
```
Nome: Maria, Idade: 30
Maria tem 3 vezes minha idade, ela tem 90 anos
```

O resultado será: Nome: Maria, Idade: 30

Maria tem 3 vezes minha idade, ela tem 90 anos

Repare que a expressão matemática de multiplicação foi realizada dentro da máscara de substituição, e ela não gerou nenhum erro pois o interpretador está trabalhando com seu valor atribuído, que neste caso era 30.

Representação visual:



FUNÇÕES

Já vimos anteriormente as funções print( ) e input( ), duas das mais utilizadas quando estamos criando estruturas básicas para nossos programas. Agora que você já passou por outros tópicos e já acumula uma certa bagagem de conhecimento, seria interessante nos aprofundarmos mais no assunto funções, uma vez que existem muitas possibilidades novas quando dominarmos o uso das mesmas, já que o principal propósito é executar certas ações por meio de funções.

Uma função, independentemente da linguagem de programação, é um bloco de códigos que podem ser executados e reutilizados livremente, quantas vezes forem necessárias.

Uma função pode conter ou não parâmetros, que nada mais são do que instruções a serem executadas cada vez que a referenciamos associando as mesmas com variáveis, sem a necessidade de escrever repetidas vezes o mesmo bloco de código da mesma funcionalidade.

Para ficar mais claro imagine que no corpo de nosso código serão feitas diversas vezes a operação de somar dois valores e atribuir seu resultado a uma variável, é possível criar uma vez esta calculadora de soma e definir ela como uma função, de forma que eu posso posteriormente no mesmo código reutilizar ela apenas mudando seus parâmetros.

Funções predefinidas

A linguagem Python já conta com uma série de funções pré definidas e pré carregadas por sua IDE, prontas para uso. Como dito anteriormente, é muito comum utilizarmos, por exemplo, a função print( ) que é uma função dedicada a exibir em tela o que é lhe dado como parâmetro (o que está dentro dos parênteses). Outro exemplo seria a função len( ) que mostra o tamanho de um determinado dado.

Existe a possibilidade de importarmos módulos externos, ou seja, funções já criadas e testadas que estão disponíveis para o usuário, porém fazem parte de bibliotecas externas, que não vêm carregadas por padrão.

Por exemplo é possível importar de uma biblioteca chamada math as funções sin( ), cos( ) e tg( ), que respectivamente, como seu nome sugere, funções prontas para calcular o seno, o cosseno e a tangente de um determinado valor.

Funções personalizadas

Seguindo o nosso raciocínio, temos funções prontas pré carregadas, funções prontas que simplesmente podemos importar para nosso código e certamente haverão situações onde o meio mais rápido ou prático será criar uma função própria personalizada para uma determinada situação.

Basicamente quando necessitamos criar uma função personalizada, ela começará com a declaração de um def, é o meio para que o interpretador assume que a partir dessa palavra reservada está sendo criada uma função.

Uma função personalizada pode ou não receber parâmetros (variáveis / objetos instanciados dentro de parênteses), pode retornar ou não algum dado ou valor ao usuário e por fim, terá um bloco de código indentado para receber suas instruções.

Função simples, sem parâmetros

```
1  def boas_vindas():
2      print('Olá, seja bem vindo ao meu programa!!!')
3      print('Espero que você tenha uma boa experiência...')
4
5  print(boas_vindas())
6
```

```
Olá, seja bem vindo ao meu programa!!!
Espero que você tenha uma boa experiência...
None
```

Analisando o código podemos perceber que na primeira linha está o comando def seguido do nome dado a nova função, neste caso boas_vindas( ).

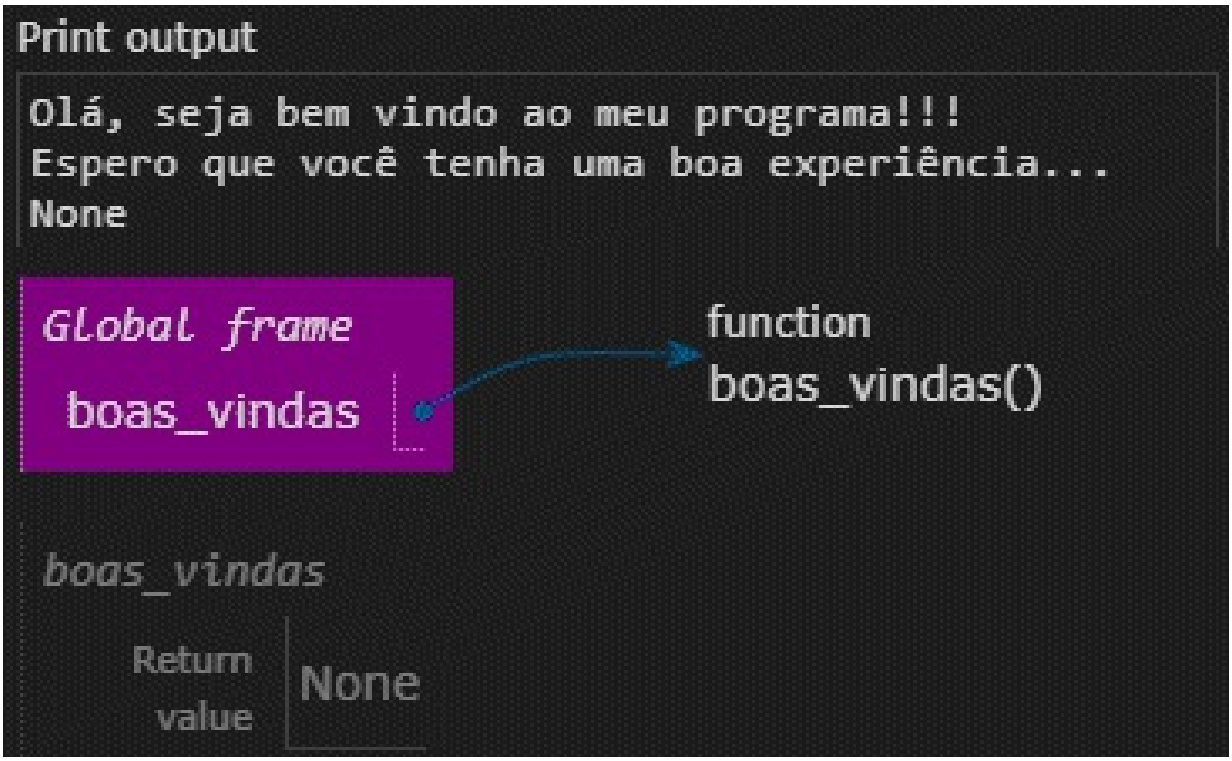
Repare que após o nome existe dois pontos : e na linha abaixo, indentado, existem dois comandos print( ) com duas frases.

Por fim, dado o comando print(boas_vindas()) foi chamada a função e seu conteúdo será exibido em tela.

O retorno será: Olá, seja bem vindo ao meu programa!!!

Espero que você tenha uma boa experiência...

Representação visual:



Como mencionado anteriormente, uma vez definida essa função boas_vindas( ) toda vez que eu quiser usar ela (seu conteúdo na verdade) basta fazer a referência corretamente.

Função composta, com parâmetros

```
1  def eleva_numero_ao_cubo(num):
2      valor_a_retornar = num * num * num
3      return(valor_a_retornar)
4
5  num = eleva_numero_ao_cubo(5)
6
7  print(num)
8
```

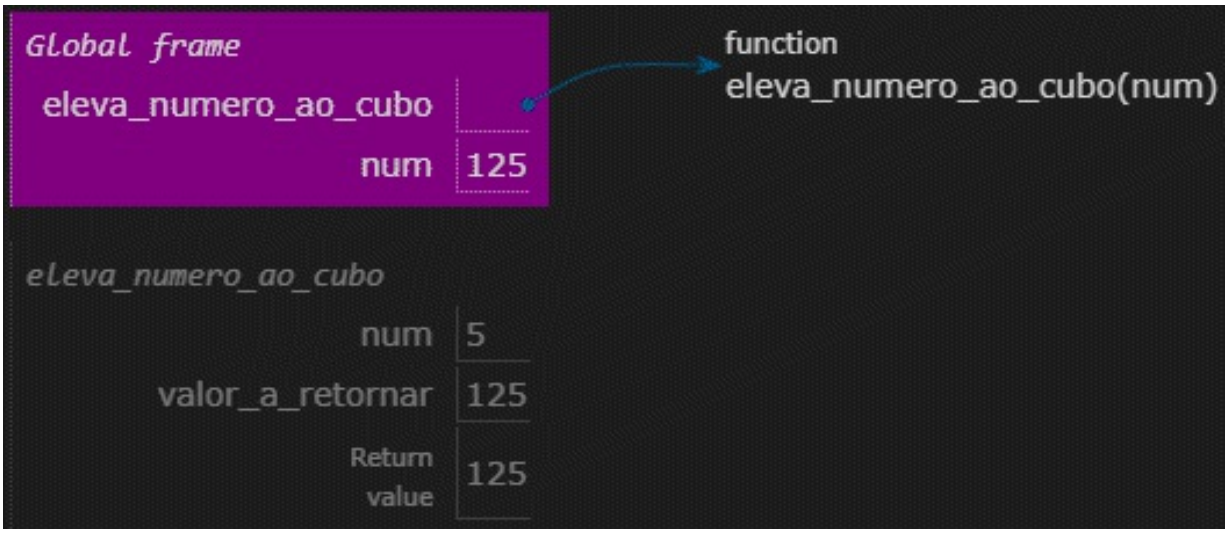
```
125
```

Repare que foi definida a função eleva_numero_ao_cubo(num) com num como parâmetro, posteriormente foi criada uma variável temporária de nome valor_a_retornar que pega num e o multiplica por ele mesmo 3 vezes (elevando ao cubo), por fim existe a instrução de retornar o valor atribuído a valor_a_retornar.

Seguindo o código existe uma variável num que chama a função eleva_numero_ao_cubo e dá como parâmetro o valor 5 (que pode ser alterado livremente), e finalizando executa o comando print( ) de num que irá executar toda a mecânica criada na função eleva_numero_ao_cubo com o parâmetro dado, neste caso 5.

O retorno será: 125

Representação visual:


Global frame
eleva_numero_ao_cubo
num
125
function
eleva_numero_ao_cubo(num)
eleva_numero_ao_cubo
num
5
valor_a_retornar
125
Return value
125

Função composta, com *args e **kwargs

Outra possibilidade que existe em Python é a de trabalharmos com *args e com **kwargs, que nada mais são do que instruções reservadas ao sistema para que permita o uso de uma quantidade variável de argumentos.

Por conversão se usa a nomenclatura args e kwargs mas na verdade você pode usar o nome que quiser, o marcador que o interpretador buscará na verdade serão os * asteriscos simples ou duplos.

Importante entender também que, como estamos falando em passar um número variável de parâmetros, isto significa que internamente no caso de *args os mesmos serão tratados como uma lista e no caso de **kwargs eles serão tratados como dicionário.

Exemplo ***args**

```
1  def print_2_vezes(*args):
2      for parametro in args:
3          print(parametro + '!' + parametro + '!')
4
5  print_2_vezes('Olá Mundo!!! ')
6
```

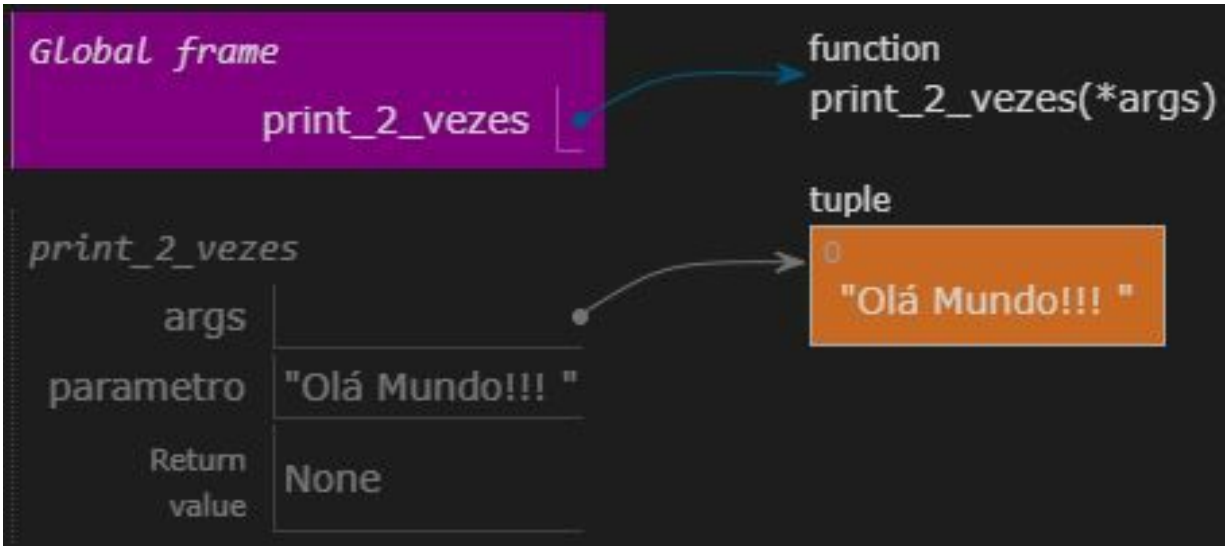
```
Olá Mundo!!! !Olá Mundo!!! !
```

Definida a função print_2_vezes(*args) é passado como instrução um comando for, ou seja um laço que irá verificar se a variável temporária parametro está contida em args, como instrução do laço é passado o comando print, que irá exibir duas vezes e concatenar o que estiver atribuído a variável temporária parametro.

Por fim, seguindo o código é chamada a função print_2_vezes e é passado como parâmetro a string ‘Olá Mundo!!!’. Lembrando que como existem instruções dentro de instruções é importante ficar atento a indentação das mesmas.

O retorno será: Olá Mundo!!! !Olá Mundo!!!

Representação visual:



Exemplo de ****kwargs** :

```
1  def informacoes(**kwargs):
2      for dado, valor in kwargs.items():
3          print(dado + '-' + str(valor))
4
5  pessoa = informacoes(nome='Fernando',
6                       idade=30,
7                       nacionalidade='Brasileiro')
8
9  print(pessoa)
10
```

```
nome-Fernando
idade-30
nacionalidade-Brasileiro
None
```

Definida a função informacoes(**kwargs) é passado como instrução o comando for, que irá verificar a presença dos dados atribuídos a

dado e valor, uma vez que kwargs espera que, como num dicionário, sejam atribuídas chaves:valores.

Por fim, se estes parametros estiverem contidos em kwargs será dado o comando print dos valores de dados e valores, convertidos para string e concatenados com o separador " – ".

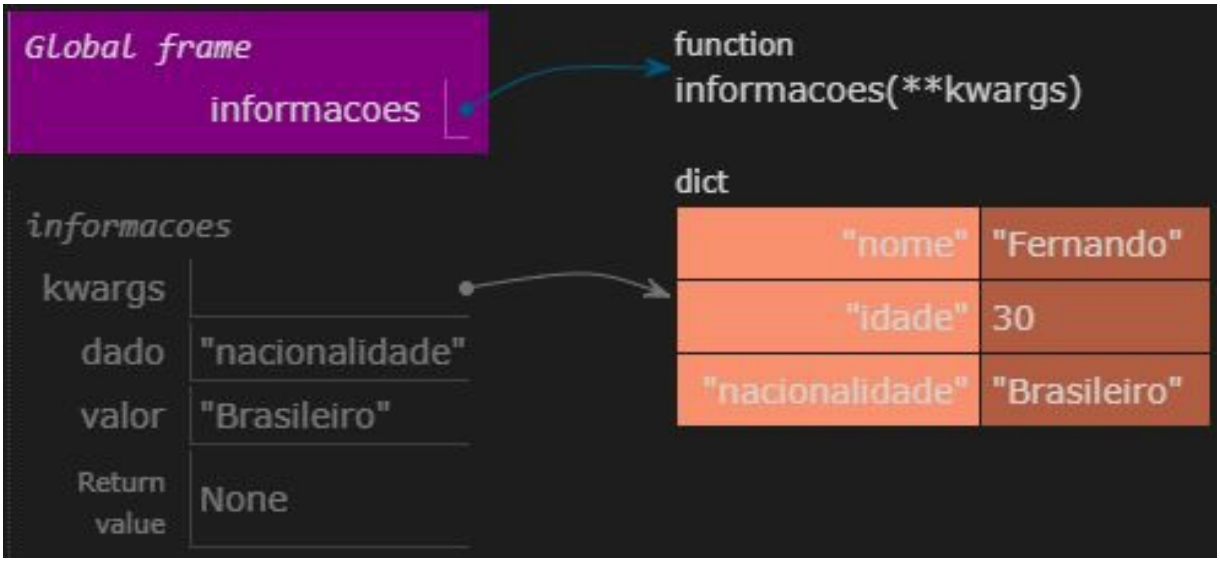
Então é chamada a função informações( ) pela variável pessoa e é passado como chaves e valores para a mesma os dados contidos no código acima.

O retorno será: nome-Fernando

idade-30

nacionalidade-Brasileiro

Representação visual:



dir( ) e help( )

O Python como padrão nos fornece uma série de operadores que facilitam as nossas vidas para executar comandos com nossos códigos, mas é interessante você raciocinar que o que convencionalmente usamos quando estamos codificando nossas ideias não chega a ser 10% do que o python realmente tem para oferecer.

Estudando mais a fundo a documentação ou usando o comando dir, podemos perguntar a um tipo de objeto (int, float, string, list, etc...) todos os recursos internos que ele possui, além disso é possível importar para o python novas funcionalidades, que veremos futuramente neste mesmo curso.

Como exemplo digamos que estamos criando uma lista para guardar nela diversos tipos de dados. Pela sintaxe, uma lista é um objeto/variável que pode receber qualquer nome e tem como característica o uso de colchetes [ ] para guardar dentro os seus dados. Ex:

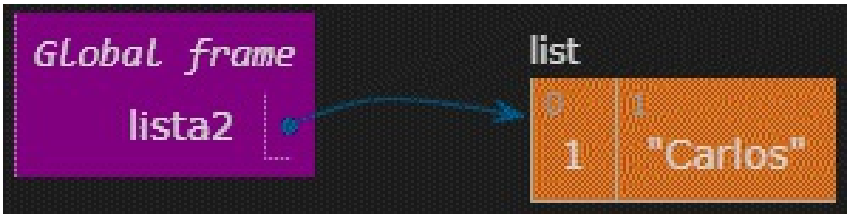
```
1  lista1 = []
2
```

Lista vazia

```
1  lista2 = [1,'Carlos']
2
```

Lista com o valor 1 alocado na posição 0 do índice e com 'Carlos' alocado na posição 1 do índice.

Representação visual:



```
1  lista2 = [1,'Carlos']
2
3  print(dir(lista2))
4
```

Usando o comando dir(lista2) Iremos obter uma lista de todas as possíveis funções que podem ser executadas sobre uma lista

['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']

```
1  lista2 = [1,'Carlos']
2
3  print(help(lista2))
4
```

Muito parecido com o dir( ) é o comando help( ), você pode usar o comando help(lista2) e você receberá também uma lista (agora muito mais detalhada) com todas possíveis funções a serem executadas sobre a suas listas. Ex:

O retorno será:

```
|  list(iterable=(), /)
|
|  Built-in mutable sequence.
|
|  If no argument is given, the constructor creates a new empty list.
|  The argument must be an iterable if specified.
|
|  Methods defined here:
|
|  __add__(self, value, /)
|      Return self+value.
|
|  __contains__(self, key, /)
|      Return key in self.
|
|  __delitem__(self, key, /)
|      Delete self[key].
|
|  __eq__(self, value, /)
|      Return self==value.
|
|  __ge__(self, value, /)
|      Return self>=value.
|
|  __getattribute__(self, name, /)
|      Return getattr(self, name).
|
|  __getitem__(...)
|      x.__getitem__(y) <==> x[y]
|
|  __gt__(self, value, /)
```

```
|      Return self>value.
|
|  __iadd__(self, value, /)
|      Implement self+=value.
|
|  __imul__(self, value, /)
|      Implement self*=value.
|
|  __init__(self, /, *args, **kwargs)
|      Initialize self.  See help(type(self)) for accurate signature.
|
|  __iter__(self, /)
|      Implement iter(self).
|
|  __le__(self, value, /)
|      Return self<=value.
|
|  __len__(self, /)
|      Return len(self).
|
|  __lt__(self, value, /)
|      Return self<value.
|
|  __mul__(self, value, /)
|      Return self*value.
|
|  __ne__(self, value, /)
|      Return self!=value.
|
|  __repr__(self, /)
|      Return repr(self).
|
```

```
|  __reversed__(self, /)
|      Return a reverse iterator over the list.
|
|  __rmul__(self, value, /)
|      Return value*self.
|
|  __setitem__(self, key, value, /)
|      Set self[key] to value.
|
|  __sizeof__(self, /)
|      Return the size of the list in memory, in bytes.
|
|  append(self, object, /)
|      Append object to the end of the list.
|
|  clear(self, /)
|      Remove all items from list.
|
|  copy(self, /)
|      Return a shallow copy of the list.
|
|  count(self, value, /)
|      Return number of occurrences of value.
|
|  extend(self, iterable, /)
|      Extend list by appending elements from the iterable.
|
|  index(self, value, start=0, stop=9223372036854775807, /)
|      Return first index of value.
|
|      Raises ValueError if the value is not present.
|
```

```
|  insert(self, index, object, /)
|      Insert object before index.
|
|  pop(self, index=-1, /)
|      Remove and return item at index (default last).
|
|      Raises IndexError if list is empty or index is out of range.
|
|  remove(self, value, /)
|      Remove first occurrence of value.
|
|      Raises ValueError if the value is not present.
|
|  reverse(self, /)
|      Reverse *IN PLACE*.
|
|  sort(self, /, *, key=None, reverse=False)
|      Stable sort *IN PLACE*.
|
|  ----------------------------------------------------------------------
|  Static methods defined here:
|
|  __new__(*args, **kwargs) from builtins.type
|      Create and return a new object.  See help(type) for accurate
signature.
|
|  ----------------------------------------------------------------------
|  Data and other attributes defined here:
|
|  __hash__ = None
```

Repare que foi classificado e listado cada recurso possível de ser executado

em cima de nossa lista2, assim como o retorno que é esperado em cada uso.

BUILTINS

Builtins, através do comando dir( ), nada mais é do que uma forma de você pesquisar quais são os módulos e recursos que já vieram pré-alocados em sua IDE. De forma que você pode pesquisar quais são as funcionalidades que estão disponíveis, e, principalmente as indisponíveis, para que você importe o módulo complementar necessário.

Para ficar mais claro, imagine que você consegue fazer qualquer operação matemática básica pois essas funcionalidades já estão carregadas, pré-alocadas e prontas para uso em sua IDE.

Caso você necessite usar expressões matemáticas mais complexas, é necessário importar a biblioteca math para que novas funcionalidades sejam inclusas em seu código.

Na prática você pode executar o comando dir(__builtins__) para ver tudo o que está disponível em sua IDE em tempo real.

```
1  print(dir(__builtins__))
2
```

O retorno será:

['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning', 'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError', 'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning', 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False', 'FileExistsError', 'FileNotFoundError', 'FloatingPointError', 'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning', 'IndentationError', 'IndexError', 'InterruptedError', 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'LookupError', 'MemoryError', 'ModuleNotFoundError', 'NameError', 'None', 'NotADirectoryError', 'NotImplemented', 'NotImplementedError', 'OSError', 'OverflowError', 'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError', 'RecursionError', 'ReferenceError', 'ResourceWarning', 'RuntimeError',

'RuntimeWarning', 'StopAsyncIteration', 'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'TimeoutError', 'True', 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning', 'ValueError', 'Warning', 'WindowsError', 'ZeroDivisionError', '__build_class__', '__debug__', '__doc__', '__import__', '__loader__', '__name__', '__package__', '__spec__', 'abs', 'all', 'any', 'ascii', 'bin', 'bool', 'breakpoint', 'bytearray', 'bytes', 'callable', 'chr', 'classmethod', 'compile', 'complex', 'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'exec', 'exit', 'filter', 'float', 'format', 'frozenset', 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass', 'iter', 'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview', 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'quit', 'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']

Cada elemento dessa lista é um módulo pré-alocado, pronta para uso. Importante lembrar que o que faz com que certas palavras e expressões fiquem reservadas ao sistema é justamente um módulo ou biblioteca carregados no builtin.

Não é possível excluir tudo o que está pré-carregado, mas só a fim de exemplo, se você excluísse esses módulos básicos o interpretador não teria parâmetros para verificar no código o que é o que de acordo com a sintaxe.

Repare na sintaxe, __builtins__ é um tipo de variável, mas ela é reservada do sistema, você não pode alterá-la de forma alguma, assim como outros elementos com prefixo ou sufixo __.

Importando bibliotecas

Em Python, de acordo com a nossa necessidade, existe a possibilidade de trabalharmos com as bibliotecas básicas já inclusas, ou importarmos outras bibliotecas que nos tragam novas funcionalidades.

Python como já dissemos anteriormente algumas vezes, é uma linguagem de programação "com pilhas inclusas", ou seja, as bibliotecas básicas que você necessita para grande parte das funções básicas já vem incluídas e pré-alocadas, de forma que basta chamarmos a função que queremos para de fato usá-la.

Porém, dependendo de casos específicos, podemos precisar fazer o uso de funções que não são nativas ou que até mesmo são de bibliotecas externas ao Python padrão, sendo assim necessário que importemos tais bibliotecas para acessarmos suas funções.

Em Python existem duas formas básicas de trabalharmos com as importações, de forma bastante simples, podemos importar uma biblioteca inteira a referenciando pelo nome após o comando import, ou podemos importar apenas algo de dentro de uma biblioteca externa para incorporarmos em nosso código, através do comando from (nome da biblioteca) import (nome da função). Por exemplo:

O Python em todas suas versões já conta com funções matemáticas pré-alocadas, porém para usarmos algumas funções ou constantes matemáticas, é necessário importá-las. Supondo que por algum motivo nosso código precise de pi, para que façamos o cálculo de alguma coisa. Podemos definir manualmente o valor de pi e atribuir a algum objeto, ou podemos importá-lo de alguma biblioteca externa.

No primeiro exemplo, podemos importar a biblioteca math, assim, quando referenciarmos o valor de pi ele já estará pré-alocado para uso, por exemplo.

```
1  import math
2
3  raio = 15.3
4
5  print('Area do circulo', math.pi * raio ** 2)
6
```

```
Area do circulo 735.4154242788347
```

O valor de pi, que faz parte da biblioteca math será multiplicado pelo raio e elevado ao quadrado. Como você está importando esse item da biblioteca math, não é necessário especificar o seu valor, neste caso, o valor de pi (3,1416) já está associado a pi.

O resultado será: 735.41

No segundo exemplo, podemos importar apenas a função pi de dentro da biblioteca math, da seguinte forma.

```
1  from math import pi
2
3  raio = 15.3
4
5  print('Area do circulo', pi * raio ** 2)
6
```

```
Area do circulo 735.4154242788347
```

O valor de pi será multiplicado pelo raio e elevado ao quadrado.

O retorno será: 735.41

Por fim, apenas para concluir o nosso raciocínio, uma biblioteca é um conjunto de parâmetros e funções já prontas para facilitar a nossa vida, aqui, podemos definir manualmente o valor de pi ou usar ele já pronto de dentro de uma biblioteca.

Apenas faça o seguinte exercício mental, imagine ter que programar

manualmente todas as funções de um programa (incluindo as interfaces, entradas e saídas, etc...), seria absolutamente inviável.

Todas linguagens de alto nível já possuem suas funções de forma pré-configuradas de forma que basta incorporá-las ao nosso builtin e fazer o uso.

MÓDULOS E PACOTES

Um módulo, na linguagem Python, equivale ao método de outras linguagens, ou seja, o programa ele executa dentro de um módulo principal e à medida que vamos o codificando, ele pode ser dividido em partes que podem ser inclusive acessadas remotamente.

Um programa bastante simples pode rodar inteiro em um módulo, mas conforme sua complexidade aumenta, e também para facilitar a manutenção do mesmo, ele começa a ser dividido em partes.

Já uma função inteira que você escreve e define, e que está pronta, e você permite que ela seja importada e usada dentro de um programa, ela recebe a nomenclatura de um pacote.

Por padrão, implícito, quando você começa a escrever um programa do zero ele está rodando como modulo __main__. Quem vem de outras linguagens de programação já está familiarizado a, quando se criava a estrutura básica de um programa, manualmente criar o método main.

Em Python essa e outras estruturas básicas já estão pré-definidas e funcionando por padrão, inclusive fazendo a resolução dos nomes de forma interna e automática.

Modularização

Uma vez entendido como trabalhamos com funções, sejam elas pré definidas ou personalizadas, hora de entendermos o que significa modularizações em programação.

Apesar do nome assustar um pouco, um módulo nada mais é do que pegarmos blocos de nosso código e o salvar em um arquivo externo, de forma que podemos importar o mesmo posteriormente conforme nossa necessidade.

Raciocine que como visto em capítulos anteriores, uma função é um bloco de código que está ali pré-configurado para ser usado dentro daquele código, mas e se pudéssemos salvá-lo de forma a reutilizar o mesmo em outro código, outro programa até mesmo de um terceiro. Isto é possível simplesmente o transformando em um módulo.

Como já mencionado anteriormente diversas vezes, Python é uma linguagem com pilhas inclusas, e isto significa que ela já nos oferece um ambiente pré-configurado com uma série de bibliotecas e módulos pré carregados e prontos para uso.

Além disto é possível importar novos módulos para adicionar novas funcionalidades a nosso código assim como também é possível criarmos um determinado bloco de código e o exportar para que se torne um módulo de outro código.

Seguindo com nosso raciocínio aqui é onde começaremos a trabalhar mais diretamente importando e usando o conteúdo disponível em arquivos externos, e isto é muito fácil de se fazer desde que da maneira correta.

Em modularização tudo começa com um determinado bloco de código que se tornará um módulo, código pronto e revisado, sem erros e pronto para executar a sua função desejada, salve o mesmo com um nome de fácil acesso e com a extensão .py. Em Python todo arquivo legível pela IDE recebe inicialmente a extensão reservada ao sistema .py.

Posteriormente compilando o executável de seu programa isto pode ser alterado, mas por hora, os arquivos que estamos trabalhando recebem a

extensão .py.

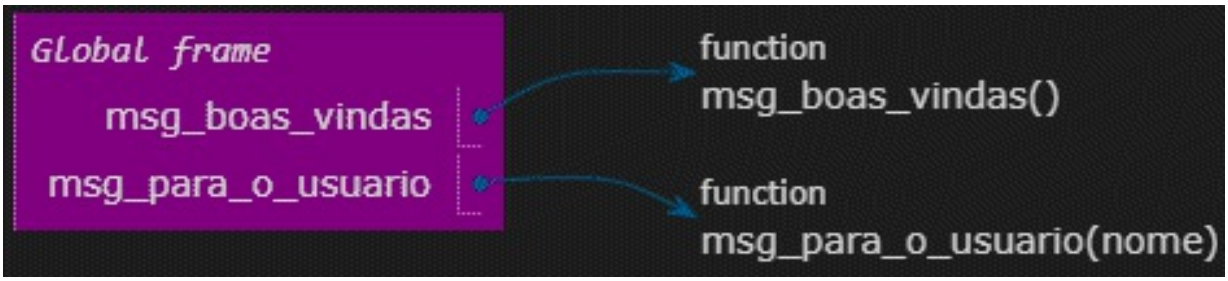
Partindo para prática:

Vamos modularizar o código abaixo:

```
1  def msg_boas_vindas():
2      print('Seja Muito Bem Vindo ao Meu Programa')
3
4  def msg_para_o_usuario(nome):
5      print(f'{nome}, espero que esteja tendo uma boa experiência...')
6
```

Salve este bloco de código com um nome de fácil identificação, por exemplo, boasvindas.py

Representação visual:



Agora vamos importar esse módulo para dentro de outro arquivo de seu código. Em sua IDE, abra ou crie outro arquivo em branco qualquer, em seguida vamos usar o comando import para importar para dentro desse código o nosso módulo previamente salvo boasvindas.py.

```
1  import boasvindas
2
```

Em seguida, importado o módulo, podemos dar comandos usando seu nome seguido de uma instrução, por exemplo ao digitarmos boasvindas e inserirmos um ponto, a IDE irá nos mostrar que comandos podemos usar de dentro dele, no nosso caso, usaremos por enquanto a primeira opção, que se refere a função msg_boas_vindas( ).

```
1  import boasvindas
2
3  boasvindas.msg_boas_vindas()
4
```

Se mandarmos rodar o código, o retorno será: Seja Muito Bem Vindo ao Meu Programa

Seguindo com o uso de nosso módulo, temos uma segunda função criada que pode receber uma string como argumento.

Então pela lógica usamos o nome do módulo, “ponto”, nome da função e passamos o argumento. Por exemplo a string ‘Fernando’.

```
1  import boasvindas
2
3  boasvindas.msg_boas_vindas()
4  boasvindas.msg_para_o_usuario('Fernando')
5
```

O Retorno será: Seja Muito Bem Vindo ao Meu Programa

Fernando, espero que esteja tendo uma boa experiência...

Outro exemplo, supondo que estamos criando uma calculadora, onde para obtermos uma performance melhor (no que diz respeito ao uso de memória), separamos cada operação realizada por essa calculadora em um módulo diferente.

Dessa forma, no corpo de nossa aplicação principal podemos nos focar ao código que irá interagir com o usuário e chamar módulos e suas funções enquanto as funções em si só são executadas de forma independente e no momento certo.

Ex: Inicialmente criamos um arquivo de nome soma.py que ficará responsável por conter o código da função que irá somar dois números.

```
1  def soma(num1, num2):
2      s = int(num1) + int(num2)
3      return s
4
```

Repare no código, dentro desse arquivo temos apenas três linhas que contém tudo o que precisamos, mas em outras situações, a ideia de modularizar blocos de código é que eles podem ser bastante extensos dependendo de suas funcionalidades.

Aqui apenas criamos uma função de nome soma( ) que recebe como

parâmetros duas variáveis num1 e num2, internamente a variável temporária s realiza a operação de somar o primeiro número com o segundo e guardar em si esse valor. Por fim, esse valor é retornado para que possa ser reutilizado fora dessa função.

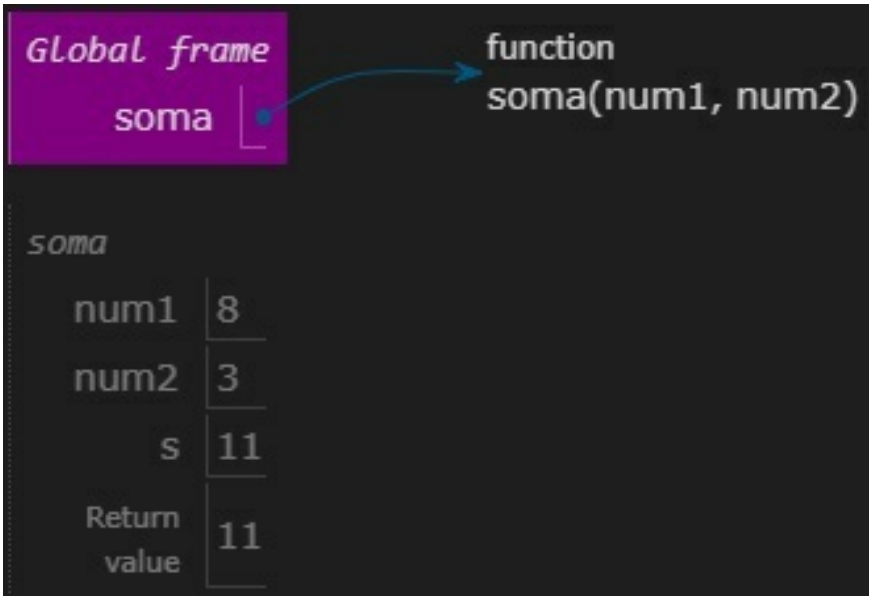
Agora criamos um arquivo de nome index.py e dentro dele o seguinte código:

```
1  import soma
2
3  print(f'O resultado da soma é: {soma.soma(8,3)}')
4
```

Note que inicialmente estamos importando o arquivo soma.py (na importação não é necessário o uso da extensão do arquivo).

Por fim criamos uma mensagem onde na máscara de substituição estamos executando a função soma e passando como parâmetros (para num1 e num2) os números 8 e 3. Nesse caso o retorno será 11.

Representação visual:



Caso você queira criar a interação com o usuário, para nesse exemplo anterior, que o mesmo digite os números, esse processo deve ser feito dentro do módulo da função.

Em soma.py

```
1  def soma(num1, num2):
2      num1 = input('Digite um número: ')
3      num2 = input('Digite outro número: ')
4      s = int(num1) + int(num2)
5      return s
6
```

Em index.py

```
1  import soma
2
3  print(f'O resultado da soma é: {soma.soma(0,0)}')
4
```

Apenas note que este código é idêntico ao anterior, e na execução do programa, esses valores 0 e 0 definidos aqui serão substituídos e processados pelos valores que o usuário digitar.

Uma dúvida bastante comum a quem está aprendendo a programar em Python é se precisamos usar a extensão do arquivo quando importamos algum módulo e a resposta é não.

Porém, o arquivo de módulo em questão precisa estar na mesma pasta em que estamos trabalhando com os demais arquivos, para que a IDE o ache e consiga o importar.

Na verdade, o Python inicialmente procurará o arquivo em questão no diretório atual, se não encontrar ele buscará o arquivo nos diretórios salvos como PATH. O recomendável é que você mantenha seus arquivos agrupados em um diretório comum.

Outro conceito importante de ser citado a esta altura é o conceito de Pacotes. Em Python não existe uma grande distinção entre esta forma de se trabalhar com módulos se comparado a outras linguagens de programação.

Raciocine que, à medida que você for modularizando seu código, é interessante também dividir categorias de módulos em pastas para melhor

organização em geral.

Toda vez que criarmos pastas que guardam módulos dentro de si, em Python costumamos chamar isso de um pacote.

Raciocine também que quanto mais modularizado e organizado é seu código, mais fácil será fazer a manutenção do mesmo ou até mesmo a implementação de novas funcionalidades porque dessa forma, cada funcionalidade está codificada em um bloco independente, e assim, algum arquivo corrompido não prejudica muito o funcionamento de todo o programa.

Por fim, na prática, a grande diferença que haverá quando se trabalha com módulos agrupados por pacotes será a forma com que você irá importar os mesmos para dentro de seu código principal.

Importando de módulos

```
1  import soma
2
```

Importando o módulo soma (supondo que o mesmo é um arquivo de nome soma.py no mesmo diretório).

```
1  import soma as sm
2
```

Importando soma e o referenciando como sm por comodidade.

```
1  from soma import calculo_soma
2
```

Importando somente a função calculo_soma do módulo soma.

Importando de pacotes:

```
1  import calc.calculadora_soma
2
```

Importando calculadora_soma que faz parte do pacote calc. Essa sintaxe calc . calculadora_soma indica que calc é uma subpasta onde se encontra o módulo calculadora_soma.

```
1  from calc.calculadora_soma import soma
2
```

Importando a função soma, do módulo calculadora_soma, do pacote calc.

```
1  from calc.calculadora_soma import soma as sm
2
```

Importando a função soma do módulo calculadora_soma do pacote calc e a referenciando como sm.

Por fim, tenha em mente que todo programa, dependendo claro de sua complexidade, pode possuir incontáveis funcionalidades explícitas ao usuário, mas que o programador definiu manualmente cada uma delas e as categorizou, seguindo uma hierarquia, em funções, módulos, pacotes e bibliotecas.

Você pode fazer o uso de toda essa hierarquia criando a mesma manualmente ou usando de funções/módulos/pacotes e bibliotecas já prontas e pré-configuradas disponíveis para o Python.

O uso de uma biblioteca inteira ou partes específicas dela tem grande impacto na performance de seu programa, pelas boas práticas de programação, você deve ter na versão final e funcional de seu programa apenas aquilo que interessa, modularizar os mesmos é uma prática comum para otimizar o código.

Ex: Criamos um pacote de nome vendas (uma pasta no diretório com esse nome) onde dentro criamos o arquivo calc_preco.py que será um módulo.Na sequência criamos uma função interna para calc_preco. Aqui, apenas para fins de exemplo, uma simples função que pegará um valor e aplicará um aumento percentual.

Em vendas\calc_preco.py

```
1  def aum_preco(preco, porcentagem):
2      npreco = preco + (preco * (porcentagem / 100))
3      return npreco
4
```

Em index.py

```
1  import vendas.calc_preco
2
3  preco = 19.90
4  preco_novo = vendas.calc_preco.aum_preco(preco, 4)
5
6  print(f'O valor corrigido é: {preco_novo}')
7
```

Repare que pela sintaxe, inicialmente importamos o módulo calc_preco do pacote vendas. Em seguida, como já fizemos outras vezes em outros exemplos, usamos essa função passando parâmetros (um valor inicial e um aumento, nesse caso, de 4%) e por fim exibimos em tela o resultado dessa função.

Poderíamos otimizar esse código da seguinte forma:

```
1  from vendas.calc_preco import aum_preco as ap
2
3  preco = 19.90
4  preco_novo = ap(preco, 4)
5
6  print(f'O valor corrigido é: {preco_novo}')
7
```

O resultado seria o mesmo, com um ganho de performance por parte do processamento. Nesse exemplo, o retorno seria: 20.69.

Representação visual:



Apenas a nível de curiosidade, se você consultar a documentação do Python verá uma infinidade de códigos prontos e pré-configurados para uso.

Dependendo o uso, Python tem à disposição bibliotecas inteiras, que contém dentro de si inúmeros pacotes, com inúmeros módulos, com inúmeras funções a disposição, bastando as importar da maneira correta e incorporar em seu código.

PROGRAMAÇÃO ORIENTADA A OBJETOS

*Fundamentos

Classes

Antes de mais nada é importante salientar aqui que, em outras linguagens de programação quando começamos a nos aprofundar nos estudos de classes normalmente há uma separação desde tipo de conteúdo dos demais por estar entrando na área comumente chamada de orientação a objetos

Em Python não há necessidade de fazer tal distinção uma vez que toda a linguagem em sua forma já é nativa orientada a objetos, logo, progressivamente fomos avançando em seus conceitos e avançaremos muito mais sem a necessidade dessa divisão.

Apenas entenda que classes estão diretamente ligadas a programação orientada a objeto, que nada mais é uma abstração de como lidaremos com certos tipos de problemas em nosso código, criando e usando objetos de uma forma mais complexa, a partir de estruturas “moldes”.

Uma classe dentro das linguagens de programação nada mais é do que um objeto que ficará reservado ao sistema tanto para indexação quanto para uso de sua estrutura, é como se criássemos uma espécie de molde de onde podemos criar uma série de objetos a partir desse molde.

Assim como podemos inserir diversos outros objetos dentro deles de forma que fiquem instanciáveis (de forma que permita manipular seu conteúdo), modularizados, oferecendo uma complexa, mas muito eficiente maneira de se trabalhar com objetos.

Parece confuso, mas na prática é relativamente simples, tenha em mente que para Python toda variável é um objeto, a forma como lhe instanciamos e/ou irá fazer com que o interpretador o trate com mais ou menos privilégios dentro do código.

O mesmo acontece quando trabalharmos com classes. Porém o fato de haver este tipo de dado específico “classe” o define como uma estrutura lógica modelável e reutilizável para nosso código.

Uma classe fará com que uma variável se torne de uma categoria reservada ao sistema para que possamos atribuir dados, valores ou parâmetros de maior complexidade, é como se transformássemos uma simples variável

em uma super variável, de maior possibilidade de recursos e de uso.

Muito cuidado para não confundir as coisas, o que é comum de acontecer no seu primeiro contato com programação orientada a objetos, raciocine de forma direta que uma classe será uma estrutura molde que poderá comportar dados dentro de si assim como servir de molde para criação de novas variáveis externas.

Pela sintaxe convencionalmente usamos o comando class (palavra reservada ao sistema) para especificar que a partir deste ponto estamos trabalhando com este tipo de dado, na sequência definimos um nome para essa classe, onde por convenção, é dado um nome qualquer desde que o mesmo se inicie com letra maiúscula.

Por exemplo:

```
1  class Carro:
2      carro1 = 'Gol modelo 2016 completo'
3      carro2 = 'Celta ano 2015 4 portas'
4      carro3 = 'Uno ano 2015 baixa quilometragem'
5      carro4 = 'Clio 2018 flex doc vencido'
6
7  print(Carro.carro1)
8
```

```
Gol modelo 2016 completo
```

O Retorno será: Gol modelo 2016 completo

Inicialmente foi criada a classe Carro (class é uma palavra reservada ao sistema e o nome da classe deve ser case sensitive, ou seja, iniciar com letra maiúscula), dentro dessa classe foram inseridos 4 carros com suas respectivas características.

Por fim, o comando print( ) mandou exibir em tela a instância carro1 que pertence a Carro. Seguindo a lógica do conceito explicado anteriormente, Carro é uma super variável que tem carro1 (uma simples variável) contida nele.

O prefixo class faz com que o interpretador reconheça essa hierarquia de variáveis e trabalhe da maneira correta com elas.

Abstraindo esse exemplo para simplificar seu entendimento, Carro é uma classe/categoria/tipo de objeto, dentro dessa categoria existem diversos tipos de carros, cada um com seu nome e uma descrição relacionada ao seu modelo.

Representação visual:



Para entendermos de forma prática o conceito de uma “super variável”, podemos raciocinar vendo o que ela tem de diferente de uma simples variável contendo dados de qualquer tipo.

Raciocine que pelo básico de criação de variáveis, quando precisávamos criar, por exemplo, uma variável pessoa que dentro de si, como atributos, guardasse dados como nome, idade, sexo, etc...

isto era feito transformando esses dados em uma lista ou dicionário, ou até mesmo criando várias variáveis uma para cada tipo de dado, o que na prática, em programas de maior complexidade, não é nada eficiente.

Criando uma classe Pessoa, poderíamos da mesma forma ter nossa variável pessoa, mas agora instanciando cada atributo como objeto da classe Pessoa, teríamos acesso direto a esses dados posteriormente, ao invés de buscar dados de índice de um dicionário por exemplo.

Dessa forma, instanciar, referenciar, manipular esses dados a partir de uma classe também se torna mais direto e eficiente a partir do momento que você domina essa técnica.

Ex:

```
1  class Pessoa:
2      pass
3
4  pessoa1 = Pessoa()
5
6  pessoa1.nome = 'Fernando'
7  pessoa1.idade = 32
8
9  print(pessoa1.nome)
10 print(pessoa1.idade)
11
```

```
Fernando
32
```

Apenas iniciando o entendimento desse exemplo, inicialmente definimos a classe Pessoa que por sua vez está vazia de argumentos. Em seguida criamos a variável pessoa1 que recebe como atribuição a classe Pessoa( ).

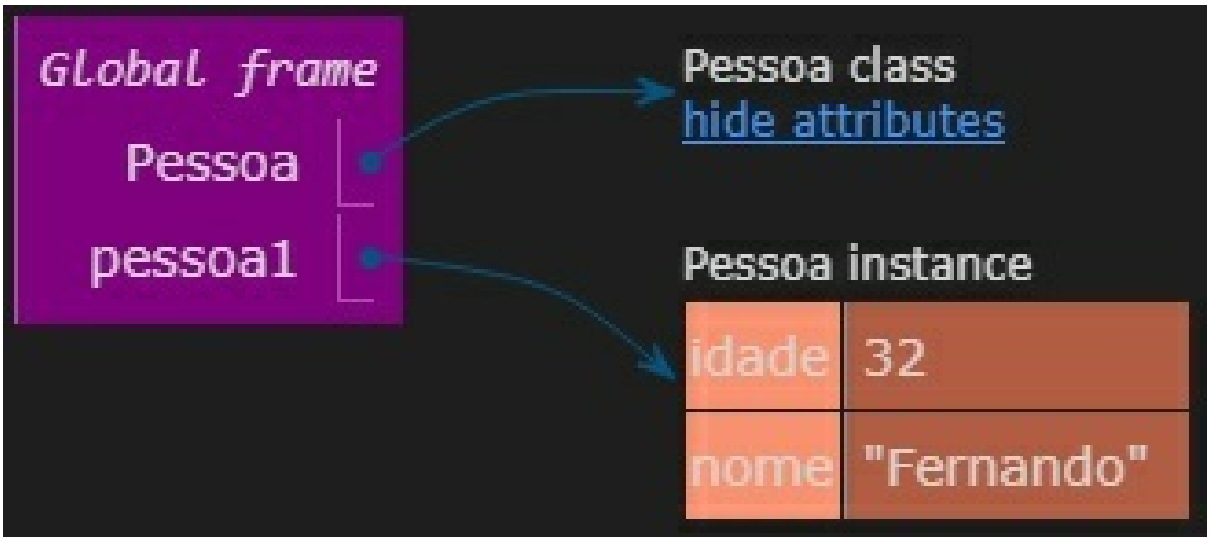
A partir desse ponto, podemos começar a inserir dados (atributos) que ficarão guardados na estrutura dessa classe. Simplesmente aplicando sobre a variável o comando pessoa1.nome = ‘Fernando’ estamos criando dentro dessa variável pessoa1 a variável nome que tem como atributo a string ‘Fernando’.

Da mesma forma, dentro da variável pessoa1 é criada a variável idade = 32. Note que pessoa1 é uma super variável por conter dentro de si outras variáveis com diferentes dados/valores/atributos... Por fim, passando como parâmetro para nossa função print( ) pessoa1.nome e pessoa1.idade.

O retorno será: Fernando

32

Representação visual:



No exemplo anterior, bastante básico, a classe em si estava vazia, o que não é o convencional, mas ali era somente para exemplificar a lógica de guardar variáveis e seus respectivos atributos dentro de uma classe.

Normalmente a lógica de se usar classes é justamente que elas guardem dados e se necessário executem funções a partir dos mesmos. Apenas para entender o básico sobre esse processo, analise o bloco de código seguinte:

```
1  class Pessoa:
2      def acao1(self):
3          print('Ação 1 está sendo executada...')
4
5  pessoa1 = Pessoa()
6
7  pessoa1.acao1()
8
```

```
Ação 1 está sendo executada...
```

Criada a classe Pessoa, dentro dela é definida a função acao1(self) que por sua vez exibe em tela uma mensagem. Na sequência é criada a variável pessoa1 que tem como atribuição tudo o que estiver dentro da classe Pessoa.

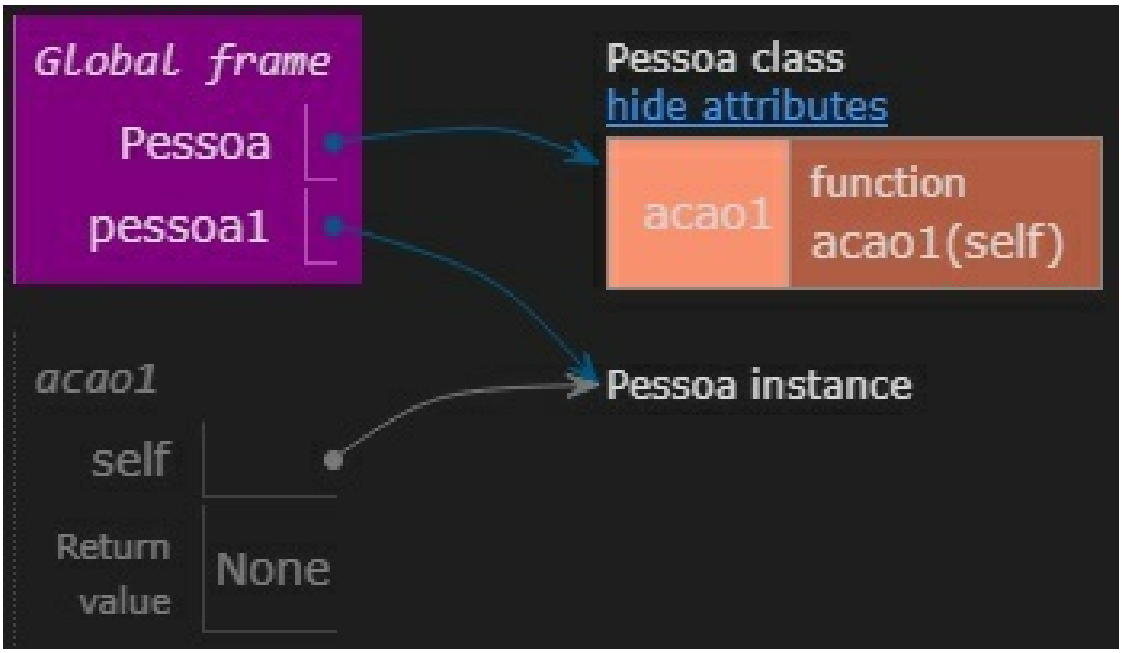
Assim como no exemplo anterior adicionamos variáveis com

atributos dentro dessa classe a partir da variável inicial, podemos também chamar uma função interna da classe sobre essa variável diretamente aplicando sobre si o nome da função, nesse caso pessoa1.acao1( ) executará a função interna na respectiva variável.

O retorno será: Ação 1 está sendo executada...

Por fim, note que para tal feito simplesmente referenciamos a variável pessoa1 e chamamos diretamente a função .acao1( ) sobre ela.

Representação visual:



Definindo uma classe

Em Python podemos manualmente definir uma classe respeitando sua sintaxe adequada, seguindo a mesma lógica de sempre, há uma maneira correta de identificar tal tipo de objeto para que o interpretador o reconheça e trabalhe com ele.

Basicamente quando temos uma função dentro de uma classe ela é chamada de método dessa classe, que pode executar qualquer coisa.

Outro ponto é que quando definimos uma classe manualmente começamos criando um construtor para ela, uma função que ficará reservada ao sistema e será chamada sempre que uma instância dessa classe for criada/usada para que o interpretador crie e use a instância da variável que usa essa classe corretamente.

Partindo para prática, vamos criar do zero uma classe chamada Usuario:

```
1  class Usuario:
2      def __init__(self, nome, idade):
3          self.nome = nome
4          self.idade = idade
5
6      def boas_vindas(self):
7          print(f'Usuário: {self.nome}, Idade: {self.idade}')
8
9  usuario1 = Usuario(nome='Fernando', idade='31')
10 usuario1.boas_vindas()
11
12 print(usuario1.nome)
13
```

```
Usuário: Fernando, Idade: 31
Fernando
```

Repare que o código começa com a palavra reservada class seguida do nome da classe que estamos definindo, Usuario, que pela sintaxe o nome de uma classe começa com letra maiúscula.

Em seguida é declarado e definido o construtor da classe __init__ , que sempre terá self como parâmetro (instância padrão), seguido de quantos parâmetros personalizados o usuário criar, desde que separados por vírgula.

Pela indentação existem duas chamadas de self para atribuir um nome a variável nome e uma idade a variável idade.

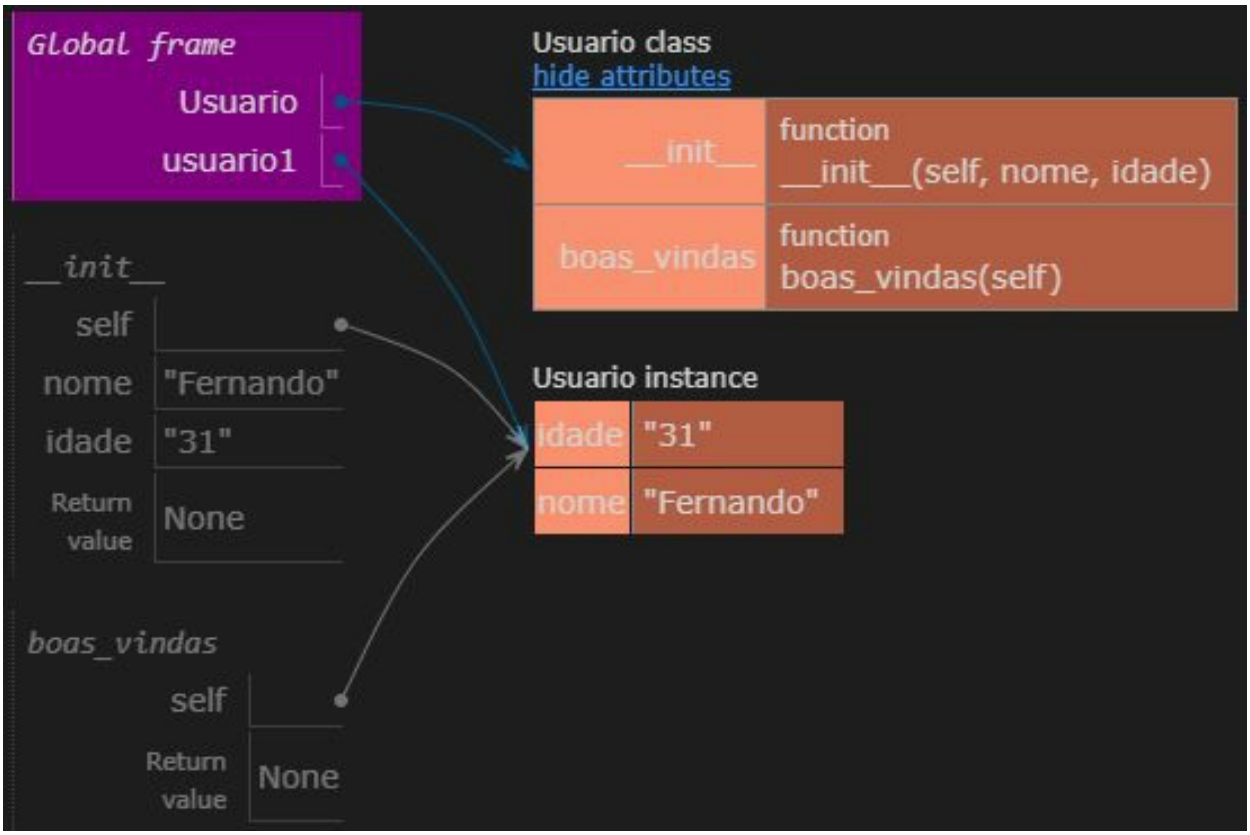
Na sequência há uma nova função apenas chamando a função print para uma string que interage com as variáveis temporárias declaradas anteriormente. Por fim é criada uma variável usuario1 que recebe a classe Usuário e a atribui dados para nome e idade.

Na execução de usuario1.boas_vindas( ) tudo construído e atribuído até o momento finalmente irá gerar um retorno ao usuário. Há ainda uma linha adicional que apenas chama a função boas_vindas para a variável usuário1, apenas por convenção mesmo.

O retorno será: Usuário: Fernando, Idade: 31

Fernando

Representação visual:



Alterando dados/valores de uma instância

Muito bem, mas e se em algum momento você precisa alterar algum dado ou valor de algo instanciado dentro de uma classe, isto é possível? Sim e de forma bastante fácil, por meio de atribuição direta ou por intermédio de funções que são específicas para manipulação de objetos de classe.

Exemplo de manipulação direta:

```
1  usuario1 = Usuario(nome='Fernando', idade='31')
2  usuario1.nome = 'Fernando Feltrin'
3  usuario1.boas_vindas()
4
5  print(usuario1.nome)
6
```

```
Usuário: Fernando Feltrin, Idade: 31
Fernando Feltrin
```

Aproveitando um trecho do código anterior, repare que na terceira linha é chamada a variável usuario1 seguida de .nome atribuindo uma nova string ‘Fernando Feltrin’. Ao rodar novamente este código o retorno será: Usuário: Fernando Feltrin, Idade: 31

Exemplo de manipulação via função específica:

```
1  usuario1 = Usuario(nome='Fernando', idade='31')
2  usuario1.nome = 'Fernando Feltrin'
3
4  setattr(usuario1, 'nome', 'Fernando B. Feltrin')
5  delattr(usuario1, 'idade',)
6  setattr(usuario1, 'idade', '32')
7
8  print(usuario1.nome)
9  print(usuario1.idade)
10
```

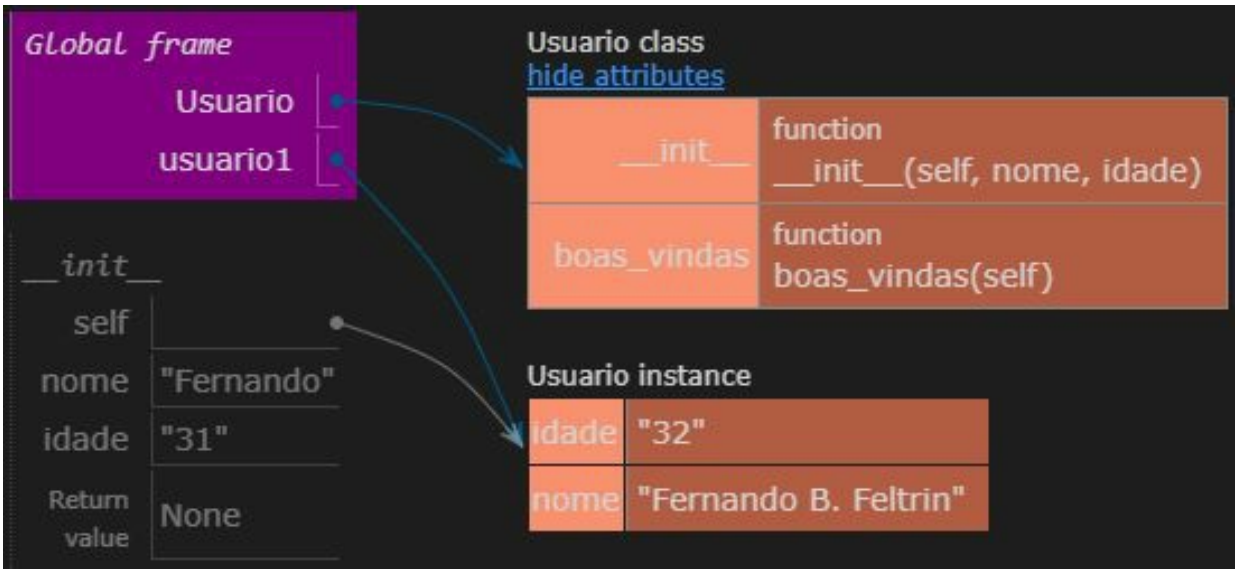
```
Fernando B. Feltrin
32
```

Repare que estamos trabalhando ainda no mesmo trecho de código anterior, porém agora na terceira linha temos a função setattr( ) que tem como parâmetro a variável usuario1 e irá pegar sua instância nome e alterar para nova string ‘Fernando B. Feltrin’.

Logo em seguida existe a função delattr( ) que pega a variável usuario1 e deleta o que houver de dado sob a instância idade, logo na sequência uma nova chamada da função setattr( ) irá pegar novamente a variável usuario1, agora na instância idade e irá atribuir o novo valor, ‘32’, em forma de string mesmo.

Por fim, dando os respectivos comandos print para que sejam exibidas as instâncias nome e idade da variável usuario1 o retorno serão os valores atualizados. Neste caso o retorno será: Usuário Fernando B. Feltrin, Idade: 32

Representação visual:



Em suma, quando estamos trabalhando com classes existirão funções específicas para que se adicionem novos dados (função setattr(variavel, ‘instancia’, ‘novo dado’)), assim como para excluir um determinado dado interno de uma classe (função delattr(variavel, ‘instancia’)).

Aplicando recursividade

O termo recursividade em linguagens de programação diz respeito à quando um determinado objeto ou função chamar ele mesmo dentro da execução de um bloco de código executando sobre si um incremento.

Imagine que você tem um objeto com dois parâmetros, você aplicará uma condição a ser atingida, porém para deixar seu código mais enxuto você passará funções que retornam e executam no próprio objeto. Por exemplo:

```
1  def imprimir(maximo, atual):
2      if atual >= maximo:
3          return
4      print(atual)
5      imprimir(maximo, atual + 1)
6
7  imprimir(10, 1)
8
```

```
1
2
3
4
5
6
7
8
9
```

O resultado será: 1

2

3

4

5

6

7

8

9

Repare que o que foi feito é que definimos um objeto imprimir onde como parametros (variáveis) temos maximo e atual, em seguida colocamos a condição de que se atual for maior ou igual a maximo, pare a execução do código através de seu retorno. Se essa condição não for atingida, imprima atual, em sequida chame novamente o objeto imprimir, mudando seu segundo parâmetro para atual + 1.

Por fim, definimos que imprimir recebe como argumentos (o que iremos atribuir a suas variáveis) 10 para maximo e 1 para atual. Enquanto atual não for maior ou igual a 10 ele ficará repetindo a execução desse objeto o incrementando em 1.

Em outras palavras, estamos usando aquele conceito de incremento, visto em for anteriormente, mas aplicado a objetos.

Existe a possibilidade de deixar esse código ainda mais enxuto em sua recursividade, da seguinte forma:

```python
def imprimir(maximo, atual):
    if atual < maximo:
        print(atual)
        imprimir(maximo, atual + 1)

imprimir(10, 1)

```

```
1
2
3
4
5
6
7
8
9
```

O Retorno será: 1

2

3

4

5

6

7

8

9

Representação visual:


Global frame
imprimir
function
imprimir(maximo, atual)
imprimir
maximo 10
atual 1
Return value None
imprimir
maximo 10
atual 5
Return value None
imprimir
maximo 10
atual 2
Return value None
imprimir
maximo 10
atual 6
Return value None
imprimir
maximo 10
atual 3
Return value None
imprimir
maximo 10
atual 7
Return value None
imprimir
maximo 10
atual 4
Return value None
imprimir
maximo 10
atual 8
Return value None
imprimir
maximo 10
atual 9
Return value None

Herança

Basicamente quando falamos em herança, a lógica é exatamente a mesma de uma herança “na vida real”, se sou herdeiro de meus pais, automaticamente eu herdo certas características e comportamentos dos mesmos.

Em programação a herança funciona da mesma forma, temos um objeto com capacidade de herdar certos parametros de outro, na verdade uma classe que herda métodos, parâmetros e outras propriedades de sua classe referência.

Imagine que você tem um objeto Carro com uma série de características (modelo, ano, álcool ou gasolina, manual ou automático), a partir dele é criado um segundo objeto Civic, que é um Carro inclusive com algumas ou todas suas características. Dessa forma temos objetos que herdam características de objetos anteriores.

Em Python, como já vimos anteriormente, a estrutura básica de um programa já é pré configurada e pronta quando iniciamos um novo projeto em nossa IDE, fica subentendido que ao começar um projeto do zero estamos trabalhando em cima do método principal do mesmo, ou método __main__.

Em outras linguagens inclusive é necessário criar tal método e importar bibliotecas para ele manualmente. Por fim quando criamos uma classe que será herdada ou que herdará características, começamos a trabalhar com o método __init__ que poderá, por exemplo, rodar em cima de __main__ sem substituir suas características, mas sim adicionar novos atributos específicos.

Na prática, imagine que temos uma classe Carros, e como herdeira dela teremos duas novas classes chamadas Gol e Corsa, que pela sintaxe deve ser montada com a seguinte estrutura lógica:

```
1  class Carros():
2      def __init__(self, nome, cor):
3          self.nome = nome
4          self.cor = cor
5
6      def descricao(self):
7          print(f'O carro: {self.nome} é {self.cor}')
8
9  class Gol(Carros):
10     def __init__(self, nome, cor):
11         super().__init__(nome, cor)
12
13 class Corsa(Carros):
14     def __int__(self, nome ,cor):
15         super().__init__(nome, cor)
16
```

Repare que primeiro é declarada a classe Carros, sem parâmetro nenhum, porém com um bloco de código indentado, ou seja, que pertence a ele.

Dentro deste bloco existe o construtor com os tipos de dados aos quais farão parte dessa classe, neste caso, iremos atribuir informações de nome e de cor a esta classe.

Há um segundo método definido onde é uma declaração de descrição, onde será executada a impressão de uma string que no corpo de sua sentença irá substituir as máscaras pelas informações de nome e cor que forem repassadas.

Então é criada a classe Gol que tem como parâmetro Carros, sendo assim, Gol é uma classe filha de Carros.

Na linha de código indentada existe o construtor e a função super( ) que é responsável por herdar as informações contidas na classe mãe. O mesmo é feito com a classe filha Corsa.

A partir disto, de toda essa estrutura montada, é possível criar variáveis que agora sim atribuirão dados de nome e cor para as classes anteriormente criadas.

```python
class Carros():
    def __init__(self, nome, cor):
        self.nome = nome
        self.cor = cor

    def descricao(self):
        print(f'O carro: {self.nome} é {self.cor}')

class Gol(Carros):
    def __init__(self, nome, cor):
        super().__init__(nome, cor)

class Corsa(Carros):
    def __int__(self, nome ,cor):
        super().__init__(nome, cor)

gol1 = Gol('Gol 2019 Completo', 'branco')
corsa2 = Corsa('Corsa 2017 2 Portas', 'vermelho')

print(gol1.descricao())
print(corsa2.descricao())

```

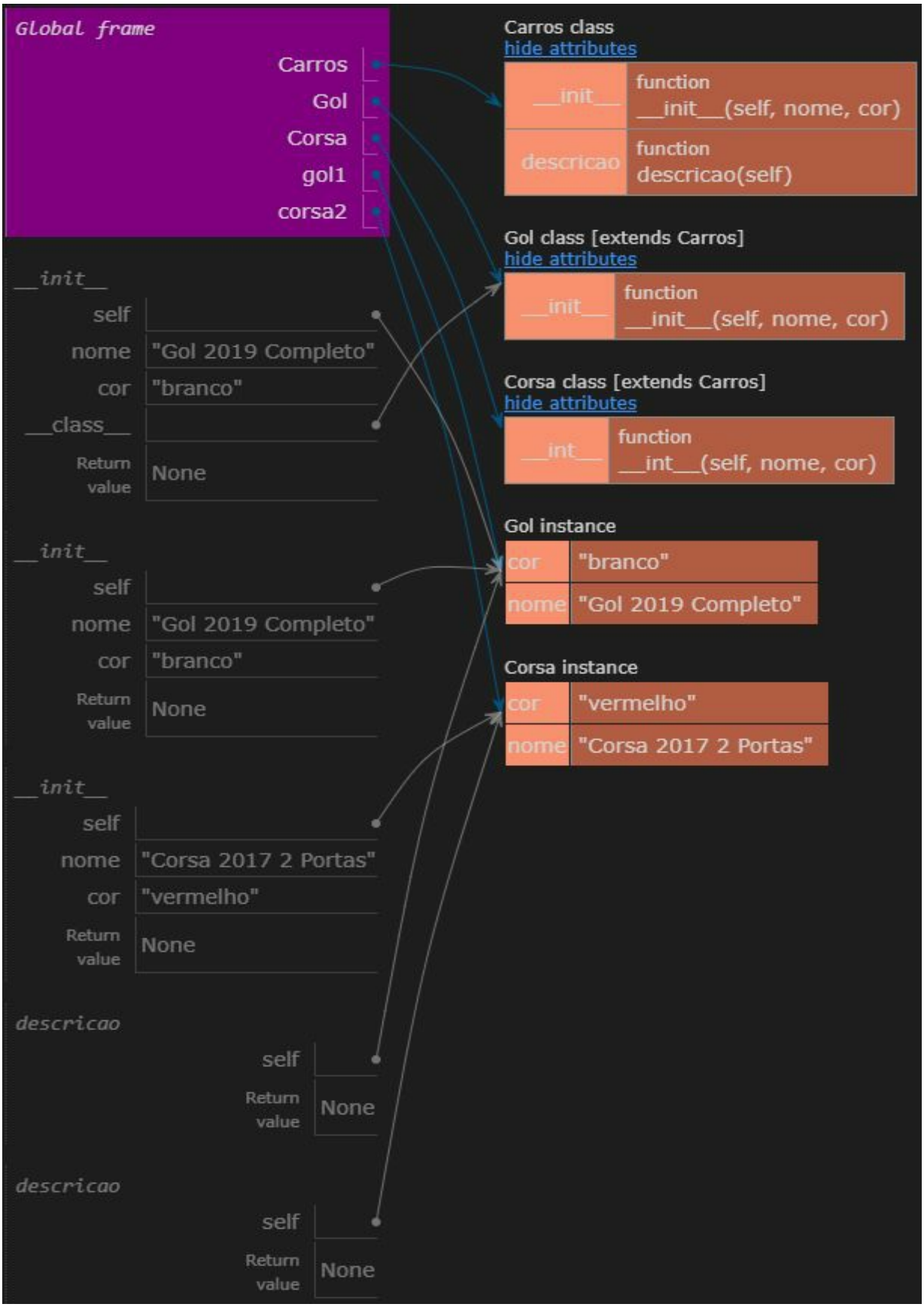
```
O carro: Gol 2019 Completo é branco
None
O carro: Corsa 2017 2 Portas é vermelho
```

Repare que é criada a variável gol1 que recebe a classe Gol e passa como parâmetro duas strings, que pela sequência serão substituídas por nome e cor na estrutura da classe. O mesmo é feito criando a variável corsa2 e atribuindo parametros a classe Corsa.

Por fim é dado o comando print, que recebe como parâmetro a variável gol1 seguido da função definida descricao. gol1 receberá os dados atribuídos a classe Gol e .descricao irá executar sua função print, substituindo as devidas máscaras em sua sentença.

O retorno será: O carro: Gol 2019 Completo é branco

O carro: Corsa 2017 2 Portas é vermelho


Global frame
Carros
Gol
Corsa
gol1
corsa2
__init__
self
nome "Gol 2019 Completo"
cor "branco"
__class__
Return value None
__init__
self
nome "Gol 2019 Completo"
cor "branco"
Return value None
__init__
self
nome "Corsa 2017 2 Portas"
cor "vermelho"
Return value None
descricao
self
Return value None
descricao
self
Return value None
Carros class
hide attributes
__init__ function __init__(self, nome, cor)
descricao function descricao(self)
Gol class [extends Carros]
hide attributes
__init__ function __init__(self, nome, cor)
Corsa class [extends Carros]
hide attributes
__int__ function __int__(self, nome, cor)
Gol instance
cor "branco"
nome "Gol 2019 Completo"
Corsa instance
cor "vermelho"
nome "Corsa 2017 2 Portas"

Em suma, é possível trabalhar livremente com classes que herdam características de outras, claro que estruturas de tamanha complexidade apenas serão usadas de acordo com a necessidade de seu código possuir de trabalhar com a manipulação de tais tipos de dados.

Polimorfismo

Polimorfismo basicamente é a capacidade de você reconhecer e reaproveitar as funcionalidades de um objeto para uma situação adversa, se um objeto que você já possui já tem as características que você necessita pra quê você iria criar um novo objeto com tais características do zero.

Na verdade, já fizemos polimorfismo no capítulo anterior enquanto usávamos a função super( ) que é dedicada a possibilitar de forma simples que possamos sobrescrever ou estender métodos de uma classe para outra conforme a necessidade.

```
1  class Carros():
2      def __init__(self, nome, cor):
3          self.nome = nome
4          self.cor = cor
5
6  class Gol(Carros):
7      def __init__(self, nome, cor):
8          super().__init__(nome, cor)
9
```

Repare que a classe filha Gol possui seu método construtor e logo em seguida usa da função super( ) para buscar de Carros os parâmetros que ela não tem.

Encapsulamento

O conceito de encapsulamento basicamente é que existe como um objeto e suas funcionalidades ficar encapsulado, fechado de forma que eu consiga instanciá-los sem mesmo conhecer seu conteúdo interior e também é uma forma de tornar um objeto privado, reservado ao sistema de forma que possamos usar livremente porém se o criamos e encapsulamos a ideia é que ele realmente seja imutável.

Principalmente quando usarmos um código de terceiros, muitas das vezes teremos o conhecimento daquele objeto, sua funcionalidade e conseguiremos incorporá-lo em seu código sem a necessidade de alterar ou configurar algo de seu conteúdo, apenas para uso mesmo.

Imagine uma cápsula de um determinado remédio, você em grande parte das vezes desconhece a maioria dos componentes químicos que estão ali dentro, porém você sabe o princípio ativo (para que serve) aquele remédio e o toma conforme a necessidade.

Porém, outro fator importante é que em outras linguagens de programação este conceito de tornar-se um objeto privado é muito levado a sério, de forma que em certos casos ele é realmente imutável.

Em Python, como ela é uma linguagem dinamicamente tipada, na verdade não existirão objetos com atributos privados ao sistema, até porque não há necessidade disso, você ter o controle sempre é importante, e quando necessário, bastará transformar um objeto em um __objeto__ para que o mesmo fique reservado ao sistema e ainda seja possível o modificar ou incrementar caso necessário.

```
1  objeto1 = 'Descrição por extenso'
2  #variável/objeto de uso comum.
3
4  __objeto1__ = 'Descrição por extenso'
5  #variável/objeto que está reservado ao sistema.
6
7  print(__objeto1__)
8
```

```
Descrição por extenso
```

O retorno será: Descrição por extenso

Como em Python tudo é objeto, tudo é dinâmico, e a linguagem coloca o controle total em suas mãos, há a convenção de alguns autores de que o encapsulamento em Python seria mais um aspecto estético (ao bater o olho em qualquer underline duplo __ saber que ali é algo reservado ao sistema) do que de fato ter de se preocupar com o acesso e a manipulação daquele tipo de variável/objeto, dado ou arquivo.

TRACEBACKS / EXCEÇÕES

Uma das situações que ainda não havíamos comentado, mas que certamente já ocorreu com você ao longo de seus estudos foi o fato de que em certas circunstâncias, quando houver algum erro de lógica ou de sintaxe, o interpretador irá gerar um código de erro.

Tais códigos em nossa IDE são chamados de tracebacks e eles tem a finalidade de tentar apontar ao usuário qual é o erro e em que linha do código o mesmo está ocorrendo.

Na prática grande parte das vezes um traceback será um erro genérico que apenas irá nos informar o erro, mas não sua solução.

Partindo para camada de software que o usuário tem acesso, nada pior do que ele tentar executar uma determinada função de seu programa e o mesmo apresentar algum erro e até mesmo travar e fechar sozinho. Lembre-se que sempre será um erro humano, de lógica ou de sintaxe.

Por exemplo:

```
 1  num1 = 13
 2  num2 = ad
 3
 4  soma = num1 + num2
 5
 6  print(soma)
 7
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call
<ipython-input-137-85e1e38a52fa> in <module>()
      1 num1 = 13
----> 2 num2 = ad
      3
      4 soma = num1 + num2
      5

NameError: name 'ad' is not defined
```

O retorno será:

Traceback (most recent call last):

File "C:/Users/Fernando/teste_001.py", line 2, in <module> num2 = ad

NameError: name 'ad' is not defined

Repare no código, declaradas duas variáveis num1 e num2 e uma terceira que faz a soma das duas anteriores, executado o comando print o retorno será um traceback.

Analisando o traceback ele nos mostra que na execução no nosso atual arquivo (no exemplo teste_001.py), na linha 2 o dado/valor ad não é reconhecido como nenhum tipo de dado.

Comandos try, except e finally

Pelas boas práticas de programação, uma solução elegante é prevermos os possíveis erros e/ou na pior das hipóteses apenas mostrar alguma mensagem de erro ao usuário, apontando que está havendo alguma exceção, algo não previsto durante a execução do programa.

Ainda trabalhando em cima do exemplo anterior, o traceback se deu pelo fato de estarmos tentando somar um int de valor 13 e um dado ad que não faz sentido algum para o interpretador.

Através do comando try (em tradução livre do inglês = tentar) podemos fazer, por exemplo, com que o interpretador tente executar a soma dos valores daquelas variáveis.

Se não for possível, será executado o comando except (em tradução livre = exceção), que terá um print mostrando ao usuário uma mensagem de erro e caso for possível realizar a operação o mesmo executará um comando finally (em tradução livre = finalmente) com o retorno e resultado previsto.

```
1
2  try:
3      num1 = int(input('Digite o primeiro numero: '))
4      num2 = int(input('Digite o segundo numero: '))
5
6  except:
7      print('Numero invalido, tente novamente;')
8
9  finally:
10     soma = int(num1) + int(num2)
11     print(f'O resultado da soma é: {soma}')
12
```

Repare que o comando inicial deste bloco de código é o try, pela indentação, note que é declarada a variavel num1 que pede para o usuário que digite o primeiro número, em seguida é declarada uma segunda variavel de nome num2 e novamente se pede para que o usuário digite o segundo número a ser somado.

Como já vimos anteriormente, o comando input aceita qualquer coisa que o usuário digitar, de qualquer tamanho, inclusive com espaços e comandos especiais, o input encerra sua fase de captação quando o usuário finalmente aperta ENTER.

Supondo que o usuário digitou nas duas vezes que lhe foi solicitado um número, o código irá executar o bloco finally, que por sua vez cria a variável temporária soma, faz a devida soma de num1 e num2 e por fim exibe em tela uma string com o resultado da soma.

Mas caso ainda no bloco try o usuário digitar algo que não é um número, tornando impossível a soma dos mesmos, isto irá gerar uma exceção, o bloco except é responsável por capturar esta exceção, ver que algo do bloco anterior está errado, e por fim, neste caso, exibe uma mensagem de erro previamente declarada.

Este é um exemplo de calculadora de soma de dois números onde podemos presumir que os erros que ocorrerão são justamente quando o usuário digitar algo fora da normalidade.

Importante salientar que se você olhar a documentação do Python em sua versão 3 você verá que existem vários muitos tipos de erro que você pode esperar em seu código e por fins de performance (apenas por curiosidade, Python 3 oferece reconhecimento a 30 tipos de erro possíveis), junto do comando except: você poderia declarar o tipo de erro para poupar processamento.

Supondo que é um programa onde o tipo de arquivo pode gerar uma exceção, o ideal, pelas boas práticas de programação seria declarar um except TypeError: assim o interpretador sabe que é previsto aquele tipo de erro em questão (de tipo) e não testa todos os outros possíveis erros.

Porém em seus primeiros programas não há problema nenhum em usar um except: que de forma básica chama esta função que irá esperar qualquer tipo de erro dentro de seu banco de dados sintático.

Na prática você verá que é bastante comum usarmos try e except em operações onde o programa interage com o usuário, uma vez que é ele que pode inserir um dado inválido no programa gerando erro.

AVANÇANDO EM POO

*Revisando fundamentos e aprofundando conceitos.

Em capítulos anteriores tivemos nosso primeiro contato com o paradigma da orientação a objetos em Python, agora iremos revisar tais conceitos vistos anteriormente e iremos nos aprofundar nesta metodologia de programação.

Não se assuste quanto a alguns tópicos parecerem se repetir ao longo dos capítulos subsequentes, em certos casos, usaremos de outra abordagem para tratar do mesmo assunto, de modo que você entenda por completo esta etapa um tanto quanto mais avançada de programação.

Objetos e Classes

Independente se você está começando do zero com a linguagem Python ou se você já programa nela ou em outras linguagens, você já deve ter ouvido a clássica frase “em Python tudo é objeto” ou “em Python não há distinção entre variável e objeto”.

De fato, uma das características fortes da linguagem Python é que uma variável pode ser simplesmente uma variável assim como ela pode “ganhar poderes” e assumir o papel de um objeto sem nem ao menos ser necessário alterar sua sintaxe.

Por convenção, dependendo que tipo de problema computacional que estaremos abstraindo, poderemos diretamente atribuir a uma variável/objeto o que quisermos, independente do tipo de dado, sua complexidade ou seu comportamento dentro do código. O interpretador irá reconhecer normalmente uma variável/objeto conforme seu contexto e será capaz de executá-lo em tempo real.

Outro ponto que precisamos começar a entender, ou ao menos por hora desmistificar, é que na programação orientada a objetos, teremos variáveis/objetos que receberão como atributo uma classe. Raciocine que na programação convencional estruturada você ficava limitado a variável = atributo.

Supondo que tivéssemos uma variável de nome lista1 e seu atributo fosse uma lista (tipo de dado lista), a mesma estaria explícita como atributo de lista1 e sua funcionalidade estaria ali definida (todas características de uma lista).

Agora raciocine que, supondo que houvéssemos um objeto de nome mercado1, onde lista1 fosse apenas uma das características/atributos internas desse objeto, essas demais estruturas estariam dentro de uma classe, que pode inclusive ser encapsulada ou modularizada para separar seu código do restante.

Em outras palavras, apenas tentando exemplificar a teoria por trás de uma classe em Python, uma classe é uma estrutura de dados que pode

perfeitamente guardar dentro de si uma infinidade de estruturas de dados, como variáveis e seus respectivos dados/valores/atributos, funções e seus respectivos parâmetros, estruturas lógicas e/ou condicionais, etc... tudo atribuído a um objeto facilmente instanciável, assim como toda estrutura dessa classe podendo ser reutilizada como molde para criação de novos objetos, ou podendo interagir com outras classes.

Como dito anteriormente, não se assuste com estes tópicos iniciais e a teoria envolvida, tudo fará mais sentido à medida que formos exemplificando cada conceito destes por meio de blocos de código.

Variáveis vs Objetos

Partindo para a prática, vamos inicialmente entender no código quais seriam as diferenças em um modelo estruturado e em um modelo orientado a objetos.

Para esse exemplo, vamos começar do básico apenas abstraindo uma situação onde simplesmente teríamos de criar uma variável de nome pessoa1 que recebe como atributos um nome e uma idade.

Inicialmente raciocine que estamos atribuindo mais de um dado (nome e idade) que inclusive serão de tipos diferentes (para nome uma string e para idade um int).

Convencionalmente isto pode ser feito sem problemas por meio de um dicionário, estrutura de dados em Python que permite o armazenamento de dados em forma de chave / valor, aceitando como dado qualquer tipo de dado alfanumérico desde que respeitada sua sintaxe:

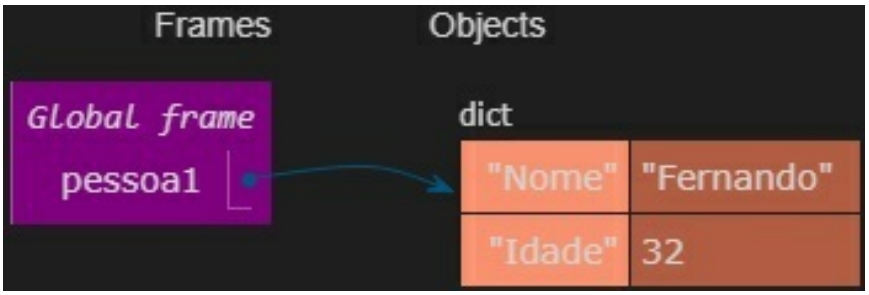
```
1  pessoa1 = {'Nome':'Fernando', 'Idade':32}
2  print(pessoa1)
3
```

```
{'Nome': 'Fernando', 'Idade': 32}
```

Então, finalmente dando início aos nossos códigos, inicialmente criamos na primeira linha uma variável de nome pessoa1 que por sua vez, respeitando a sintaxe, recebe um dicionário onde as chaves e valores do mesmo são respectivamente Nome : Fernando e Idade : 32.

Em seguida usamos a função print( ) para exibir o conteúdo de pessoa1.

O retorno será: Nome : Fernando, Idade : 32



Como dito anteriormente, nesses moldes, temos uma estrutura estática, onde podemos alterar/adicionar/remover dados manipulando o dicionário, mas nada além das funcionalidades padrão de um dicionário.

Partindo para um modelo orientado a objetos, podemos criar uma classe de nome Pessoa onde Nome e Idade podem simplesmente ser variáveis com seus respectivos atributos guardados lá dentro. E não nos limitamos somente a isso, podendo adicionar manualmente mais características a pessoa1 conforme nossa necessidade.

```
1  class Pessoa:
2      pass
3
4  pessoa1 = Pessoa()
5  pessoa1.Nome = 'Fernando'
6  pessoa1.Idade = 32
7
```

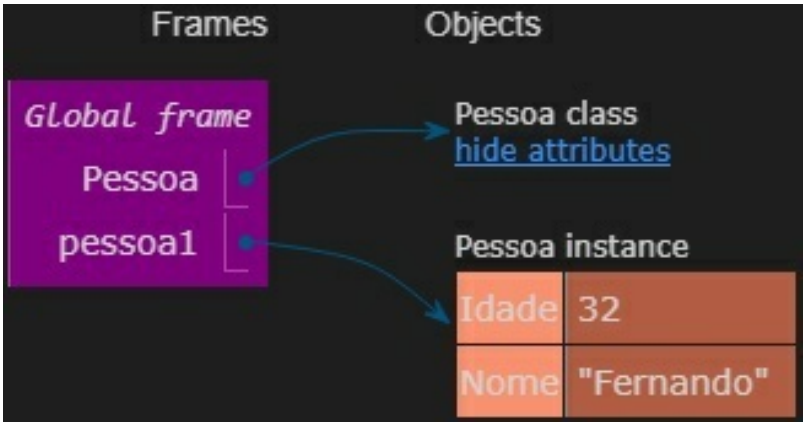
Inicialmente, pela sintaxe Python, para criarmos a estrutura de uma classe de forma que o interpretador faça a sua leitura léxica correta, precisamos escrever class seguido do nome da classe, com letra maiúscula apenas para diferenciar do resto.

Dentro da classe, aqui nesse exemplo, colocamos apenas o comando pass, dessa forma, esta será inicialmente uma classe “em branco”, por enquanto vazia.

Na sequência, fora da classe (repare pela indentação) criamos o objeto de nome pessoa1 que inicializa a classe Pessoa sem parâmetros mesmo, apenas a colocando como atributo próprio.

Em seguida, por meio do comando pessoa1.Nome, estamos criando

a variável Nome dentro da classe Pessoa, que por sua vez recebe ‘Fernando’ como atributo, o mesmo é feito adicionando a variável Idade com seu respectivo valor.



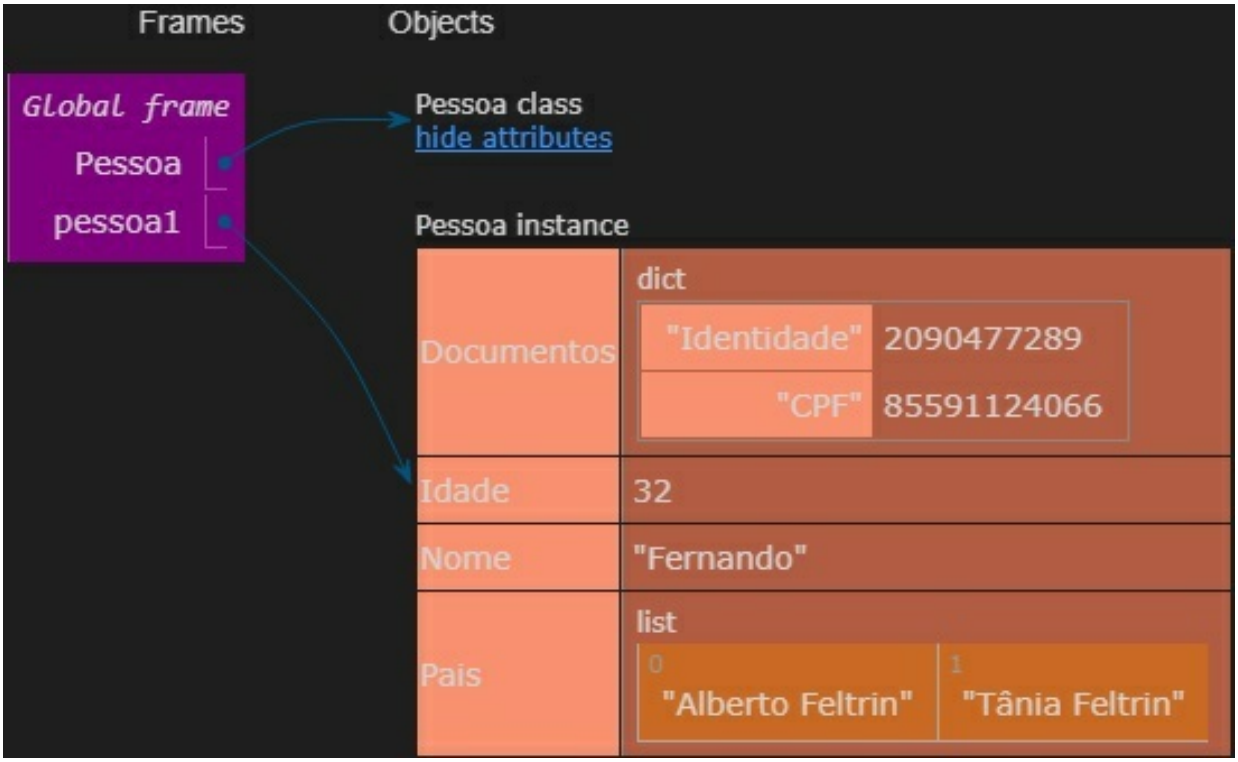
Note que em comparação ao código anterior temos mais linhas de código e de maior complexidade em sua lógica sintática, neste exemplo em particular ocorre isto mesmo, porém você verá posteriormente que programar blocos de código dentro de classes assim como as modularizar tornará as coisas muito mais eficientes, além de reduzir a quantidade de linhas/blocos de código.

Tratando os dados como instâncias de uma classe, temos liberdade para adicionar o que quisermos dentro da mesma, independente do tipo de dado e de sua função no contexto do código.

```
1  class Pessoa:
2      pass
3
4  pessoa1 = Pessoa()
5  pessoa1.Nome = 'Fernando'
6  pessoa1.Idade = 32
7  pessoa1.Documentos = {'Identidade':2090477289,
8                        'CPF':85591124066}
9  pessoa1.Pais = ['Alberto Feltrin', 'Tânia Feltrin']
10
```

Repare que seguindo com o exemplo, adicionalmente inserimos um

dicionário de nome Documentos com valores de Identidade e CPF, assim como uma lista de nome Pais que tem atribuída a si Alberto Feltrin e Tânia Feltrin.



Toda essa estrutura, guardada na classe Pessoa, se fosse ser feita de forma convencional estruturada demandaria a criação de cada variável em meio ao corpo do código geral, dependendo o contexto, poluindo bastante o mesmo e acarretando desempenho inferior em função da leitura léxica do interpretador da IDE.

Sendo assim, aqui inicialmente apenas mostrando de forma visual os primeiros conceitos, podemos começar a entender como se programa de forma orientada a objetos, paradigma este que você notará que será muito útil para determinados tipos de códigos / determinados tipos de problemas computacionais.

Nos capítulos subsequentes, gradualmente vamos começar a entender as funcionalidades que podem ser atribuídas a uma classe de um determinado objeto, assim como a forma como instanciaremos dados internos

desse objeto para interação com outras partes do código geral.

Outro ponto a levar em consideração neste momento inicial, pode estar parecendo um pouco confuso de entender a real distinção entre se trabalhar com uma variável (programação estruturada) e um objeto (programação orientada a objetos).

Para que isso fique bem claro, relembre do básico da programação em Python, uma variável é um espaço de memória alocado onde guardamos algum dado ou valor, de forma estática.

Já na programação orientada a objetos, uma variável ganha “super poderes” a partir do momento em que ela possui atribuída a si uma classe, dessa forma ela pode inserir e utilizar qualquer tipo de dado que esteja situado dentro da classe, de forma dinâmica.

Supondo que pela modularização tenhamos um arquivo de nome pessoa.py, dentro de si o seguinte bloco de código, referente a uma classe vazia.

```
1  class Pessoa:
2      pass
3
```

Agora, em nosso arquivo principal de nome main.py realizamos a importação dessa classe Pessoa e a atribuímos a diferentes variáveis/objetos.

```
 1  from pessoa import Pessoa
 2
 3  pessoa1 = Pessoa()
 4  pessoa2 = Pessoa()
 5  pessoa3 = Pessoa()
 6
 7  print(pessoa1)
 8  print(pessoa2)
 9  print(pessoa3)
10
```

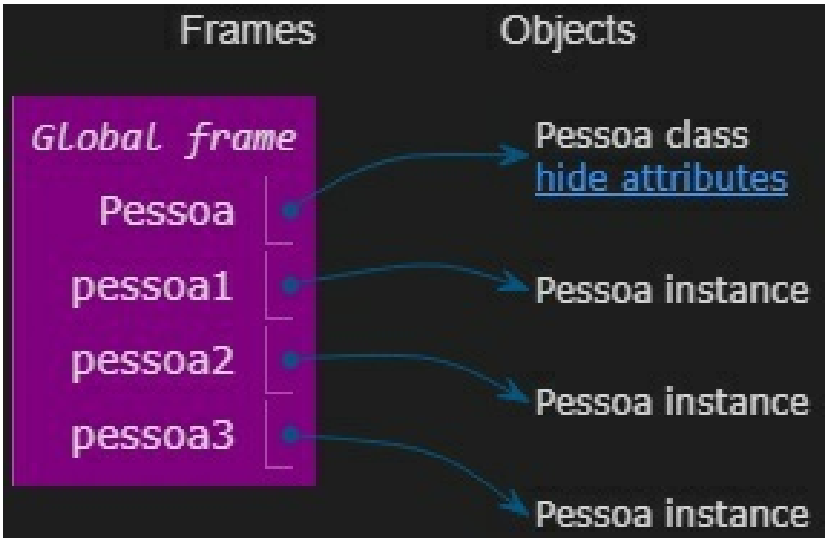
Diferentemente da programação estruturada, cada variável pode usar a classe Pessoa inserindo diferentes dados/valores/atributos a mesma.

Neste contexto, a classe Pessoa serve como “molde” para criação de

outros objetos, sendo assim, neste exemplo, cada variável que inicializa Pessoa( ) está criando toda uma estrutura de dados única para si, usando tudo o que existe dentro dessa classe (neste caso não há nada mesmo, porém poderia ter perfeitamente blocos e mais blocos de códigos...) sem modificar a estrutura de dados da classe Pessoa.

```
<pessoa.Pessoa object at 0x000001DAA659AA08>
<pessoa.Pessoa object at 0x000001DAA6565048>
<pessoa.Pessoa object at 0x000001DAA84E2208>
PS C:\Users\Fernando\PycharmProjects\Programação Orientada a Objetos>
```

Por meio da função print( ) podemos notar que cada variável pessoa está alocada em um espaço de memória diferente, podendo assim usar apenas uma estrutura de código (a classe Pessoa) para atribuir dados/valores a cada variável (ou para mais de uma variável).



Raciocine inicialmente que, a vantagem de se trabalhar com classes, enquanto as mesmas tem função inicial de servir de “molde” para criação de objetos, é a maneira como podemos criar estruturas de código reutilizáveis.

Para o exemplo anterior, imagine uma grande estrutura de código onde uma pessoa ou um item teria uma série de características (muitas mesmo), ao invés de criar cada característica manualmente para cada variável, é mais interessante se criar um molde com todas estas características

e aplicar para cada nova variável/objeto que necessite destes dados.

Em outras palavras, vamos supor que estamos a criar um pequeno banco de dados onde cada pessoa cadastrada tem como atributos um nome, uma idade, uma nacionalidade, um telefone, um endereço, um nome de pai, um nome de mãe, uma estimativa de renda, etc...

Cada dado/valor destes teria de ser criado manualmente por meio de variáveis, e ser recriado para cada nova pessoa cadastrada neste banco de dados... além de tornar o corpo do código principal imenso.

É logicamente muito mais eficiente criar uma estrutura de código com todos estes parâmetros que simplesmente possa ser reutilizado a cada novo cadastro, ilimitadamente, simplesmente atribuindo o mesmo a uma variável/objeto, e isto pode ser feito perfeitamente por meio do uso de classes (estrutura de dados de classe / estrutura de dados orientada a objeto).

Entendidas as diferenças básicas internas entre programação estruturada convencional e programação orientada a objetos, hora de começarmos a de fato criar nossas primeiras classes e suas respectivas usabilidades, uma vez que como dito anteriormente, este tipo de estrutura de dados irá incorporar em si uma série de dados/valores/variáveis/objetos/funções/etc... de forma a serem usados, reutilizados, instanciados e ter interações com diferentes estruturas de blocos de código.

Criando uma classe vazia

Como dito anteriormente, uma classe é uma estrutura de código que permite associar qualquer tipo de atribuição a uma variável/objeto. Também é preciso que se entenda que tudo o que será codificado dentro de uma classe funcionará como um molde para que se criem novos objetos ou a interação entre os mesmos.

Dessa forma, começando a entender o básico das funcionalidades de uma função, primeira coisa a se entender é que uma classe tem uma sintaxe própria que a define, internamente ela pode ser vazia, conter conteúdo próprio previamente criado ou inserir conteúdo por meio da manipulação da variável/objeto ao qual foi atribuída.

Além disso, uma classe pode ou não retornar algum dado ou valor de acordo com seu uso em meio ao código.

```
1  class Pessoa:
2      pass
3
```

Começando do início, a palavra reservada ao sistema class indica ao interpretador que logicamente estamos trabalhando com uma classe.

A nomenclatura usual de uma classe deve ser o nome da mesma iniciando com letra maiúscula.

Por fim, especificado que se trata de uma classe, atribuído um nome a mesma, é necessário colocar dois pontos para que seja feita sua indentação, muito parecida com uma função comum em Python.

```
1  class Pessoa:
2      pass
3
4  pessoa1 = Pessoa()
5  pessoa1.nome = 'Fernando'
6
7  print(pessoa1.nome)
8
9
```

```
Fernando
```

Dando sequência, após criada a classe Pessoa, podemos começar a manipular a mesma.

Neste exemplo, em seguida é criada no corpo do código geral a variável pessoa1 que inicializa a classe Pessoa( ), atribuindo a si todo e qualquer conteúdo que estiver inserido nessa classe.

É necessário colocar o nome da classe seguido de parênteses pois posteriormente você verá que é possível passar parâmetros neste espaço.

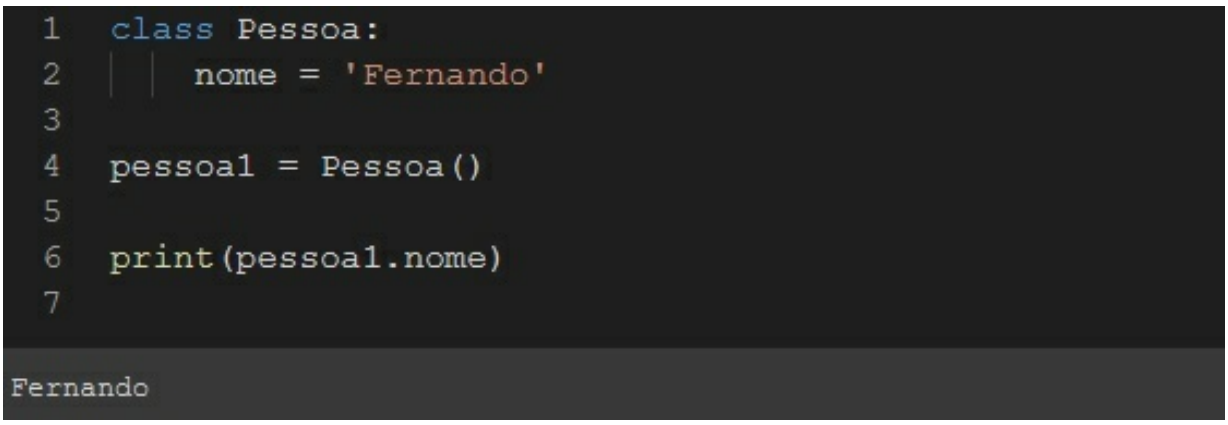
Na sequência é dado o comando pessoa1.nome = ‘Fernando’, o que pode ser entendido que, será criada dentro da classe uma variável de nome nome que recebe como dado a string ‘Fernando’. Executando a função print( ) e passando como parâmetro pessoa1.nome, o retorno será Fernando.



Atributos de classe

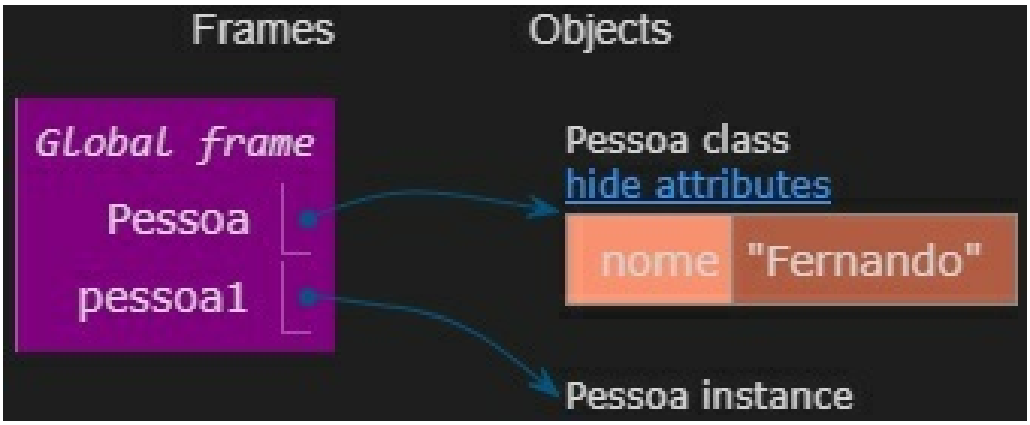
Dando mais um passo no entendimento lógico de uma classe, hora de começarmos a falar sobre os atributos de classe. No exemplo anterior criamos uma classe vazia e criamos de fora da classe uma variável de nome nome que recebia ‘Fernando’ como atributo.

Justamente, um atributo de classe nada mais é do que uma variável criada/declarada dentro de uma classe, onde de acordo com a sintaxe Python, pode receber qualquer dado ou valor.



Dessa vez criamos uma classe de nome Pessoa, internamente criamos uma variável de nome nome que recebe como atributo ‘Fernando’. Em seguida, fora da classe, criamos uma variável de nome pessoa1 que recebe como atributo a classe Pessoa, a partir desse momento, todo conteúdo de Pessoa estará atribuído a variável pessoa1.

Dessa forma, por meio da função print( ) passando como parâmetro pessoa1.nome, o resultado será: Fernando, já que a variável interna a classe Pessoa agora pertence a pessoa1.



O que fizemos, iniciando nossos estudos, foi começar a entender algumas possibilidades, como: A partir do momento que uma variável no corpo geral do código recebe como atributo uma classe, ela passa a ser um objeto, uma super variável, uma vez que a partir deste momento ela consegue ter atribuída a si uma enorme gama de possibilidades, desde armazenar variáveis dentro de variáveis até ter inúmeros tipos de dados dentro de si.

Também vimos que é perfeitamente possível ao criar uma classe, criar variáveis dentro dela que podem ser instanciadas de fora da classe, por uma variável/objeto qualquer.

Ainda neste contexto, é importante que esteja bem entendido que, a classe de nosso exemplo Pessoa, dentro de si tem uma variável nome com um atributo próprio, uma vez que esta classe seja instanciada por outra variável, que seja usada como molde para outra variável, toda sua estrutura interna seria transferida também para o novo objeto.

Manipulando atributos de uma classe

Da mesma forma que na programação convencional estruturada, existe a chamada leitura léxica do interpretador, e ela se dá garantindo que o interpretador fará a leitura dos dados em uma certa ordem e sequência (sempre da esquerda para a direita e linha após linha descendente).

Seguindo essa lógica, um bloco de código declarado posteriormente pode alterar ou "atualizar" um dado ou valor em função de que na ordem de interpretação a última informação é a que vale.

Aqui não há necessidade de nos estendermos muito, mas apenas raciocine que é perfeitamente normal e possível manipular dados de dentro de uma classe de fora da mesma.

Porém é importante salientar que quando estamos trabalhando com uma classe, ocorre a atualização a nível da variável que está instanciando a classe, a estrutura interna da classe continua intacta. Por exemplo:

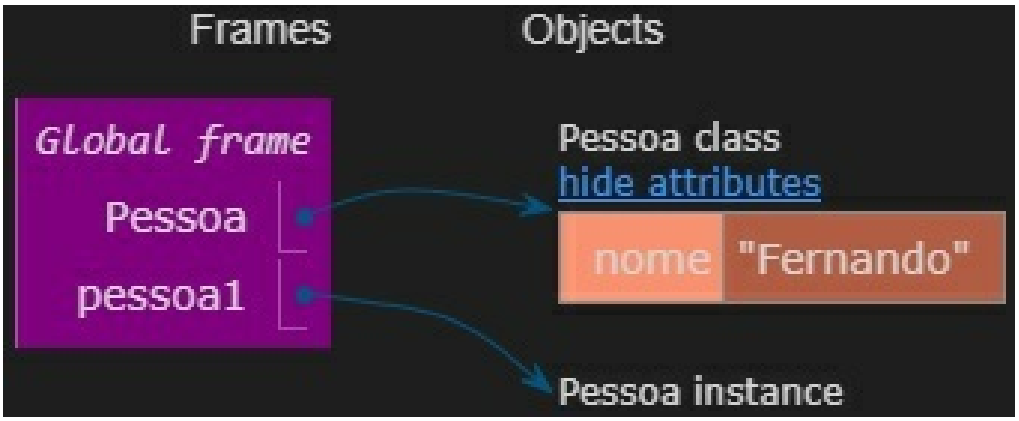
```
1  class Pessoa:
2      nome = 'Padrão'
3
4  pessoa1 = Pessoa()
5
6  print(pessoa1.nome)
7
```

```
Padrão
```

Aproveitando o exemplo anterior, note que está declarada dentro da classe Pessoa uma variável de nome nome que tem 'Padrão' como atributo.

Logo após criamos uma variável de nome pessoa1 que inicializa a classe e toma posse de seu conteúdo.

Neste momento, por meio da função print( ) passando como parâmetro pessoa1.nome o retorno é: Padrão

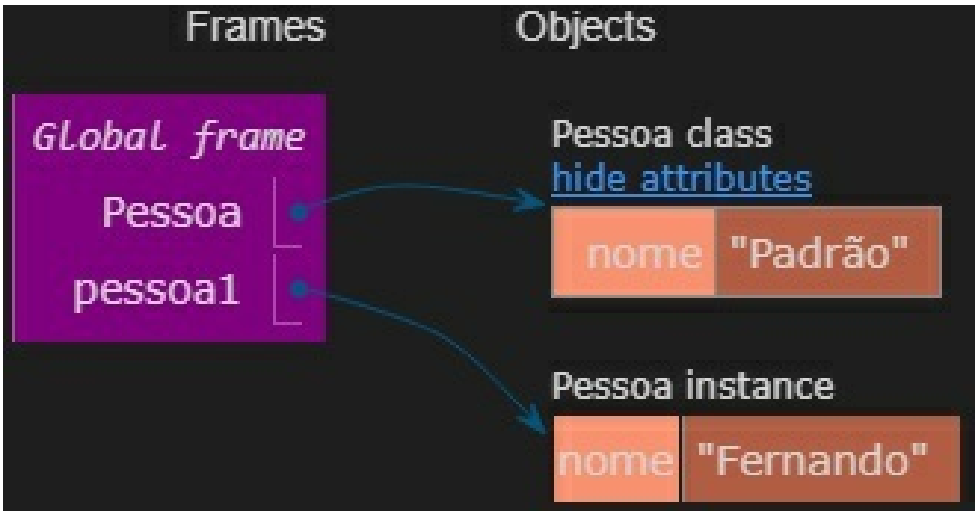



```
1  class Pessoa:
2      nome = 'Padrão'
3
4  pessoa1 = Pessoa()
5  pessoa1.nome = 'Fernando'
6
7  print(pessoa1.nome)
8
```

```
Fernando
```

Agora após criar a variável pessoa1 e atribuir a mesma a classe Pessoa, no momento que realizamos a alteração pessoa1.nome = ‘Fernando’, esta alteração se dará somente para a variável pessoa1.

Por meio da função print( ) é possível ver que pessoa1.nome agora é Fernando.



Note que a estrutura da classe, “o molde” da mesma se manteve, para a classe Pessoa, nome ainda é ‘Padrão’, para a variável pessoa1, que instanciou Pessoa e sobrescreveu nome, nome é ‘Fernando’.

Como dito anteriormente, uma classe pode servir de molde para criação de vários objetos, sem perder sua estrutura a não ser que as alterações sejam feitas dentro dela, e não pelas variáveis/objetos que a instanciam.

Importante salientar que é perfeitamente possível fazer a manipulação de dados dentro da classe operando diretamente sobre a mesma, porém, muito cuidado pois dessa forma, como a classe serve como “molde”, uma alteração diretamente na classe afeta todas suas instâncias.

Em outras palavras, diferentes variáveis/objetos podem instanciar uma classe e modificar o que quiser a partir disto, a estrutura da classe permanecerá intacta. Porém quando é feita uma alteração diretamente na classe, essa alteração é aplicada para todas variáveis/objetos que a instancia.

```
1  class Pessoa:
2      nome = 'Padrão'
3
4  pessoa1 = Pessoa()
5  pessoa2 = Pessoa()
6  pessoa3 = Pessoa()
7
8  Pessoa.nome = 'Fernando'
9
10 print(pessoa1.nome)
11 print(pessoa2.nome)
12 print(pessoa3.nome)
13
```

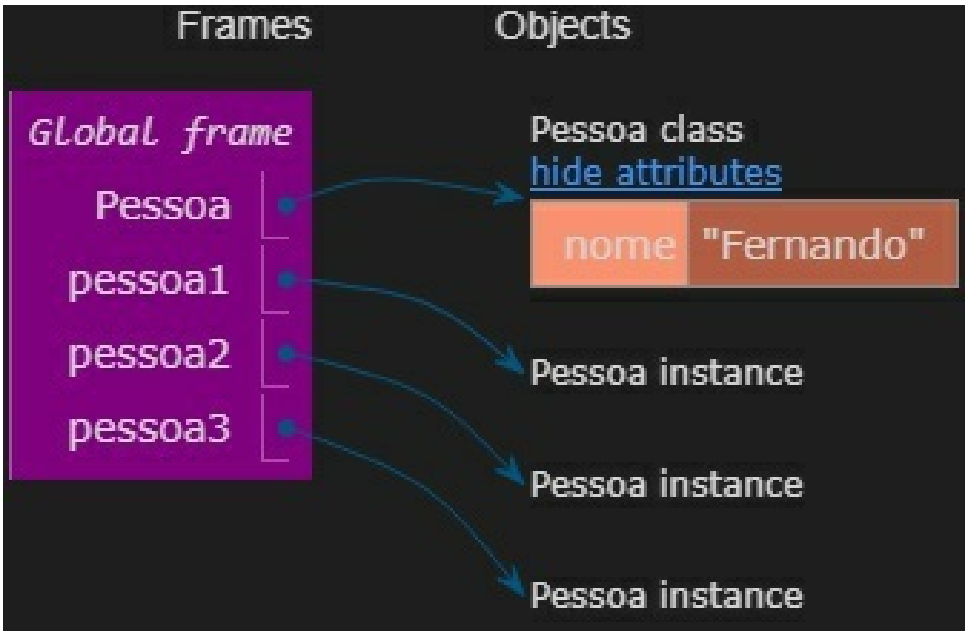
```
Fernando
Fernando
Fernando
```

Aqui seguindo com o mesmo exemplo, porém criando 3 variáveis pessoa que recebem como atributo a classe Pessoa, a partir do momento que é lida a linha de código Pessoa.nome = ‘Fernando’ pelo interpretador, a alteração da variável nome de ‘Padrão’ para ‘Fernando’ é aplicada para todas as instâncias. Sendo assim, neste caso o retorno será:

Fernando

Fernando

Fernando



Em resumo, alterar um atributo de classe via instancia altera o valor somente para a instância em questão (somente para a variável/objeto que instanciou), alterar via classe altera diretamente para todas as instâncias que a usar a partir daquele momento.

Métodos de classe

Uma prática bastante comum quando se trabalha com orientação a objetos é criar funções dentro de uma classe, essas funções por sua vez recebem a nomenclatura de métodos de classe.

Raciocine que nada disto seria possível na programação estruturada convencional, aqui, como dito anteriormente, você pode criar absolutamente qualquer coisa dentro de uma classe, ilimitadamente.

Apenas como exemplo, vamos criar uma simples função (um método de classe) dentro de uma classe, que pode ser executada por meio de uma variável/objeto.

```
1  class Pessoa:
2      def acao1(self):
3          print('Ação 1 sendo executada...')
4
```

Todo processo se inicia com a criação da classe Pessoa, dentro dela, simplesmente criamos uma função (método de classe) de nome acao1 que tem como parâmetro self.

Vamos entender esse contexto, a criação da função, no que diz respeito a sintaxe, é exatamente igual ao que você está acostumado, porém este parâmetro self é característico de algo sendo executado dentro da classe.

Uma classe implicitamente dá o comando ao interpretador pra que seus blocos de código sejam executados apenas dentro de si, apenas quando instanciados, por uma questão de performance.

Posteriormente haverão situações diferentes, mas por hora vamos nos ater a isto. Note que a função acao1 por sua vez simplesmente exibe uma string com uma mensagem pré-programada por meio da função print( ).

```python
class Pessoa:
    def acao1(self):
        print('Ação 1 sendo executada...')

pessoa1 = Pessoa()

print(pessoa1.acao1())

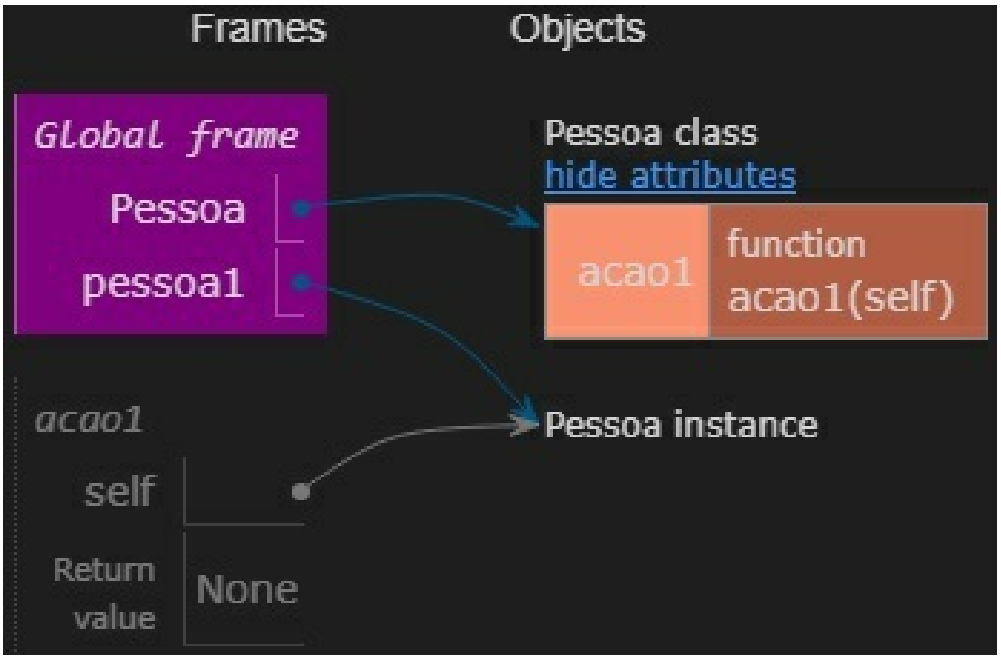
```

```
Ação 1 sendo executada...
None
```

Da mesma forma do exemplo anterior, é criada a variável pessoa1 que recebe como atributo a classe Pessoa( ), em seguida, por meio da função print( ) passamos como parâmetro a função acao1, sem parâmetros mesmo, por meio de pessoa1.acao1( ).

Neste caso o retorno será: Ação 1 sendo executada...

* dependendo qual IDE você estiver usando, pode ser que apareça em seu retorno a expressão None, que neste caso apenas significa que para aquele bloco de código não existe nenhuma ação sendo executada.



Método construtor de uma classe

Quando estamos trabalhando com classes simples, como as vistas anteriormente, não há uma real necessidade de se criar um construtor para as mesmas.

Na verdade, internamente esta estrutura é automaticamente gerada, o chamado construtor é criado manualmente quando necessitamos parametrizá-lo manualmente de alguma forma, criando estruturas que podem ser instanciadas e receber atribuições de fora da classe.

Para alguns autores, esse processo é chamado de método construtor, para outros de método inicializador, independente do nome, este é um método (uma função interna) que recebe parâmetros (atributos de classe) que se tornarão variáveis internas desta função, guardando dados/valores a serem passados pelo usuário por meio da variável que instanciar essa classe.

Em outras palavras, nesse exemplo, do corpo do código, uma variável/objeto que instanciar a classe Pessoa terá de passar os dados referentes a nome, idade, sexo e altura, já que o método construtor dessa classe está esperando que estes dados sejam fornecidos.

```
1  class Pessoa:
2      def __init__(self, nome, idade, sexo, altura):
3          self.nome = nome
4          self.idade = idade
5          self.sexo = sexo
6          self.altura = altura
7
```

Ainda trabalhando sobre o exemplo da classe Pessoa, note que agora criamos uma função interna de nome __init__(self), palavra reservada ao sistema para uma função que inicializa dentro da classe algum tipo de função.

Em sua IDE, você notará que ao digitar __init__ automaticamente ele criará o parâmetro (self), isto se dá porque tudo o que estiver indentado a essa função será executado internamente e associado a variável que é a

instância desta classe (a variável do corpo geral do código que tem esta classe como atributo).

Da mesma forma que uma função convencional, os nomes inseridos como parâmetro serão variáveis temporárias que receberão algum dado ou valor fornecido pelo usuário, normalmente chamados de atributos de classe.

Repare que neste exemplo estão sendo criadas variáveis (atributos de classe) para armazenar o nome, a idade, o sexo e a altura. Internamente, para que isto se torne variáveis instanciáveis, é criada uma estrutura self.nomedavariavel que recebe como atributo o dado ou valor inserido pelo usuário, respeitando inclusive, a ordem em que foram parametrizados.

```
1  class Pessoa:
2    def __init__(self, nome, idade, sexo, altura):
3      self.nome = nome
4      self.idade = idade
5      self.sexo = sexo
6      self.altura = altura
7
8  pessoa1 = Pessoa('Fernando', 32, 'M', 1.90)
9
```

Dessa forma, é criada a variável pessoa1 que recebe como atributo a classe Pessoa, que por sua vez tem os parâmetros ‘Fernando’, 32, ‘M’, 1.90. Estes parâmetros serão substituídos internamente, ordenadamente, pelos campos referentes aos mesmos.

O self nunca receberá atribuição ou será substituído, ele é apenas a referência de que estes dados serão guardados dentro e para a classe, mas para nome será atribuído ‘Fernando’, para idade será atribuído 32, para o sexo será atribuído M e por fim para a altura será atribuído o valor de 1.90.

```
1  class Pessoa:
2      def __init__(self, nome, idade, sexo, altura):
3          self.nome = nome
4          self.idade = idade
5          self.sexo = sexo
6          self.altura = altura
7
8  pessoa1 = Pessoa('Fernando', 32, 'M', 1.90)
9
10 print(pessoa1.nome, pessoa1.idade)
11
```

```
Fernando 32
```

A partir deste momento, podemos normalmente instanciarmos qualquer dado ou valor que está guardado na classe Pessoa.

Por meio da função print( ) passando como parâmetros pessoa1.nome e pessoa1.idade, o retorno será: Fernando, 32.

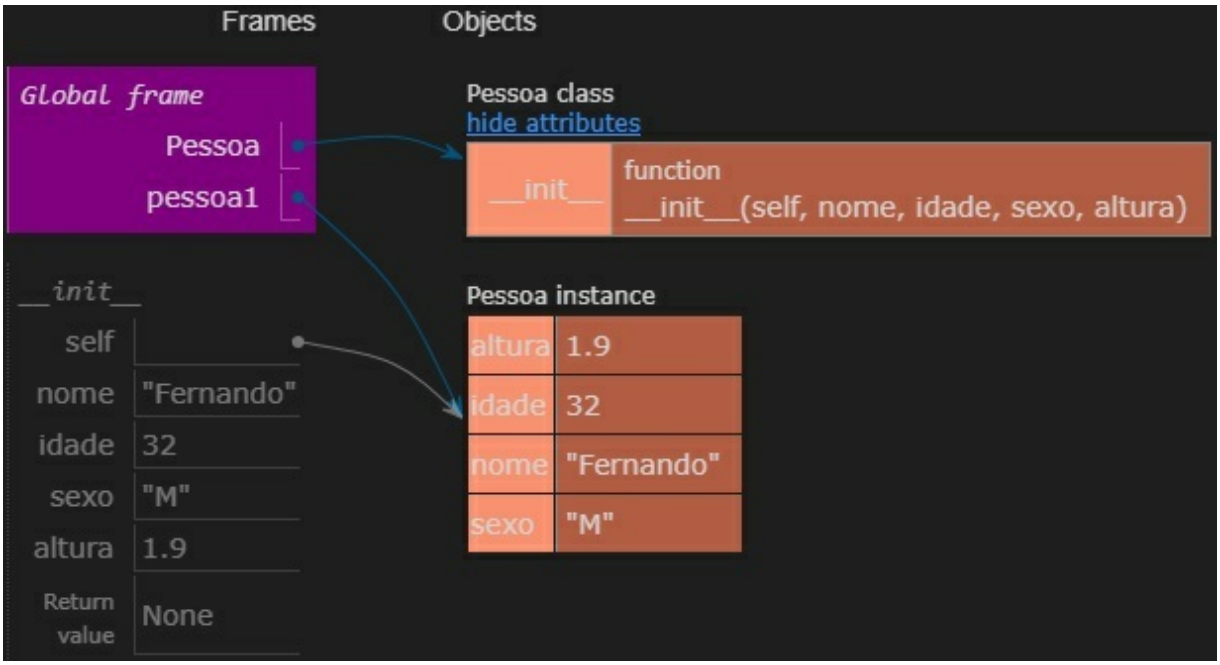
```
10 print(pessoa1.nome, pessoa1.idade)
11
12 print(f'Bem vindo {pessoa1.nome}, parabéns pelos seus {pessoa1.idade} anos!!!')
13
```

```
Fernando 32
Bem vindo Fernando, parabéns pelos seus 32 anos!!!
```

Apenas a nível de curiosidade, a partir do momento que temos variáveis internas a uma classe, com seus devidos atributos, podemos usá-los normalmente de fora da classe, no exemplo acima, criando uma mensagem usando de f strings e máscaras de substituição normalmente.

Neste caso o retorno será: Bem vindo Fernando, parabéns pelos seus 32 anos!!!

A questão da declaração de parâmetros, que internamente farão a composição das variáveis homônimas, devem respeitar a estrutura self.nomedavariavel, porém dentro do mesmo escopo, da mesma indentação, é perfeitamente normal criar variáveis independentes para guardar algum dado ou valor dentro de si.


Frames
Objects
Global frame
Pessoa
pessoa1
Pessoa class
hide attributes
__init__
function
__init__(self, nome, idade, sexo, altura)
__init__
self
nome
"Fernando"
idade
32
sexo
"M"
altura
1.9
Return value
None
Pessoa instance
altura
1.9
idade
32
nome
"Fernando"
sexo
"M"

Escopo / Indentação de uma classe

Já que uma classe é uma estrutura que pode guardar qualquer tipo de bloco de código dentro de si, é muito importante respeitarmos a indentação correta desses blocos de código, assim como entender que, exatamente como funciona na programação convencional estruturada, existem escopos os quais tornam itens existentes dentro do mesmo bloco de código que podem ou não estar acessíveis para uso em funções, seja dentro ou fora da classe. Ex:

```
1  class Pessoa:
2      administrador = 'Admin'
3
4      def __init__(self, nome):
5          self.nome = nome
6
7          msg = 'Classe Pessoa em execução.'
8          print(msg)
9
10     def metodo1(self):
11         print(msg)
12         pass
13
14 var1 = Pessoa('Fernando')
15
```

```
Classe Pessoa em execução.
```

Neste momento foque apenas no entendimento do escopo, o bloco de código acima tem estruturas que estaremos entendendo a lógica nos capítulos seguintes.

Inicialmente é criada uma classe de nome Pessoa, dentro dela existe a variável de nome administrador com sua respectiva atribuição ‘Admin’, esta variável de acordo com sua localização indentada estará disponível para uso de qualquer função dentro ou fora dessa classe.

Em seguida temos o método construtor que simplesmente recebe um nome atribuído pelo usuário e exibe uma mensagem padrão conforme declarado pela variável msg.

Na sequência é criada uma função chamada método1 que dentro de si apenas possui o comando para exibir o conteúdo de msg, porém, note que msg é uma variável dentro do escopo do método construtor, e em função disso ela só pode ser acessada dentro deste bloco de código.

Por fim, fora da classe é criada uma variável de nome var1 que instancia a classe Pessoa passando ‘Fernando’ como o parâmetro a ser substituído em self.nome.

```
1  class Pessoa:
2      administrador = 'Admin'
3
4      def __init__(self, nome):
5          self.nome = nome
6
7          msg = 'Classe Pessoa em execução.'
8          print(msg)
9
10     def metodo1(self):
11         print(msg)
12         pass
13
14 var1 = Pessoa('Fernando')
15 print(var1)
16
```

```
Classe Pessoa em execução.
<__main__.Pessoa object at 0x7f83c2a6d940>
```

Por meio da função print( ) passando como parâmetro var1 , o retorno será a exibição de msg, uma vez que é a única linha de código que irá retornar algo ao usuário.

Sendo assim o retorno será: Classe Pessoa em execução.

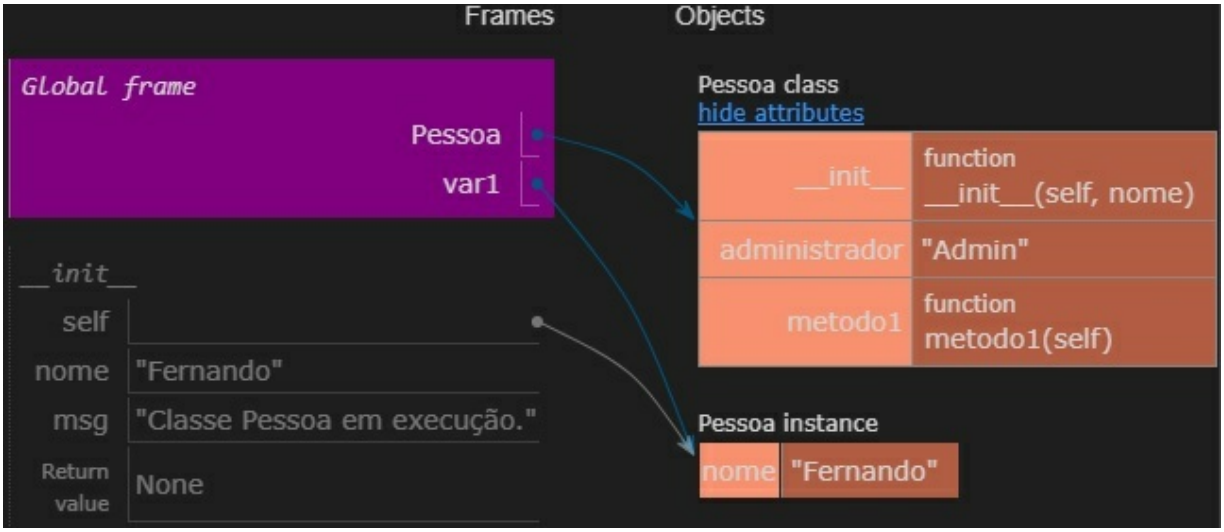
```
14   var1 = Pessoa('Fernando')
15   print(var1)
16   print(var1.metodo1())
17
```

```
Classe Pessoa em execução.
<__main__.Pessoa object at 0x7f83c2a6ddd8>
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call
<ipython-input-21-9f9ab4ca6d31> in <module>()
     14 var1 = Pessoa('Fernando')
     15 print(var1)
---> 16 print(var1.metodo1())

<ipython-input-21-9f9ab4ca6d31> in metodo1(self)
      9
     10     def metodo1(self):
---> 11         print(msg)
     12         pass
     13

NameError: name 'msg' is not defined
```

Já ao tentarmos executar msg dentro da função metodo1, o retorno será NameError: name 'msg' is not defined, porque de fato, msg é uma variável interna de __init__, inacessível para outras funções da mesma classe.



Apenas concluindo o raciocínio, a variável msg, “solta” dentro do bloco de código construtor, somente é acessível e instanciável neste mesmo bloco de código.

Porém, caso quiséssemos tornar a mesma acessível para outras funções dentro da mesma classe ou até mesmo fora dela, bastaria reformular o código e indexa-la ao método de classe por meio da sintaxe self.msg, por exemplo.

```
 1  class Pessoa:
 2      administrador = 'Admin'
 3
 4      def __init__(self, nome, msg):
 5          self.nome = nome
 6          self.msg = msg
 7          print(msg)
 8
 9      def metodo1(self):
10          print(msg)
11          pass
12
13  var1 = Pessoa('Fernando', 'Classe Pessoa em execução')
14
```

```
Classe Pessoa em execução
```

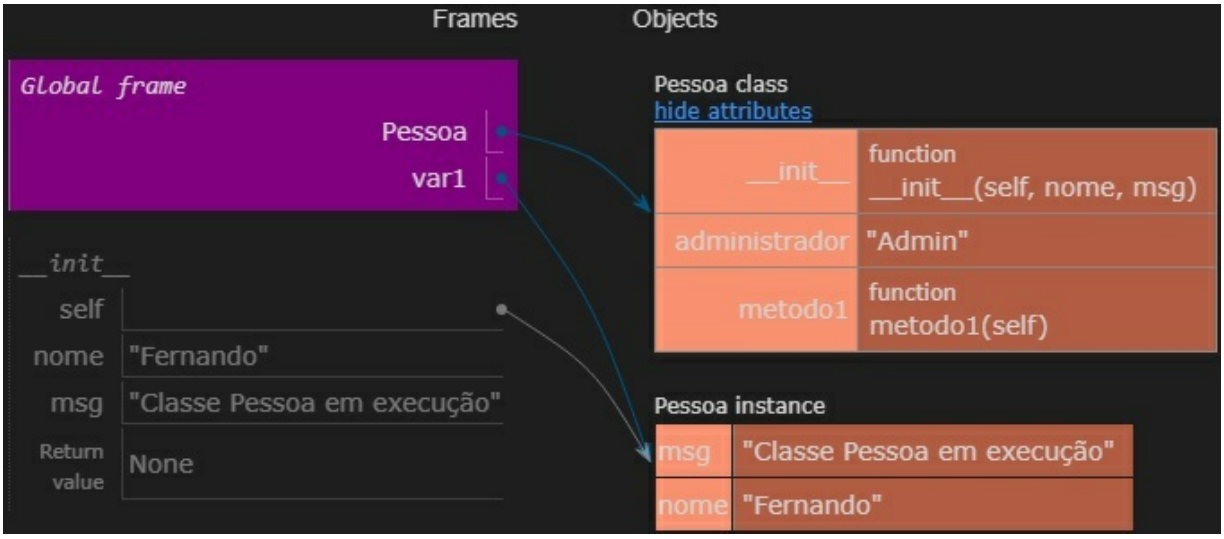
Note que agora msg é um dos parâmetros a ser repassados pelo usuário no momento da criação da variável que instancia a classe Pessoa.

```
13  var1 = Pessoa('Fernando', 'Classe Pessoa em execução')
14  print(var1.metodo1)
15
```

```
Classe Pessoa em execução
<bound method Pessoa.metodo1 of <__main__.Pessoa object at 0x7f83c2235cc0>>
```

Dessa forma, por meio da função print( ) fora da classe conseguimos perfeitamente instanciar tal variável uma vez que agora internamente a classe existe a comunicação do método construtor e da função metodo1.

De imediato isto pode parecer bastante confuso, porém por hora o mais importante é começar a entender as possibilidades que temos quando se trabalha fazendo o uso de classes, posteriormente estaremos praticando mais tais conceitos e dessa forma entendendo definitivamente sua lógica.



Apenas finalizando o entendimento de escopo, caso ainda não esteja bem entendido, raciocine da seguinte forma:

```
1  class Pessoa:
2      pessoa1 = 'Admin'
3      #pessoa1 faz parte do escopo global da classe
4      #pessoa1 é instanciável por qualquer método
5
6      def __init__(self, pessoa2):
7          self.pessoa2 = pessoa2
8          #pessoa2 faz parte do escopo do método construtor
9          #pessoa2 é acessível dentro e fora deste método
10         #pessoa2 pode ser instanciada de fora da classe
11
12         pessoa3 = 'DefaultUser'
13         #pessoa3 faz parte do escopo do método construtor
14         #pessoa3 é acessível somente dentro deste método
15
```

Por fim somente fazendo um adendo, é muito importante você evitar o uso de variáveis/objetos de classe de mesmo nome, ou ao menos ter bem claro qual é qual de acordo com seu escopo.

```
class Pessoa:
    nome = 'Padrão'

    def __init__(self):
        self.nome = 'Administrador'

usuario1 = Pessoa()

print(usuario1.nome)
print(Pessoa.nome)

```

```
Administrador
Padrão
```

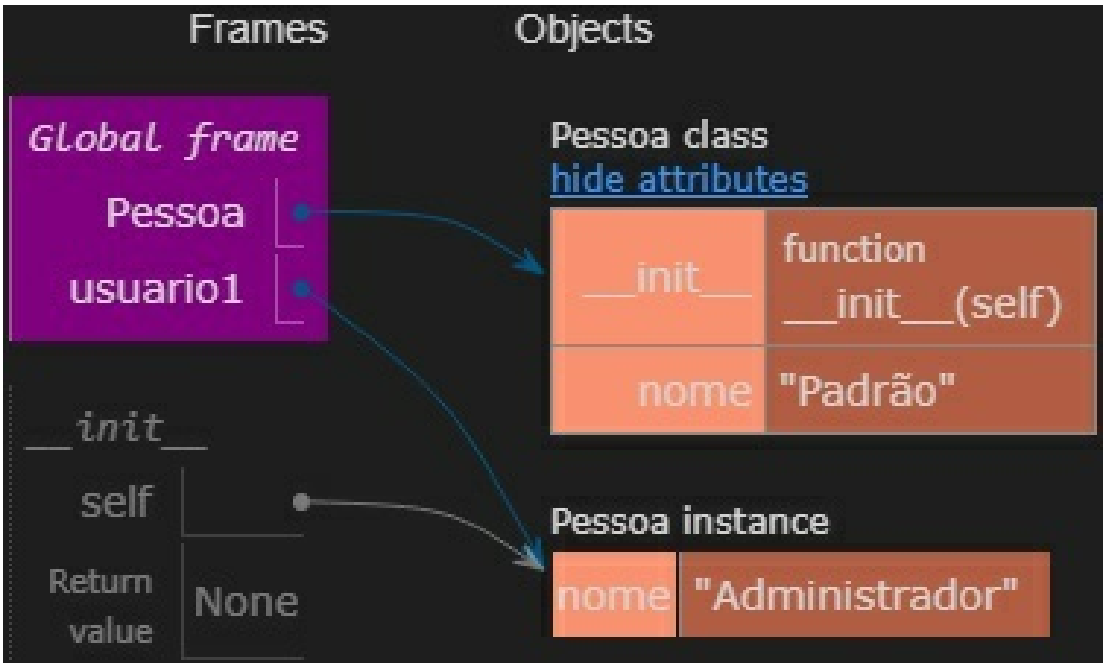
Apenas simulando uma situação dessas, note que dentro da classe Pessoa existe o objeto de classe nome, de atributo ‘Padrão’, em seguida existe um método construtor que dentro de si também possui um objeto de classe nome, agora de atributo ‘Administrador’.

Por mais estrando que pareça sempre que há um método construtor dentro de uma classe, esse é lido e interpretado por primeiro, ignorando até mesmo linhas de código anteriores a ele.

Pela leitura léxica do interpretador, é feita a leitura linha após linha, porém a prioridade é processar o método construtor dentro da estrutura da classe.

Na sequência é criada uma variável que instancia a classe Pessoa. Por meio da nossa função print( ), passando como parâmetro usuario1.nome o retorno será Administrador (instância do método construtor), passando como parâmetro Pessoa.nome, o retorno será Padrão (objeto de classe).

Então, para evitar confusão é interessante simplesmente evitar criar objetos de classe de mesmo nome, ou ao menos não confundir o uso dos mesmos na hora de incorporá-los ao restante do código.


Frames
Objects
Global frame
Pessoa
usuario1
Pessoa class
hide attributes
__init__
function
__init__(self)
nome
"Padrão"
__init__
self
Return
value
None
Pessoa instance
nome
"Administrador"

Classe com parâmetros opcionais

Outra situação bastante comum é a de definirmos parâmetros (atributos de classe) que serão obrigatoriamente substituídos e em contraponto parâmetros que opcionalmente serão substituídos.

Como dito anteriormente, uma classe pode servir de “molde” para criar diferentes objetos a partir dela, e não necessariamente todos esses objetos terão as mesmas características.

Sendo assim, podemos criar parâmetros que opcionalmente farão parte de um objeto enquanto não farão parte de outro por meio de não atribuição de dados ou valores para esse parâmetro.

Basicamente isto é feito simplesmente declarando que os parâmetros que serão opcionais possuem valor inicial definido como False. Ex:

```
1  class Pessoa:
2      def __init__(self, nome, idade, sexo=False, altura=False):
3          self.nome = nome
4          self.idade = idade
5          self.sexo = sexo
6          self.altura = altura
7
```

Repare que estamos na mesma estrutura de código do exemplo anterior, porém dessa vez, declaramos que sexo=False assim como altura=False, o que faz com que esses parâmetros sejam inicializados como None.

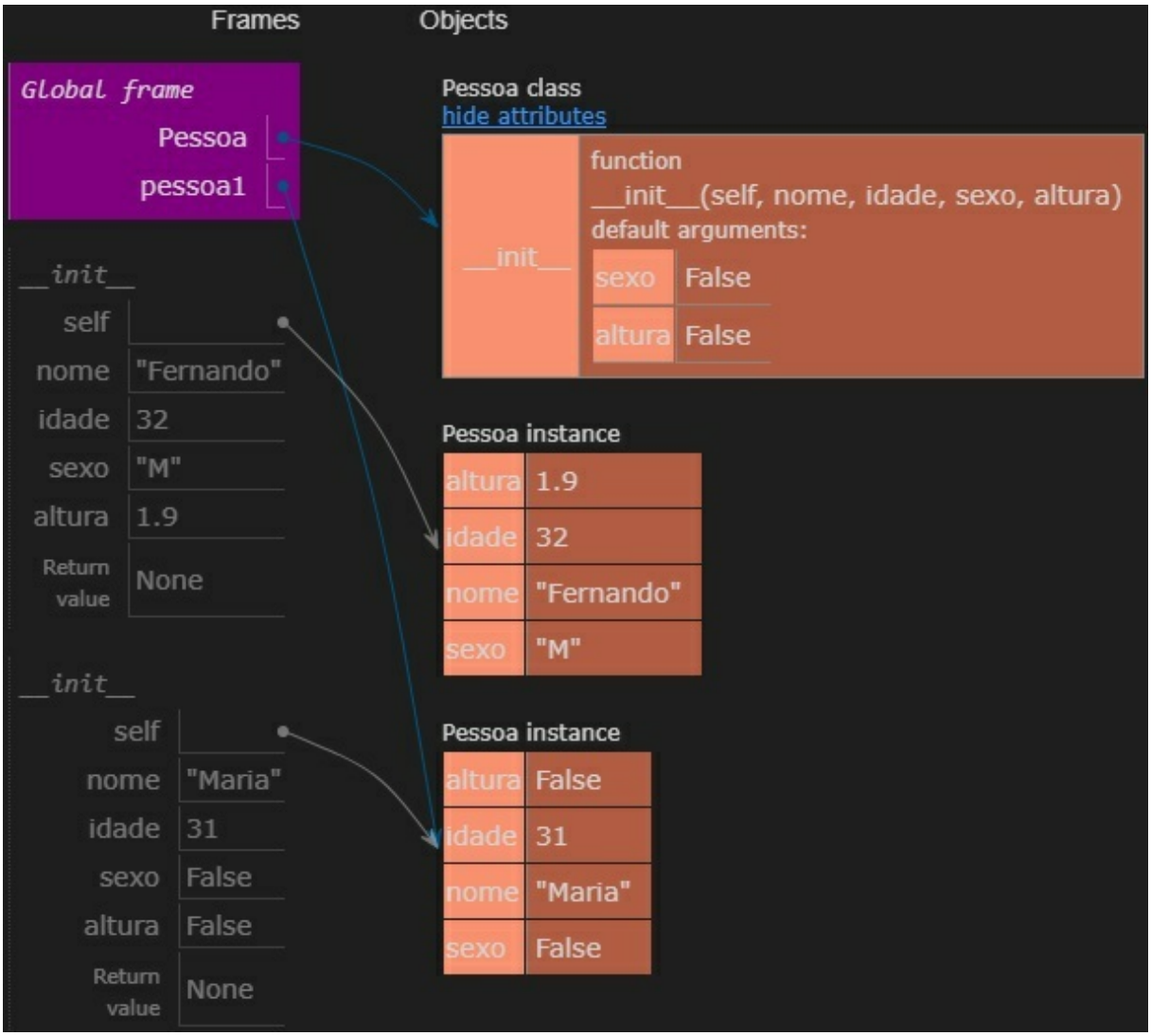
Se o usuário quiser atribuir dados ou valores a estes dados, simplesmente False será substituído pelo referente dado/valor, uma vez que False é uma palavra reservada ao sistema indicando que neste estado o interpretador ignore essa variável.

```
1  class Pessoa:
2    def __init__(self, nome, idade, sexo=False, altura=False):
3      self.nome = nome
4      self.idade = idade
5      self.sexo = sexo
6      self.altura = altura
7
8  pessoa1 = Pessoa('Fernando', 32, 'M', 1.90)
9  pessoa2 = Pessoa('Maria', 31)
10
11 print(pessoa1.nome, pessoa1.altura)
12 print(pessoa2.nome, pessoa2.idade)
13
```

```
Fernando 1.9
Maria 31
```

Dessa forma, continuamos trabalhando normalmente atribuindo ou não dados/valores sem que haja erro de interpretador.

Neste caso o retorno será: Fernando, 1.90. Maria, 31.


Frames
Objects
Global frame
Pessoa
pessoa1
__init__
self
nome "Fernando"
idade 32
sexo "M"
altura 1.9
Return value None
__init__
self
nome "Maria"
idade 31
sexo False
altura False
Return value None
Pessoa class
hide attributes
__init__
function
__init__(self, nome, idade, sexo, altura)
default arguments:
sexo False
altura False
Pessoa instance
altura 1.9
idade 32
nome "Fernando"
sexo "M"
Pessoa instance
altura False
idade 31
nome "Maria"
sexo False

Múltiplos métodos de classe

Uma prática comum é termos dentro de uma classe diversas variáveis assim como mais de uma função (métodos de classe), tudo isso atribuído a um objeto que é “dono” dessa classe.

Dessa forma, podemos criar também interações entre variáveis com variáveis e entre variáveis com funções, simplesmente respeitando a indentação dos mesmos dentro da hierarquia estrutural da classe. Por exemplo:

```
1  class Pessoa:
2      ano_atual = 2019
3
4      def __init__(self, nome, idade):
5          self.nome = nome
6          self.idade = idade
7
```

Inicialmente criamos a classe Pessoa, dentro dela, criamos uma variável de nome ano_atual que recebe como atributo o valor fixo 2019, ano_atual está no escopo geral da classe, sendo acessível a qualquer função interna da mesma.

Em seguida, é criado um construtor onde simplesmente é definido que haverão parâmetros para nome e idade fornecidos pelo usuário, estes, serão atribuídos a variáveis homônimas internas desta função.

```
1  def __init__(self, nome, idade):
2      self.nome = nome
3      self.idade = idade
4
5  def ano_nascimento(self):
6      ano_nasc = self.ano_atual - self.idade
7      print(f'Seu ano de nascimento é {ano_nasc}')
8
```

Em seguida é criada a função ano_nascimento, dentro de si é criada

a variável ano_nasc que por sua vez faz uma operação instanciando e subtraindo os valores de ano_atual e idade.

Note que esta operação está instanciando a variável ano_atual que não faz parte do escopo nem do construtor nem desta função, isto é possível porque ano_atual é integrante do escopo geral da classe e está disponível para qualquer operação dentro dela.

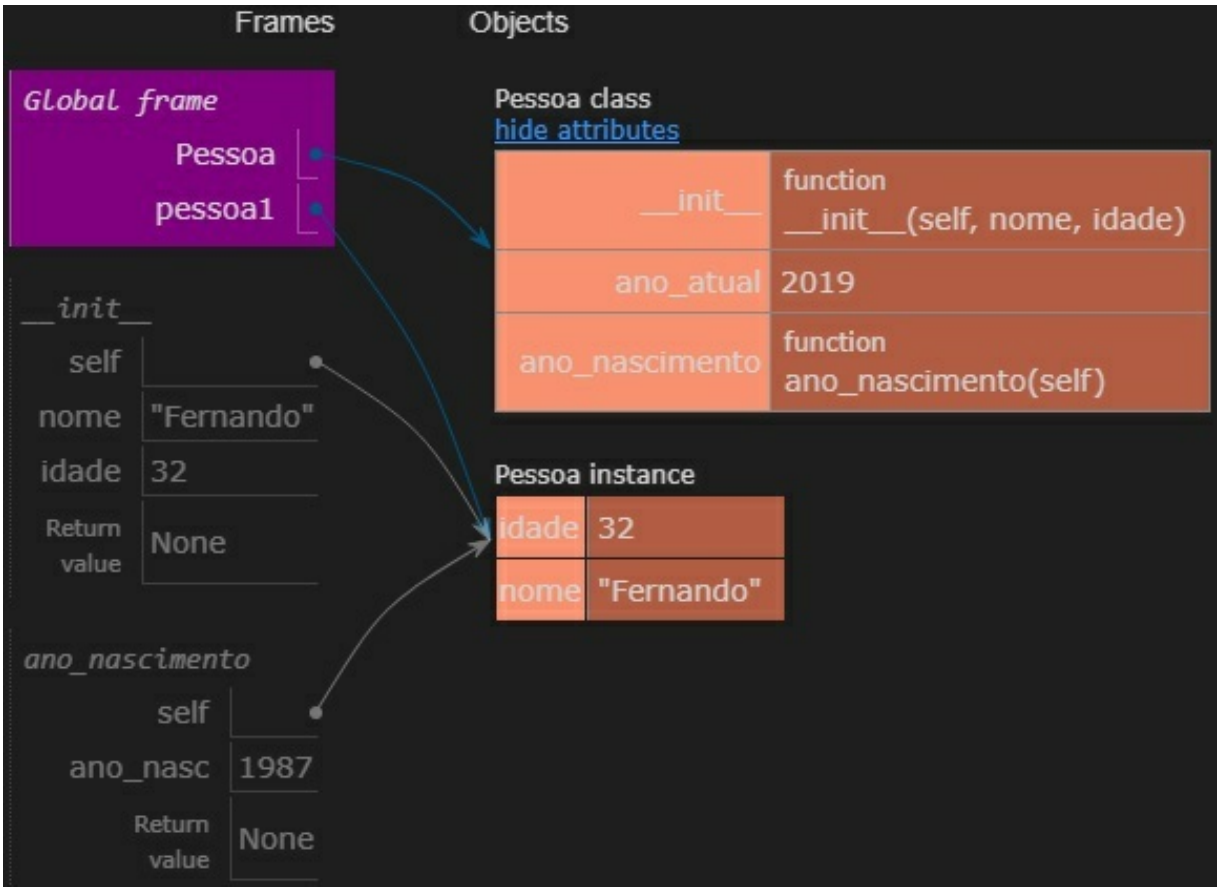
```
 1  class Pessoa:
 2      ano_atual = 2019
 3
 4      def __init__(self, nome, idade):
 5          self.nome = nome
 6          self.idade = idade
 7
 8      def ano_nascimento(self):
 9        ano_nasc = self.ano_atual - self.idade
10        print(f'Seu ano de nascimento é: {ano_nasc}')
11
12  pessoa1 = Pessoa('Fernando', 32)
13  print(pessoa1.ano_nascimento())
14
```

```
Seu ano de nascimento é: 1987
None
```

Na sequência é criada a variável pessoa1 que por sua vez inicializa a classe Pessoa passando como parâmetros ‘Fernando’, 32.

Por fim simplesmente pela função print( ) é exibida a mensagem resultante do cruzamento desses dados.

Nesse caso o retorno será: Seu ano de nascimento é 1987.


Frames
Objects
Global frame
Pessoa
pessoa1
__init__
self
nome
"Fernando"
idade
32
Return value
None
ano_nascimento
self
ano_nasc
1987
Return value
None
Pessoa class
hide attributes
__init__
function __init__(self, nome, idade)
ano_atual
2019
ano_nascimento
function ano_nascimento(self)
Pessoa instance
idade
32
nome
"Fernando"

Interação entre métodos de classe

Aprofundando um pouco nossos estudos, podemos criar um exemplo que engloba praticamente tudo o que foi visto até então.

Criaremos uma classe, com parâmetros opcionais alteráveis, que internamente realiza a interação entre métodos/funções para retornar algo ao usuário.

```
1  class Pessoa:
2      def __init__(self, nome, login=False, logoff=False):
3          self.nome = nome
4          self.login = login
5          self.logoff = logoff
6
7      def logar(self):
8          print(f'Bem vindo {self.nome}, você está logado no sistema.')
9          self.login = True
10
11 usuario = Pessoa('Fernando')
12 usuario.logar()
13
```

```
Bem vindo Fernando, você está logado no sistema.
```

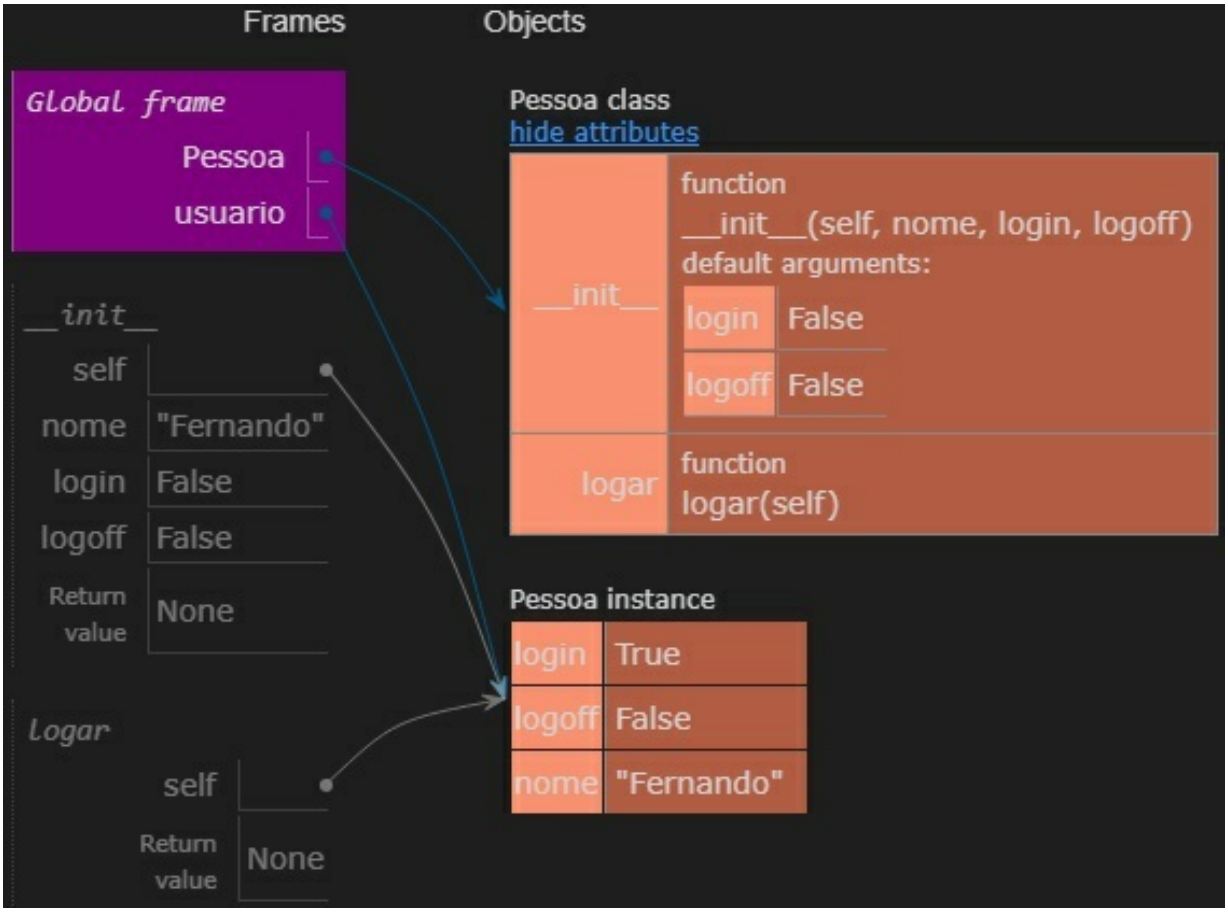
Inicialmente criamos a classe Pessoa, em seguida criamos o método construtor da mesma __init__ que recebe um nome atribuído pelo usuário e possui variáveis temporárias reservadas para login e logoff, por hora desabilitadas.

Em seguida criamos as respectivas variáveis internas instanciadas para o construtor por meio da sintaxe self.nomedavariavel.

Na sequência criamos uma função logar que simplesmente exibe em tela uma f string com máscara de substituição onde consta o nome atribuído a variável nome do método construtor, assim como atualiza o status de self.login de False para True. Fora da classe criamos uma variável chamada usuario que instancia a classe Pessoa passando como parâmetro ‘Fernando’, nome a ser substituído na variável homônima.

Por fim, dando o comando de usuário.logar( ), chamando a função

interna logar da classe Pessoa, o retorno será: Bem vindo Fernando, você está logado no sistema.



Estruturas condicionais em métodos de classe

Apenas um adendo a este tópico, pelo fato de que é bastante comum quando estamos aprendendo este tipo de abstração em programação esquecermos de criar estruturas condicionais ou validações em nosso código.

Pegando como exemplo o código anterior, supondo que fossem dados vários comandos print( ) sob a mesma função, ocorreria a repetição pura da mesma, o que normalmente não fará muito sentido dependendo da aplicação deste código.

Então, lembrando do que foi comentado anteriormente, dentro de uma classe podemos fazer o uso de qualquer estrutura de código, inclusive estruturas condicionais, laços de repetição e validações. Por exemplo:

```
1  class Pessoa:
2      def __init__(self, nome, login=False, logoff=False):
3          self.nome = nome
4          self.login = login
5          self.logoff = logoff
6
7      def logar(self):
8          print(f'Bem vindo {self.nome}, você está logado no sistema.')
9          self.login = True
10
11 usuario = Pessoa('Fernando')
12 usuario.logar()
13 usuario.logar()
14

Bem vindo Fernando, você está logado no sistema.
Bem vindo Fernando, você está logado no sistema.
```

O retorno será: Bem vindo Fernando, você está logado no sistema.

Bem vindo Fernando, você está logado no sistema.

Porém podemos criar uma simples estrutura que evita este tipo de erro, por meio de uma estrutura condicional que por sua vez para a execução do código quando seu objetivo for alcançado.

```python
class Pessoa:
    def __init__(self, nome, login=False, logoff=False):
        self.nome = nome
        self.login = login
        self.logoff = logoff

    def logar(self):
        if self.login:
            print(f'{self.nome} já está logado no sistema')
            return

        print(f'Bem vindo {self.nome}, você está logado no sistema.')
        self.login = True

usuario = Pessoa('Fernando')
usuario.logar()
usuario.logar()

```

```
Bem vindo Fernando, você está logado no sistema.
Fernando já está logado no sistema
```

Note que apenas foi criado uma condição dentro do método/função logar que, se self.login tiver o status como True, é exibida a respectiva mensagem e a execução do código para a partir daquele ponto.

Então, simulando um erro de lógica por parte do usuário, pedindo para que usuário faça duas vezes a mesma coisa, o retorno será:

Bem vindo Fernando, você está logado no sistema.

Fernando já está logado no sistema

```python
class Pessoa:
    def __init__(self, nome, login=False, logoff=False):
        self.nome = nome
        self.login = login
        self.logoff = logoff

    def logar(self):
        if self.login:
            print(f'{self.nome} já está logado no sistema')
            return

        print(f'Bem vindo {self.nome}, você está logado no sistema.')
        self.login = True

    def deslogar(self):
        if not self.login:
            print(f'{self.nome} não está logado no sistema')
            return
        print(f'{self.nome} foi deslogado do sistema')
        self.login = False

```

Apenas finalizando nossa linha de raciocínio para este tipo de código, anteriormente havíamos criado o atributo de classe logoff porém ainda não havíamos lhe dado um uso em nosso código.

Agora simulando que existe uma função específica para deslogar o sistema, novamente o que fizemos foi criar uma função deslogar onde consta uma estrutura condicional de duas validações, primeiro ela chega se o usuário não está logado no sistema, e a segunda, caso o usuário esteja logado, o desloga atualizando o status da variável self.login e exibe a mensagem correspondente.

```python
class Pessoa:
    def __init__(self, nome, login=False, logoff=False):
        self.nome = nome
        self.login = login
        self.logoff = logoff

    def logar(self):
        if self.login:
            print(f'{self.nome} já está logado no sistema')
            return

        print(f'Bem vindo {self.nome}, você está logado no sistema.')
        self.login = True

    def deslogar(self):
        if not self.login:
            print(f'{self.nome} não está logado no sistema')
            return
        print(f'{self.nome} foi deslogado do sistema')
        self.login = False

usuario = Pessoa('Fernando')
usuario.logar()
usuario.logar()
usuario.deslogar()

```

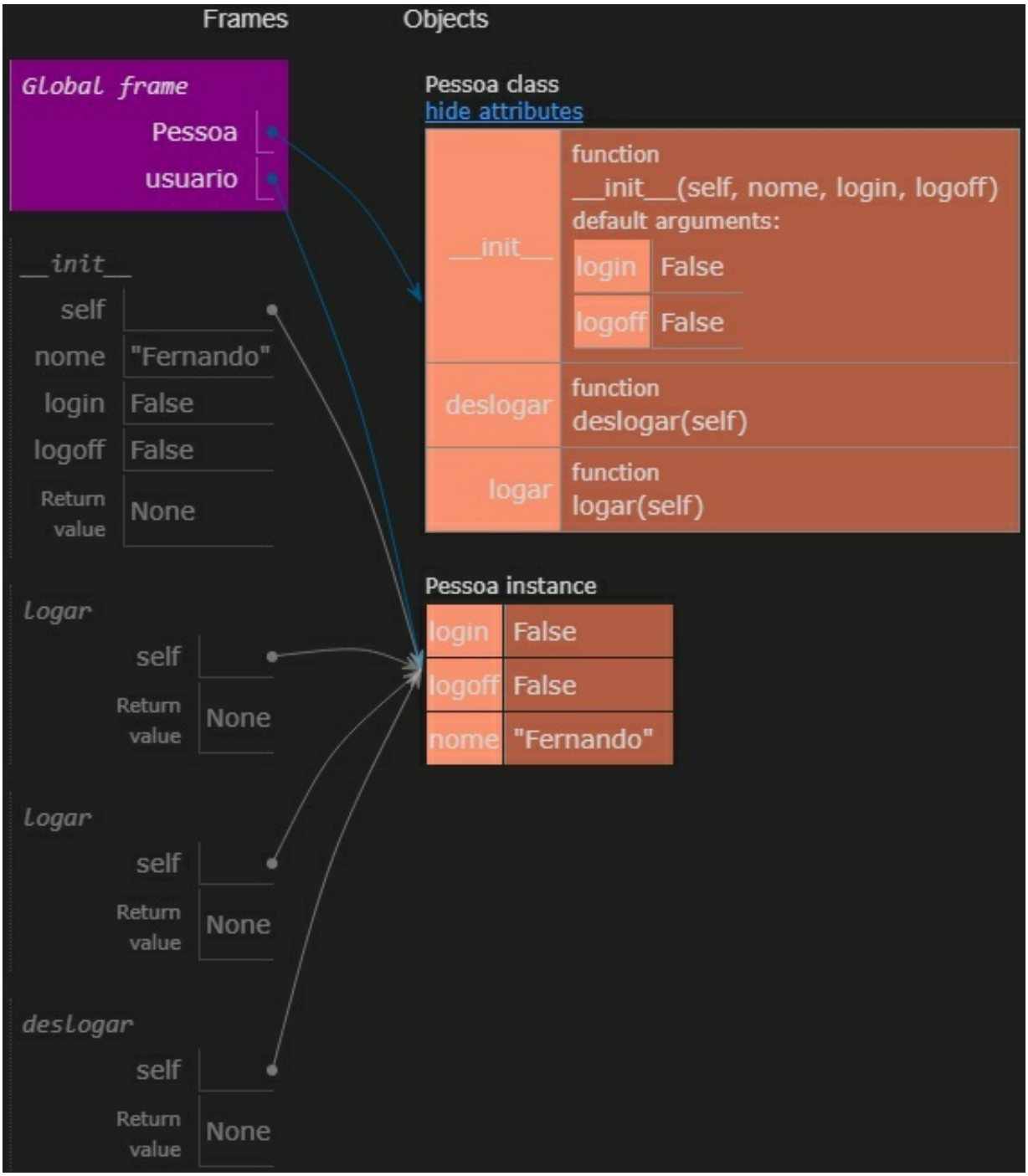
```
Bem vindo Fernando, você está logado no sistema.
Fernando já está logado no sistema
Fernando foi deslogado do sistema
```

Novamente simulando o erro, o retorno será:

Bem vindo Fernando, você está logado no sistema.

Fernando já está logado no sistema

Fernando foi deslogado do sistema


Frames
Objects
Global frame
Pessoa
usuario
__init__
self
nome
"Fernando"
login
False
logoff
False
Return value
None
logar
self
Return value
None
logar
self
Return value
None
deslogar
self
Return value
None
Pessoa class
hide attributes
__init__
function
__init__(self, nome, login, logoff)
default arguments:
login
False
logoff
False
deslogar
function
deslogar(self)
logar
function
logar(self)
Pessoa instance
login
False
logoff
False
nome
"Fernando"

Métodos de classe estáticos e dinâmicos

Outra aplicação muito usada, embora grande parte das vezes implícita ao código, é a definição manual de um método de classe quanto a sua visibilidade dentro do escopo da classe. Já vimos parte disso em tópicos anteriores, onde aprendemos que dependendo da indentação do código é criada uma hierarquia, onde determinadas variáveis dentro de funções/métodos de classe podem ser acessíveis e instanciáveis entre funções ou não, dependendo onde a mesma está declarada.

Em certas aplicações, é possível definir esse status de forma manual, e por meio deste tipo de declaração manual é possível fixar o escopo onde o método de classe irá retornar algum dado ou valor.

Em resumo, você pode definir manualmente se você estará criando objetos que utilizam de tudo o que está no escopo da classe, ou se o mesmo terá suas próprias atribuições isoladamente.

```
1  class Pessoa:
2      ano_atual = 2019
3
4      def __init__(self, nome, idade):
5          self.nome = nome
6          self.idade = idade
7
8      @classmethod
9      def ano_nascimento(cls, nome, ano_nascimento):
10         idade = cls.ano_atual - ano_nascimento
11         return cls(nome, idade)
12
```

Da mesma forma que as outras vezes, todo o processo se inicia com a criação da classe, note que estamos reutilizando o exemplo onde calculamos a idade do usuário visto anteriormente.

Inicialmente criada a classe Pessoa, dentro dela é criada a variável ano_atual

com 2019 como valor atribuído, logo após, criamos o método construtor da classe que receberá do usuário um nome e uma idade.

Em seguida declaramos manualmente que a classe a seguir é dinâmica, por meio de @classmethod (sintaxe igual a de um decorador), o que significa que dentro desse método/função, será realizada uma determinada operação lógica e o retorno da mesma é de escopo global, acessível e utilizável por qualquer bloco de código dentro da classe Pessoa.

Sendo assim, simplesmente criamos o método/função ano_nascimento que recebe como parâmetro cls, nome, ano_nascimento. Note que pela primeira vez, apenas como exemplo, não estamos criando uma função em self, aqui é criada uma variável a critério do programador, independente, que será instanciável no escopo global e internamente será sempre interpretada pelo interpretador como self.

Desculpe a redundância, mas apenas citei este exemplo para demonstrar que self, o primeiro parâmetro de qualquer classe, pode receber qualquer nomenclatura.

```
1  class Pessoa:
2      ano_atual = 2019
3
4      def __init__(self, nome, idade):
5          self.nome = nome
6          self.idade = idade
7
8      @classmethod
9      def ano_nascimento(cls, nome, ano_nascimento):
10         idade = cls.ano_atual - ano_nascimento
11         return cls(nome, idade)
12
13 pessoa2 = Pessoa.ano_nascimento('Fernando', 1987)
14 print(pessoa2.idade)
15

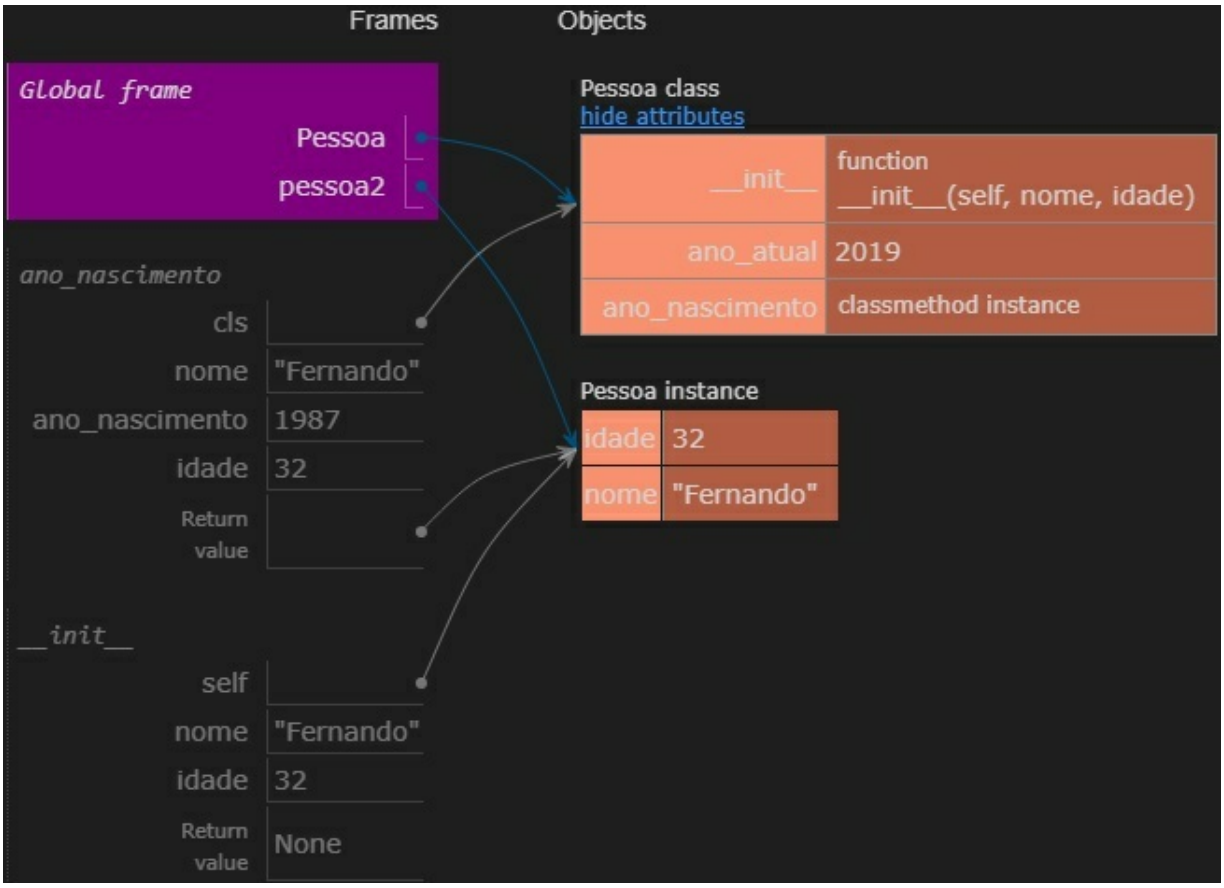
32
```

Por fim, repare que é criada uma variável de nome pessoa2 que instancia Pessoa e repassa atributos diretamente para o método

ano_nascimento.

Se não houver nenhum erro de sintaxe ocorrerão internamente o processamento das devidas funções assim como a atualização dos dados em todos métodos de classe.

Neste caso, por meio da função print( ) que recebe como parâmetro pessoa2.idade, o retorno será: 32



Entendido o conceito lógico de um método dinâmico (também chamado método de classe), hora de entender o que de fato é um método estático. Raciocine que todos métodos de classe / funções criadas até agora, faziam referência ao seu escopo (self) ou instância, assim como reservavam espaços para variáveis com atributos fornecidos pelo usuário.

Ao contrário disso, se declararmos um método que não possui instâncias como atributos, o mesmo passa a ser um método estático, como

uma simples função dentro da classe, acessível e instanciável por qualquer objeto.

Um método estático, por sua vez, pode ser declarado manualmente por meio da sintaxe @staticmethod uma linha antes do método propriamente dito.

```
 1  from random import randint
 2
 3  class Pessoa:
 4      @staticmethod
 5      def gerador_id():
 6          gerador = randint(100, 999)
 7          return gerador
 8
 9  print(Pessoa.gerador_id())
10
```

```
883
```

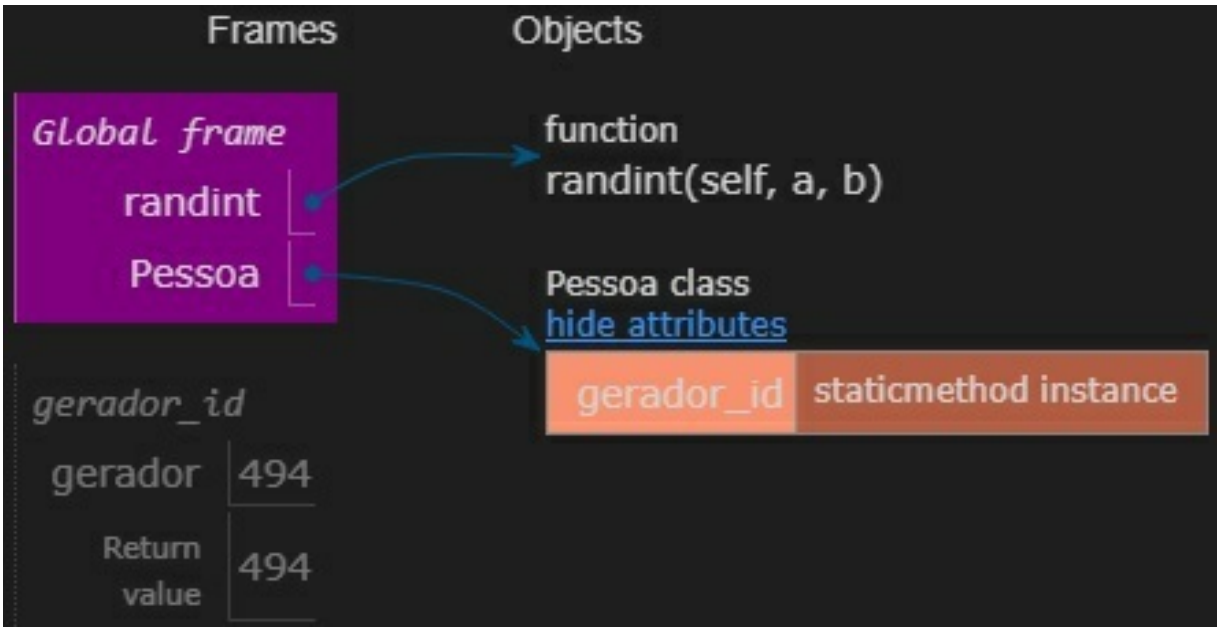
Mudando um pouco de exemplo, inicialmente importamos da biblioteca random o módulo randint, muito usado quando é necessário se gerar números inteiros aleatórios.

Em seguida é criada a classe Pessoa, é declarada que o método gerador_id( ) dentro dela é um método estático pelo @staticmethod.

O método gerador_id( ) por sua vez simplesmente possui uma variável de nome gerador que inicializa o módulo randint especificando que seja gerado um número aleatório entre 100 e 999, retornando esse valor para gerador.

Por fim, por meio da função print( ) podemos diretamente instanciar como parâmetro a classe Pessoa executando seu método estático gerador_id( ).

Neste caso o retorno será um número gerado aleatoriamente entre 100 e 999.


Frames
Objects
Global frame
randint
Pessoa
function
randint(self, a, b)
Pessoa class
hide attributes
gerador_id
staticmethod instance
gerador_id
gerador
494
Return value
494

Getters e Setters

Anteriormente vimos rapidamente que é perfeitamente possível criarmos estruturas condicionais em nossos métodos de classe, para que dessa forma tenhamos estruturas de código que tentam contornar as possíveis exceções cometidas pelo usuário (normalmente essas exceções são inserções de tipos de dados não esperados pelo interpretador).

Aprofundando um pouco mais dentro desse conceito, podemos criar estruturas de validação que contornem erros em atributos de uma classe. Pela nomenclatura usual, um Getter obtém um determinado dado/valor e um Setter configura um determinado dado/valor que substituirá o primeiro.

```
1  class Produto:
2      def __init__(self, nome, preco):
3          self.nome = nome
4          self.preco = preco
5
6  produto1 = Produto('Processador', 370)
7
8  print(produto1.preco)
9

370
```

Como sempre, o processo se inicia com a criação de nossa classe, nesse caso, de nome Produto, dentro dela criamos um construtor onde basicamente iremos criar objetos fornecendo o nome e o preço do mesmo para suas respectivas variáveis.

De fora da classe, criamos uma variável de nome produto1 que recebe como atributo a classe Produto passando como atributos ‘Processador’, 370.

Por meio do comando print( ) podemos exibir por exemplo, o preço de produto1, por meio de produto1.preco. Neste caso o retorno será 370

```
1  class Produto:
2      def __init__(self, nome, preco):
3          self.nome = nome
4          self.preco = preco
5
6      def desconto(self, percentual):
7          self.preco = self.preco - (self.preco*(percentual/100))
8
9  produto1 = Produto('Processador', 370)
10 produto1.desconto(15)
11
12 produto2 = Produto('Placa Mãe', 'R$280')
13 produto2.desconto(20)
14
15 print(produto1.preco)
16 print(produto2.preco)
17
```

Na sequência adicionamos um método de classe de nome desconto responsável por aplicar um desconto com base no percentual definido pelo usuário. Também cadastramos um segundo produto e note que aqui estamos simulando uma exceção no valor atribuído ao preço do mesmo.

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call l
<ipython-input-49-df1d7672c173> in <module>()
     11
     12 produto2 = Produto('Placa Mãe', 'R$280')
---> 13 produto2.desconto(20)
     14
     15 print(produto1.preco)

<ipython-input-49-df1d7672c173> in desconto(self, percentual)
      5
      6     def desconto(self, percentual):
----> 7         self.preco = self.preco - (self.preco*(percentual/100))
      8
      9 produto1 = Produto('Processador', 370)

TypeError: can't multiply sequence by non-int of type 'float'
```

Já que será executada uma operação matemática para aplicar desconto sobre os valores originais, o interpretador espera que todos dados a

serem processados sejam do tipo int ou float (numéricos).

Ao tentar executar tal função ocorrerá um erro: TypeError: can't multiply sequence by non-int of type 'float'.

Em tradução livre: Erro de tipo: Não se pode multiplicar uma sequência de não-inteiros pelo tipo ‘float’ (número com casas decimais).

Sendo assim, teremos de criar uma estrutura de validação que irá prever esse tipo de exceção e contornar a mesma, como estamos trabalhando com orientação a objetos, mais especificamente tendo que criar um validador dentro de uma classe, isto se dará por meio de Getters e Setters.

```python
class Produto:
    def __init__(self, nome, preco):
        self.nome = nome
        self.preco = preco

    @property #Getter
    def preco(self):
        return self.preco_valido

    @preco.setter #Setter
    def preco(self, valor):
        if isinstance(valor, str):
            valor = float(valor.replace('R$',''))
        self.preco_valido = valor

    def desconto(self, percentual):
        self.preco = self.preco - (self.preco*(percentual/100))

produto1 = Produto('Processador', 370)
produto1.desconto(15)

produto2 = Produto('Placa Mãe', 'R$280')
produto2.desconto(20)

print(produto1.preco)
print(produto2.preco)

```

```
314.5
224.0
```

Prosseguindo com nosso código, indentado como método de nossa classe Pessoa, iremos incorporar ao código a estrutura acima.

Inicialmente criamos o que será nosso Getter, um decorador (que por sua vez terá prioridade 1 na leitura do interpretador) que cria uma função preco, simplesmente obtendo o valor atribuído a preco e o replicando em uma nova variável de nome preco_valido.

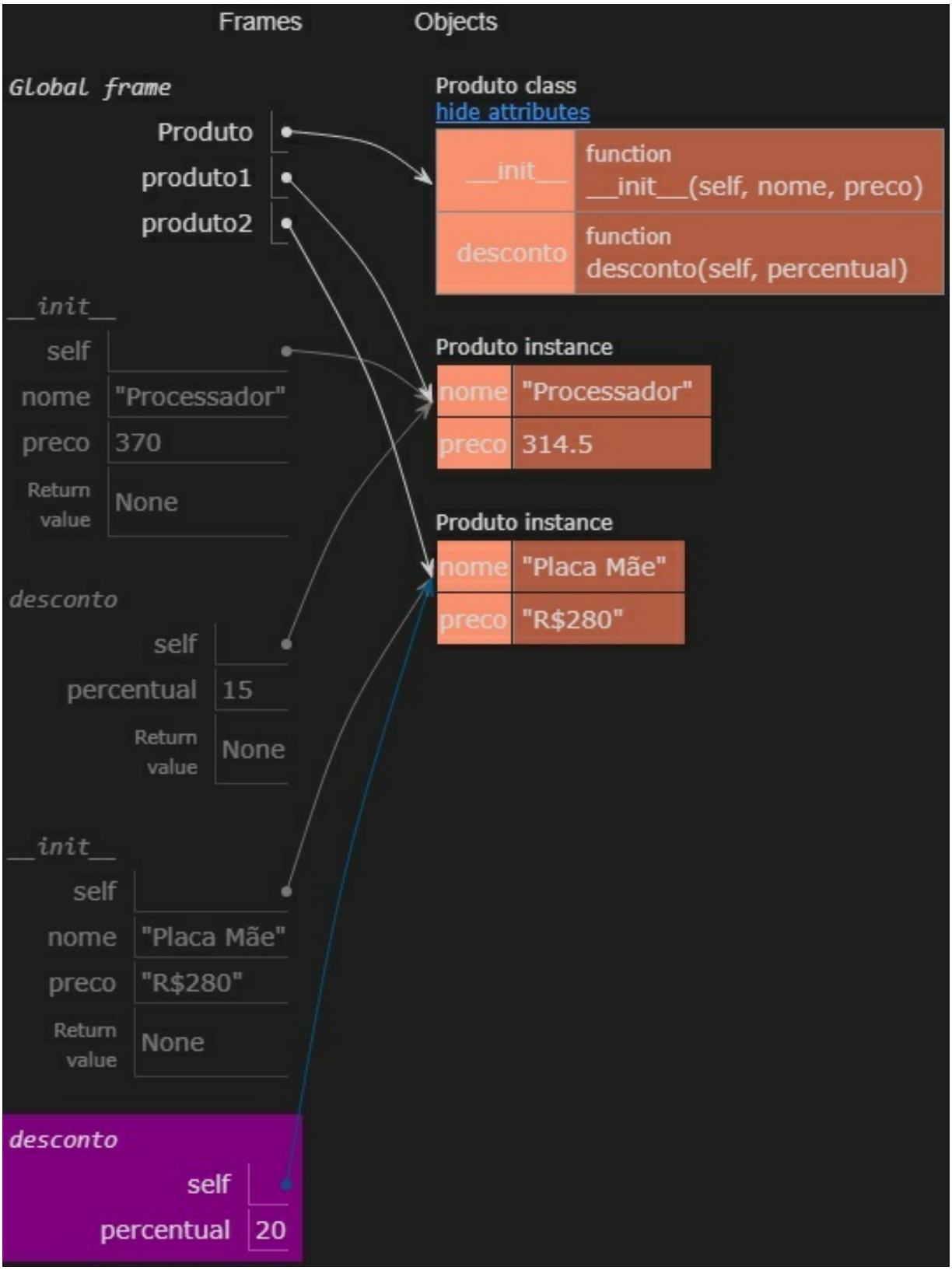
Feito isso, hora de criarmos nosso Setter, um pouco mais complexo, porém um tipo de validador funcional. Inicialmente criamos um decorador de nome @preco.setter (esse decorador deverá ter o nome da variável em questão, assim como a palavra reservada .setter).

Então criamos a função preco que recebe um valor, em seguida é criada uma estrutura condicional onde, se o valor da instância for do tipo str, valor será convertido para float assim como os caracteres desnecessários serão removidos.

Por fim, preco_valido é atualizado com o valor de valor.

Realizadas as devidas correções e validações, é aplicado sobre o preço de produto1 um desconto de 15%, e sobre o preço de produto2 20%. Desta forma, o retorno será de: 314.5

224.0


Frames
Objects
Global frame
Produto
produto1
produto2
__init__
self
nome
"Processador"
preco
370
Return value
None
desconto
self
percentual
15
Return value
None
__init__
self
nome
"Placa Mãe"
preco
"R$280"
Return value
None
desconto
self
percentual
20
Produto class
hide attributes
__init__
function __init__(self, nome, preco)
desconto
function desconto(self, percentual)
Produto instance
nome
"Processador"
preco
314.5
Produto instance
nome
"Placa Mãe"
preco
"R$280"

Encapsulamento

Se você já programa em outras linguagens está habituado a declarar manualmente em todo início de código, se o bloco de código em questão é de acesso público ou privado. Esta definição em Python é implícita, podendo ser definida manualmente caso haja necessidade.

Por hora, se tratando de programação orientada a objetos, dependendo da finalidade de cada bloco de código, é interessante você definir as permissões de acesso aos mesmos no que diz respeito a sua leitura e escrita.

Digamos que existam determinadas classes acessíveis e mutáveis de acordo com a finalidade e com sua interação com o usuário, da mesma forma existirão classes com seus respectivos métodos que devem ser protegidos para que o usuário não tenha acesso total, ou ao menos, não tenha como modifica-la.

Raciocine que classes e funções importantes podem e devem ser modularizadas, assim como encapsuladas de acordo com sua finalidade, e na prática isto é feito de forma mais simples do que você imagina, vamos ao exemplo:

```
1  class BaseDeDados:
2      def __init__(self):
3          self.dados = {}
4
5  base = BaseDeDados()
6  print(base.dados)
7
```

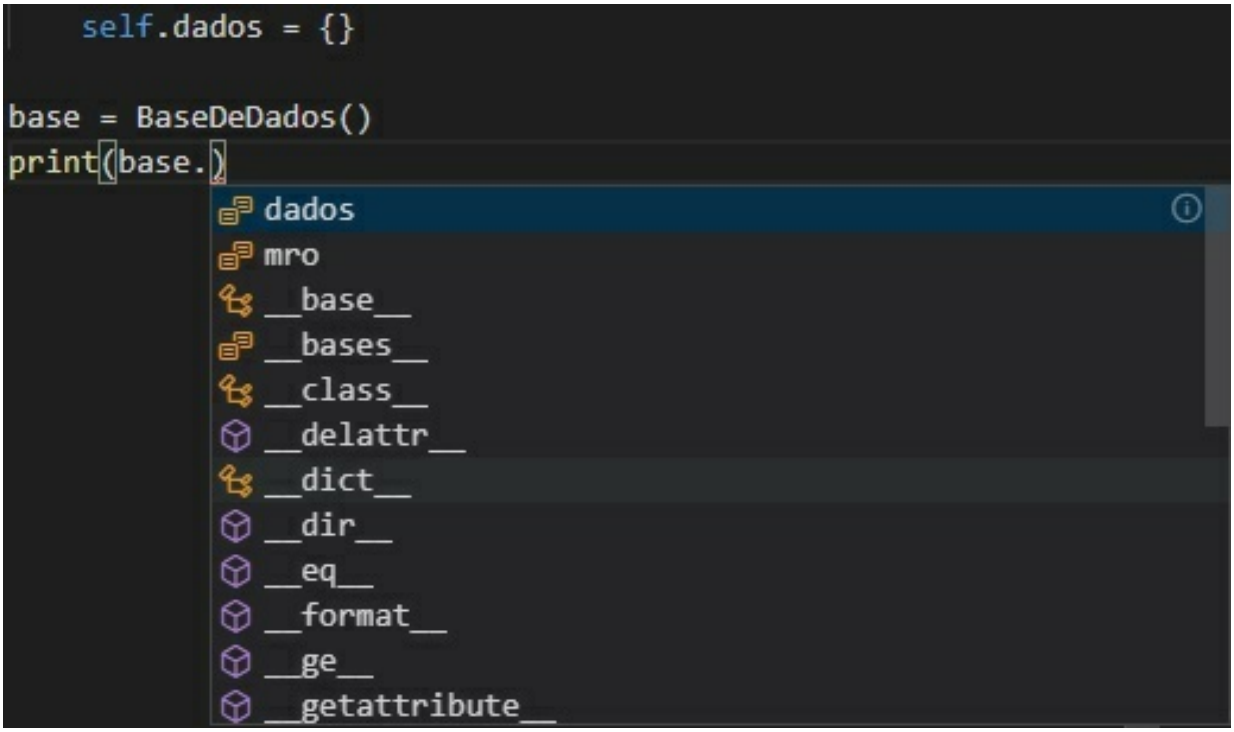
```
{}
```

Repare que aqui, inicialmente, estamos criando uma classe como já estamos habituados, neste caso, uma classe chamada BaseDeDados, dentro da mesma há um construtor onde dentro dele simplesmente existe a criação de uma variável na própria instância que pela sintaxe receberá um dicionário,

apenas como exemplo mesmo.

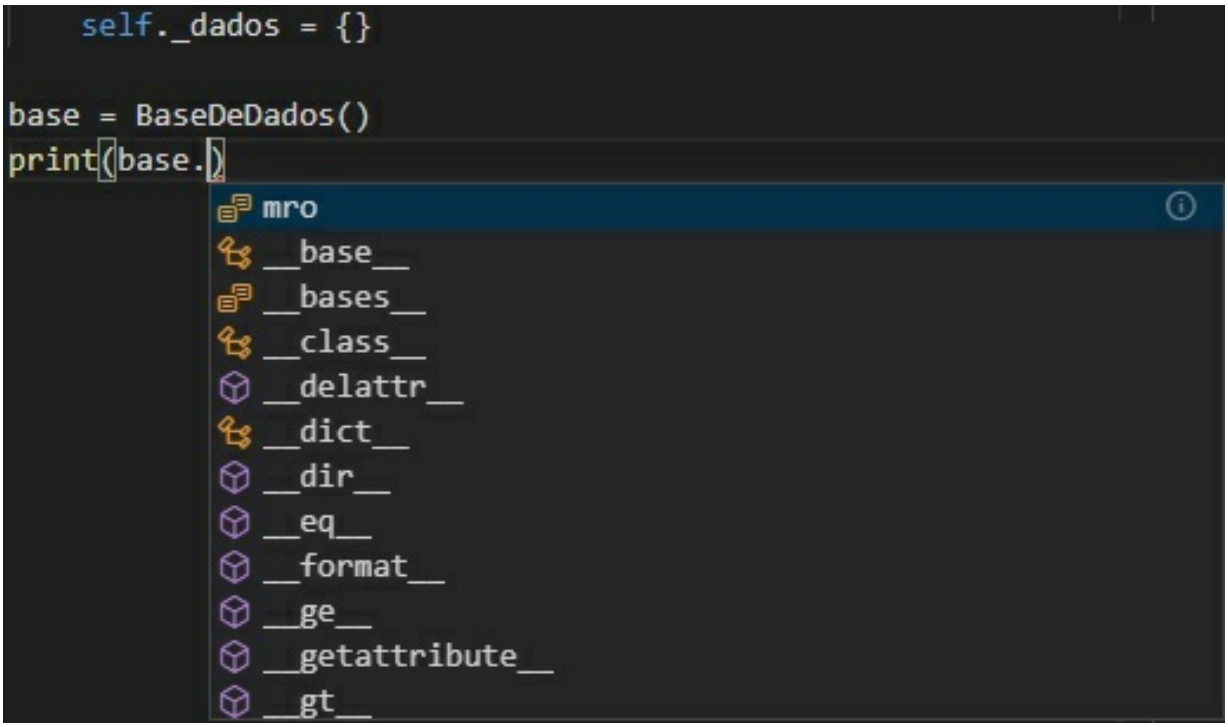
Fora da classe é criada uma variável de nome base que inicializa a classe BaseDeDados assim como um comando para exibir o conteúdo de dados.

O ponto chave aqui é, self.dados é uma variável, um atributo de classe que quando declarado sob esta nomenclatura, é visível e acessível tanto dentro dessa instância quanto de fora dela.

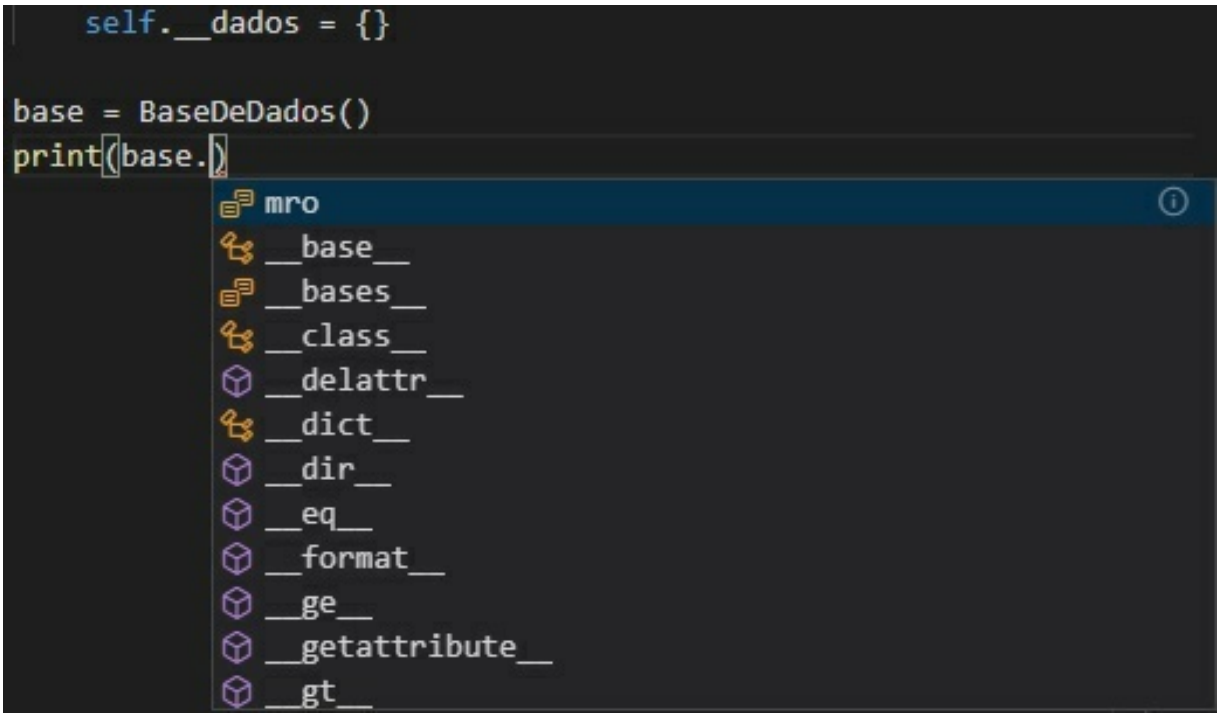


Note que em nossa função print( ) ao passar como parâmetro base, sua variável dados está disponível para ser instanciada.

A partir do momento que declaramos a variável dados pela sintaxe _dados, a mesma passa a ser uma variável protegida, não visível, ainda acessível de fora da classe, porém implícita.



Repare que ao passar base como parâmetro, dados não está mais visível, ela pode ser declarada manualmente normalmente, porém supondo que este é um código compartilhado, onde os desenvolvedores tem níveis de acesso diferentes ao código, neste exemplo, dados não estaria visível para todos, logo, é um atributo de classe protegido restrito somente a quem sabe sua instância.



Por fim, declarando a variável dados como __dados, a mesma passa a ser privada, e dessa forma, a mesma é inacessível e imutável de fora da classe.

Lembre-se que toda palavra em Python, com prefixo __ (underline duplo) é uma palavra reservada ao sistema, e neste caso não é diferente.

Em resumo, você pode definir a visibilidade de uma variável por meio da forma com que declara, uma underline torna a mesma protegida, underline duplo a torna privada e imutável. Por exemplo:

```python
class BaseDeDados:
    def __init__(self):
        self.base = {}

    def inserir(self, nome, fone):
        if 'clientes' not in self.base:
            self.base['clientes'] = {nome:fone}
        else:
            self.base['clientes'].update({nome:fone})

    def listar(self):
        for nome, fone in self.base['clientes'].items():
            print(nome, fone)

    def apagar(self, nome):
        del self.base['clientes'][nome]

```

Apenas pondo em prática o conceito explicado anteriormente, supondo que tenhamos um sistema comercial onde sua programação foi feita sem as devidas proteções de código realizadas via encapsulamento.

Inicialmente é criada uma classe BaseDeDados com um método construtor assim como métodos de classe para inserir, listar e apagar clientes de um dicionário, por meio de seus respectivos nomes e telefones.

Note que nenhum encapsulamento foi realizado (assim como nesse caso, apenas como exemplo, esta base de dados não está modularizada), toda e qualquer parte do código neste momento é acessível e alterável por qualquer desenvolvedor.

```python
class BaseDeDados:
    def __init__(self):
        self.base = {}

    def inserir(self, nome, fone):
        if 'clientes' not in self.base:
            self.base['clientes'] = {nome:fone}
        else:
            self.base['clientes'].update({nome:fone})

    def listar(self):
        for nome, fone in self.base['clientes'].items():
            print(nome, fone)

    def apagar(self, nome):
        del self.base['clientes'][nome]

relClientes = BaseDeDados()

relClientes.inserir('Ana', 991358899)
relClientes.inserir('Fernando', 981252001)
relClientes.inserir('Maria', 999111415)

relClientes.listar()

```

```
Ana 991358899
Fernando 981252001
Maria 999111415
```

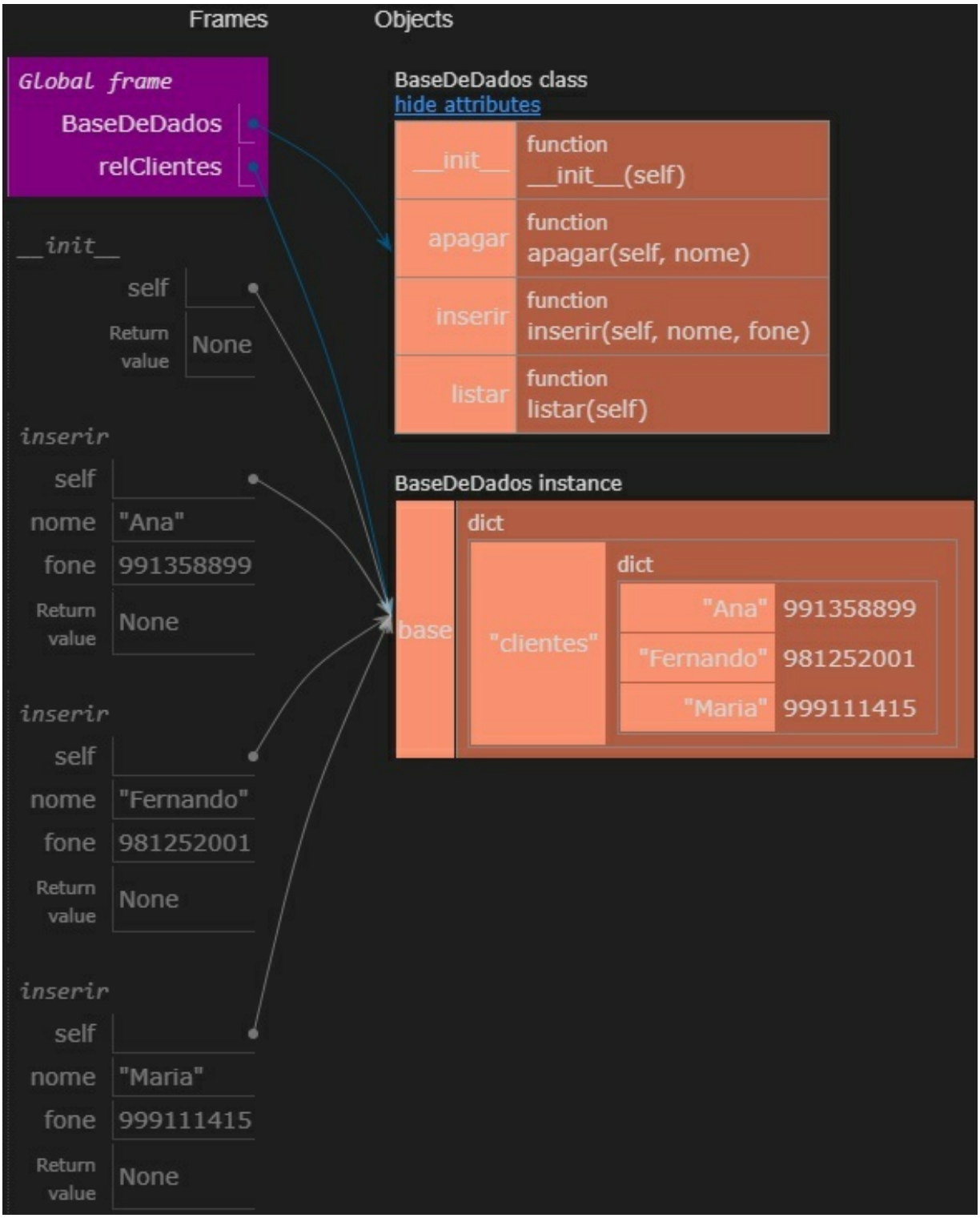
Em seguida é criada uma variável que instancia BaseDeDados( ). Dessa forma, é possível por meio dessa variável usar os métodos internos da classe para suas referidas funções, nesse caso, adicionamos 3 clientes a base de dados e em seguida simplesmente pedimos a listagem dos mesmos por meio do método de classe listar.

O retorno será:

Ana 991358899

Fernando 981252001

Maria 999111415



Agora simulando uma exceção cometida por um desenvolvedor,

supondo que o mesmo pegue como base este código, se em algum momento ele fizer qualquer nova atribuição para a variável relClientes o que ocorrerá é que a mesma atualizará toda a classe quebrando assim todo código interno pronto anteriormente.

```
18  relClientes = BaseDeDados()
19
20  relClientes.inserir('Ana', 991358899)
21  relClientes.inserir('Fernando', 981252001)
22  relClientes.inserir('Maria', 999111415)
23
24  relClientes.base = 'Novo Banco de Dados'
25  relClientes.listar()
26
```

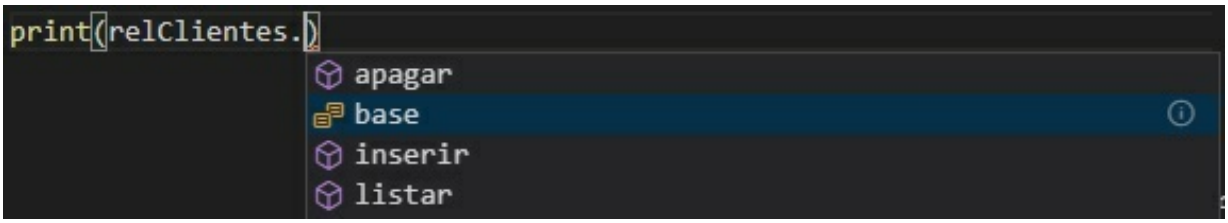
O retorno será:

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call
<ipython-input-55-728e1ffc3e97> in <module>()
     24 relClientes.listar()
     25 relClientes.base = 'Novo Banco de Dados'
---> 26 relClientes.listar()

<ipython-input-55-728e1ffc3e97> in listar(self)
     10
     11     def listar(self):
---> 12         for nome, fone in self.base['clientes'].items():
     13             print(nome, fone)
     14

TypeError: string indices must be integers
```

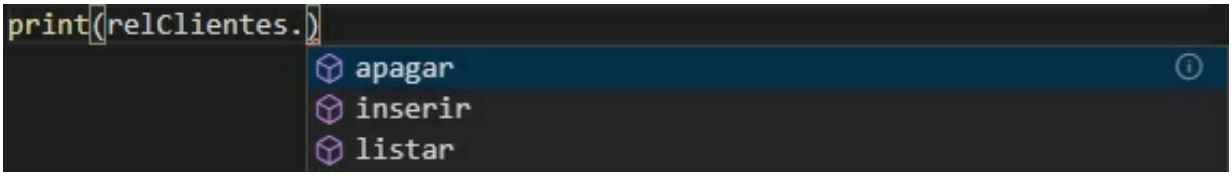
Isto se deu pelo fato de que, como não havia o devido encapsulamento para proteger a integridade da base de dados, a mesma foi simplesmente substituída pelo novo atributo e a partir desse momento, nenhum bloco de código dos métodos de classe criados anteriormente serão utilizados, uma vez que todos eles dependem do núcleo de self.base que consta no método construtor da classe.



Note que até esse momento ao tentar instanciar qualquer coisa de relClientes, entre os métodos aparece inclusive a base, que deveria ser protegida.

```
1  class BaseDeDados:
2      def __init__(self):
3          self.__base = {}
4
```

Realizando a devida modularização, renomeando todas as referências a base para __base, a partir deste momento a mesma passa a ser um objeto de classe com as devidas proteções de visualização e contra modificação.



Executando o mesmo comando print( ) como anteriormente, repare que agora base não aparece mais como um objeto instanciável para um usuário desse nível. Ou que ao menos não deveria ser instanciável.

Porém, ainda assim é possível aplicar mudanças sobre este objeto, mas como ele está definido de forma a ser imutável, o que acontecerá é que o interpretador irá fazer uma espécie de backup do conteúdo antigo, com núcleo protegido e imutável, que pode ser restaurado a qualquer momento.

```python
class BaseDeDados:
    def __init__(self):
        self.__base = {}

    def inserir(self, nome, fone):
        if 'clientes' not in self.__base:
            self.__base['clientes'] = {nome:fone}
        else:
            self.__base['clientes'].update({nome:fone})

    def listar(self):
        for nome, fone in self.__base['clientes'].items():
            print(nome, fone)

    def apagar(self, nome):
        del self.__base['clientes'][nome]

relClientes = BaseDeDados()
relClientes.inserir('Ana', 991358899)
relClientes.inserir('Fernando', 981252001)
relClientes.inserir('Maria', 999111415)

relClientes.listar()
relClientes.__base = 'Novo Banco de Dados'
print(relClientes.__base)

```

```
Ana 991358899
Fernando 981252001
Maria 999111415
Novo Banco de Dados
```

O interpretador verá que base é uma variável protegida, e que manualmente você ainda assim está forçando atribuir a mesma um novo dado/valor. Sendo assim, evitando uma quebra de código, ele irá executar esta função.

Neste caso, o retorno será Novo Banco de Dados.

```
15        def apagar(self, nome):
16            del self.__base['clientes'][nome]
17
18    relClientes = BaseDeDados()
19    relClientes.inserir('Ana', 991358899)
20    relClientes.inserir('Fernando', 981252001)
21    relClientes.inserir('Maria', 999111415)
22
23    relClientes.listar()
24    relClientes.__base = 'Novo Banco de Dados'
25    print(relClientes.__base)
26    print(relClientes._BaseDeDados__base)
27
```

```
Ana 991358899
Fernando 981252001
Maria 999111415
Novo Banco de Dados
{'clientes': {'Ana': 991358899, 'Fernando': 981252001, 'Maria': 999111415}}
```

Por meio da função print( ), passando como parâmetro _BaseDeDados__base você tem acesso ao núcleo original, salvo pelo interpretador.

Dessa forma o resultado será: {'clientes': {'Ana': 991358899, 'Fernando': 981252001, 'Maria': 999111415}}



Este é um ponto que gera bastante confusão quando estamos aprendendo sobre encapsulamento de estruturas de uma classe.

Visando por convenção facilitar a vida de quem programa, é sempre interessante quando for necessário fazer este tipo de alteração no código deixar as alterações devidamente comentadas, assim como ter o discernimento de que qualquer estrutura de código prefixada com _ deve ser tratada como um código protegido, e somente como última alternativa realizar este tipo de alterações.

Associação de classes

Associação de classes, ou entre classes, é uma prática bastante comum uma vez que nossos programas dependendo sua complexidade não se restringirão a poucas funções básicas, dependendo do que o programa oferece ao usuário uma grande variedade de funções internas são executadas ou ao menos estão à disposição do usuário para que sejam executadas.

Internamente estas funções podem se comunicar e até mesmo trabalhar em conjunto, assim como independentes elas não dependem uma da outra.

Raciocine que esta prática é muito utilizada pois quando se está compondo o código de um determinado programa é natural criar uma função para o mesmo e, depois de testada e pronta, esta pode ser modularizada, ou ser separada em um pacote, etc... de forma que aquela estrutura de código, embora parte do programa, independente para que se faça a manutenção da mesma assim como novas implementações de recursos.

É normal, dependendo da complexidade do programa, que cada “parte” de código seja na estrutura dos arquivos individualizada de forma que o arquivo que guarda a mesma quando corrompido, apenas em último caso faça com que o programa pare de funcionar.

Se você já explorou as pastas dos arquivos que compõe qualquer programa deve ter se deparado com milhares de arquivos dependendo o programa. A manutenção dos mesmos se dá diretamente sobre os arquivos que apresentam algum problema.

```python
class Usuario:
    def __init__(self, nome):
        self.__nome = nome
        self.__logar = None
    @property
    def nome(self):
        return self.__nome
    @property
    def logar(self):
        return self.__logar
    @logar.setter
    def logar(self, logar):
        self.__logar = logar

```

Partindo para a prática, note que em primeiro lugar criamos nossa classe Usuario, a mesma possui um método construtor que recebe um nome (com variável homônima declarada como privada), em seguida precisamos realizar a criação de um getter e setter para que possamos instanciar os referentes métodos de fora dessa classe.

Basicamente, pondo em prática tudo o que já vimos até então, esta seria a forma adequada de permitir acesso aos métodos de classe de Usuario, quando as mesmas forem privadas ou protegidas.

```python
class Usuario:
    def __init__(self, nome):
        self.__nome = nome
        self.__logar = None
    @property
    def nome(self):
        return self.__nome
    @property
    def logar(self):
        return self.__logar
    @logar.setter
    def logar(self, logar):
        self.__logar = logar

class Identificador:
    def __init__(self, numero):
        self.__numero = numero
    @property
    def numero(self):
        return self.__numero
    def logar(self):
        print('Logando no sistema...')

```

Na sequência criamos uma nova classe de nome Identificador que aqui, apenas como exemplo, seria o gatilho para um sistema de autenticação de usuário em um determinado sistema.

Logo, a mesma também possui um método construtor que recebe um número para identificação do usuário, novamente, é criado um getter e um setter para tornar o núcleo desse código instanciável.

```python
class Usuario:
    def __init__(self, nome):
        self.__nome = nome
        self.__logar = None
    @property
    def nome(self):
        return self.__nome
    @property
    def logar(self):
        return self.__logar
    @logar.setter
    def logar(self, logar):
        self.__logar = logar

class Identificador:
    def __init__(self, numero):
        self.__numero = numero
    @property
    def numero(self):
        return self.__numero
    def logar(self):
        print('Logando no sistema...')

usuario1 = Usuario('Fernando')
identificador1 = Identificador('0001')

usuario1.logar = identificador1
usuario1.logar.logar()

```

```
Logando no sistema...
```

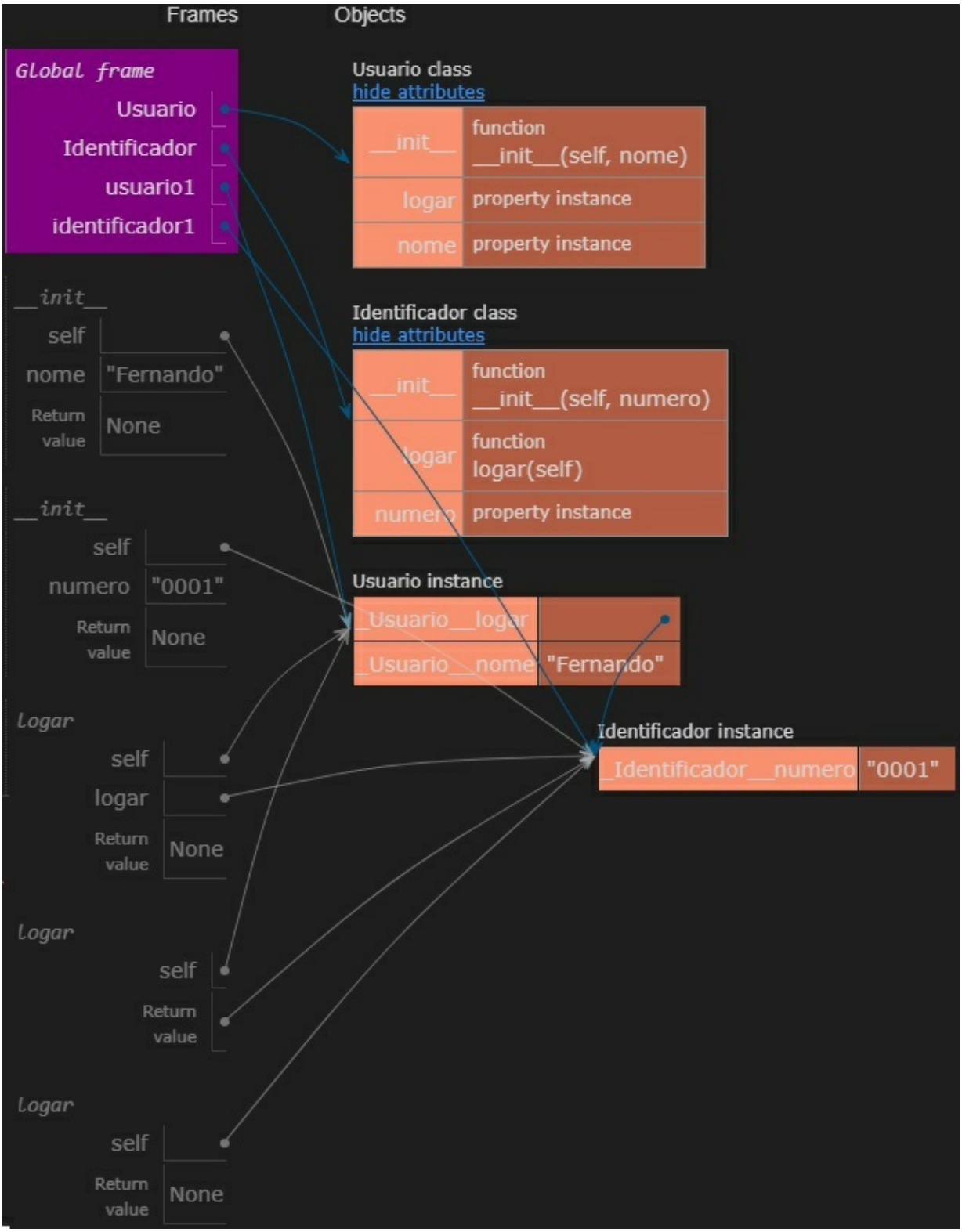
Por fim, é criado um objeto no corpo de nosso código de nome usuario1 que instancia a classe Usuario, lhe passando como atributo ‘Fernando’. O mesmo é feito para o objeto identificador1, que instancia a classe Identificador passando como parâmetro ‘0001’.

Agora vem a parte onde estaremos de fato associando essas classes

e fazendo as mesmas trabalhar em conjunto.

Para isso por meio de usuário.logar = identificador associamos 0001 a Fernando, e supondo que essa é a validação necessária para esse sistema de identificação (ou pelo menos uma das etapas).

Por meio da função usuario1.logar.logar( ) o retorno é: Logando no sistema...



Raciocine que este exemplo básico é apenas para entender a lógica de associação, obviamente um sistema de autenticação iria exigir mais dados como senha, etc... assim como confrontar esses dados com um banco de

dados...

Também é importante salientar que quanto mais associações houver em nosso código, de forma geral mais eficiente é a execução do mesmo ao invés de blocos e mais blocos de códigos independentes e repetidos para cada situação, porém, quanto mais robusto for o código, maior também será a suscetibilidade a erro, sendo assim, é interessante testar o código a cada bloco criado para verificar se não há nenhum erro de lógica ou sintaxe.

Agregação e composição de classes

Começando o entendimento pela sua lógica, anteriormente vimos o que basicamente é a associação de classes, onde temos mais de uma classe, elas podem compartilhar métodos e atributos entre si ao serem instanciadas, porém uma pode ser executada independentemente da outra.

Já quando falamos em agregação e composição, o laço entre essas classes e suas estruturas passa a ser tão forte que uma depende da outra. Raciocine que quando estamos realizando a composição de uma classe, uma classe tomará posse dos objetos das outras classes, de modo que se algo corromper essa classe principal, as outras param de funcionar também.

```
1  class Contato:
2      def __init__(self, residencial, celular):
3          self.residencial = residencial
4          self.celular = celular
5
6  class Cliente:
7      def __init__(self, nome, idade, fone=None):
8          self.nome = nome
9          self.idade = idade
10         self.fone = []
11
12     def addFone(self, residencial, celular):
13         self.fone.append(Contato(residencial, celular))
14     def listaFone(self):
15         for fone in self.fone:
16             print(fone.residencial, fone.celular)
17
```

Para esse exemplo, inicialmente criamos uma classe de nome Cliente, ela possui seu método construtor que recebe um nome, uma idade e um ou mais telefones para contato, isto porque como objeto de classe fone inicialmente recebe uma lista em branco.

Em seguida é criado um método de classe responsável por

alimentar essa lista com números de telefone. Porém, repare no diferencial, em self.fone, estamos adicionando dados instanciando como parâmetro outra classe, nesse caso a classe Contato com toda sua estrutura declarada anteriormente.

Como dito anteriormente, este laço, instanciando uma classe dentro de outra classe é um laço forte onde, se houver qualquer exceção no código, tudo irá parar de funcionar uma vez que tudo está interligado operando em conjunto.

```
class Contato:
    def __init__(self, residencial, celular):
        self.residencial = residencial
        self.celular = celular

class Cliente:
    def __init__(self, nome, idade, fone=None):
        self.nome = nome
        self.idade = idade
        self.fone = []

    def addFone(self, residencial, celular):
        self.fone.append(Contato(residencial, celular))
    def listaFone(self):
        for fone in self.fone:
            print(fone.residencial, fone.celular)

cliente1 = Cliente('Fernando', 32)
cliente1.addFone(33221766, 991357258)
print(cliente1.nome)
print(cliente1.listaFone())

```

```
Fernando
33221766 991357258
```

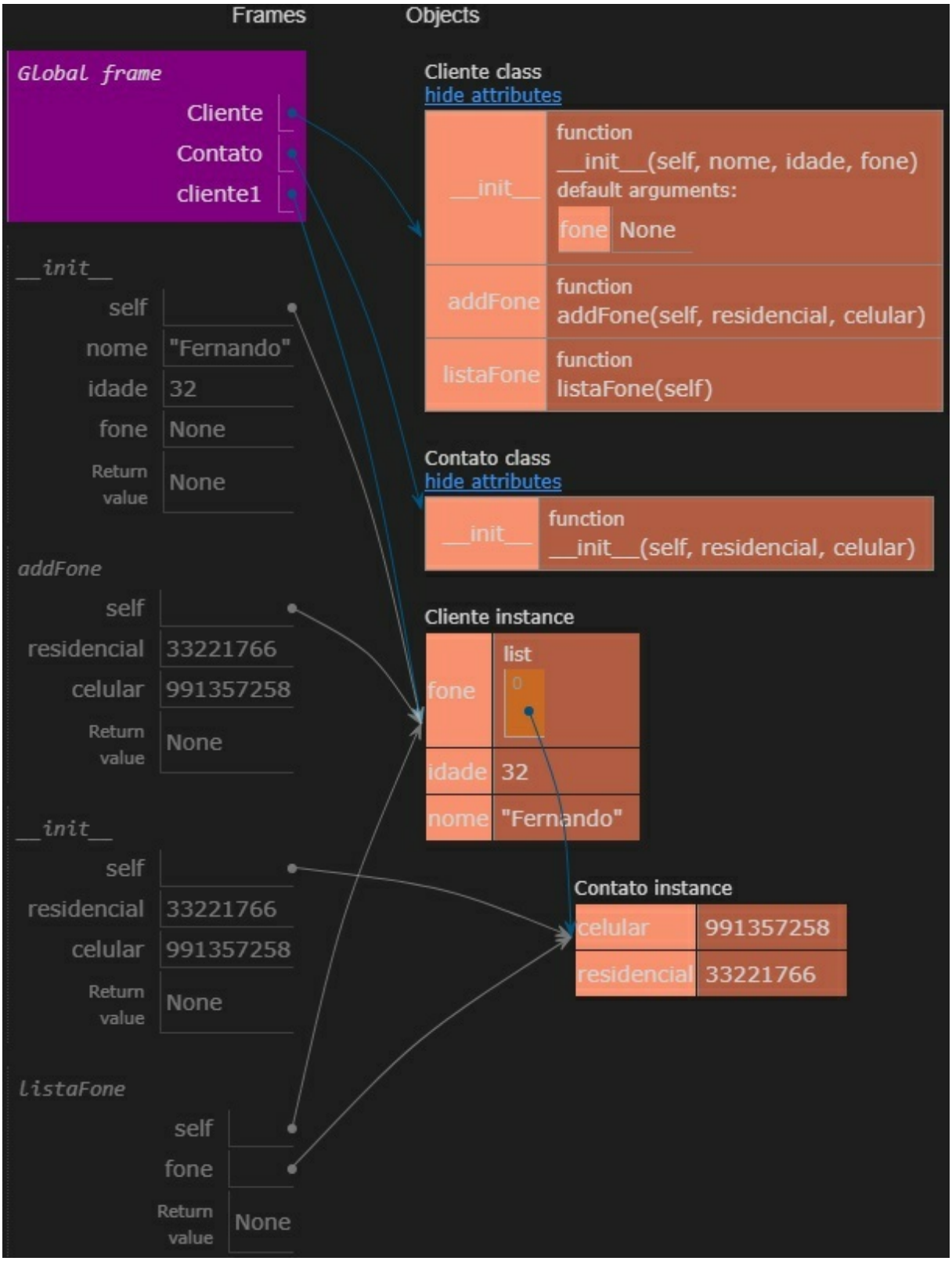
Seguindo com o exemplo, é criada uma variável de nome cliente1 que instancia Cliente passando como atributos de classe Fernando, 32. Em seguida é chamada a função addFone( ) passando como parâmetros 33221766 e 991357258.

Por fim, por meio do comando print( ) pedimos que sejam exibidos os dados de cliente1.nome de cliente1.listaFone( ).

Nesse caso o retorno será:

Fernando

33221766 991357258


Frames
Objects
Global frame
Cliente
Contato
cliente1
__init__
self
nome
"Fernando"
idade
32
fone
None
Return value
None
addFone
self
residencial
33221766
celular
991357258
Return value
None
__init__
self
residencial
33221766
celular
991357258
Return value
None
listaFone
self
fone
Return value
None
Cliente class
hide attributes
__init__
function
__init__(self, nome, idade, fone)
default arguments:
fone
None
addFone
function
addFone(self, residencial, celular)
listaFone
function
listaFone(self)
Contato class
hide attributes
__init__
function
__init__(self, residencial, celular)
Cliente instance
fone
list
0
idade
32
nome
"Fernando"
Contato instance
celular
991357258
residencial
33221766

Herança Simples

Na estrutura de código de um determinado programa todo orientado a objetos, é bastante comum que algumas classes em teoria possuam exatamente as mesmas características que outras, porém isso não é nada eficiente no que diz respeito a quantidade de linhas de código uma vez que esses blocos, já que são idênticos, estão repetidos no código. Por Exemplo:

```
1  class Corsa:
2      def __init__(self, nome, ano):
3          self.nome = nome
4          self.ano = ano
5
6  class Gol:
7      def __init__(self, nome, ano):
8          self.nome = nome
9          self.ano = ano
10
```

Dado o exemplo acima, note que temos duas classes para dois carros diferentes, Corsa e Gol, e repare que as características que irão identificar os mesmos, aqui nesse exemplo bastante básico, seria um nome e o ano do veículo. Duas classes com a mesma estrutura de código repetida duas vezes.

Podemos otimizar esse processo criando uma classe molde de onde serão lidas e interpretadas todas essas características de forma otimizada por nosso interpretador.

Vale lembrar que neste exemplo são apenas duas classes, não impactando na performance, mas numa aplicação real pode haver dezenas ou centenas de classes idênticas em estrutura, o que acarreta em lentidão se lidas uma a uma de forma léxica.

```python
class Carro:
    def __init__(self, nome, ano):
        self.nome = nome
        self.ano = ano

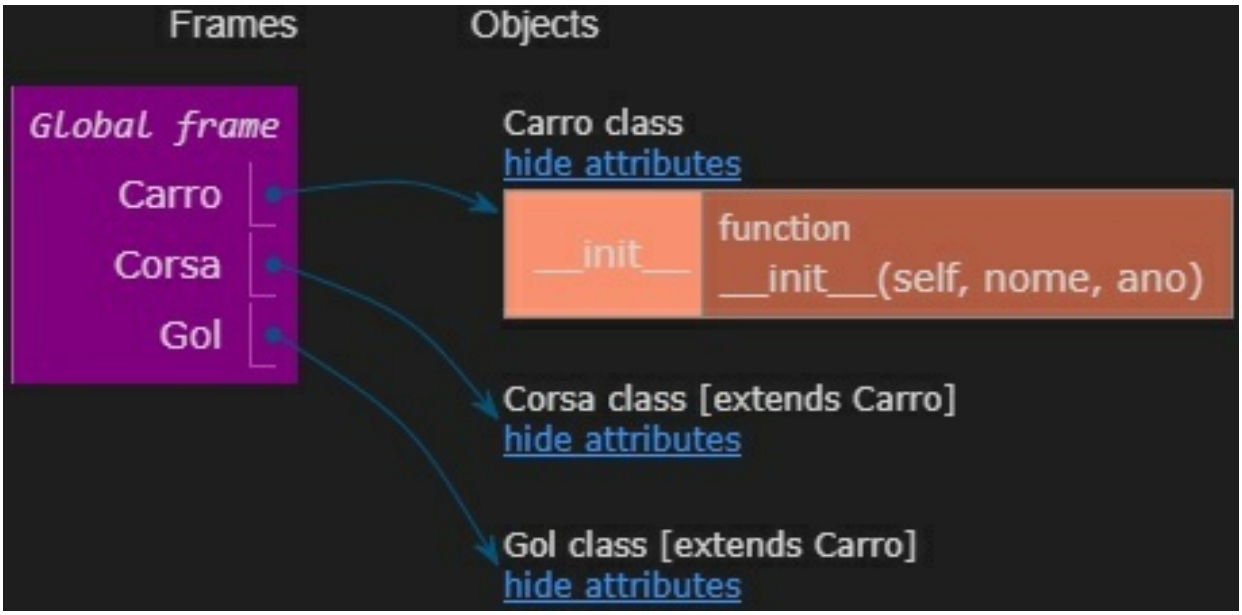
class Corsa(Carro):
    pass

class Gol(Carro):
    pass

```

Reformulando o código, é criada uma classe de nome Carro que servirá de molde para as outras, dentro de si ela tem um método construtor assim como recebe como atributos de classe um nome e um ano.

A partir de agora as classes Corsa e Gol simplesmente recebem como parâmetro a classe Carro, recebendo toda sua estrutura. Internamente o interpretador não fará distinção e assumirá que cada classe é uma classe normal, independente, inclusive com alocações de memória separadas para cada classe, porém a estrutura do código dessa forma é muito mais enxuta.



Na literatura, principalmente autores que escreveram por versões anteriores do Python costumavam chamar esta estrutura de classes e

subclasses, apenas por convenção, para que seja facilitada a visualização da hierarquia das mesmas.

Vale lembrar também que uma “subclasse” pode ter mais métodos e atributos particulares a si, sem problema nenhum, a herança ocorre normalmente a tudo o que estiver codificado na classe mãe, porém as classes filhas desta tem flexibilidade para ter mais atributos e funções conforme necessidade.

Ainda sob o exemplo anterior, a subclasse Corsa, por exemplo, poderia ser mais especializada tendo maiores atributos dentro de si além do nome e ano herdado de Carro.

Cadeia de heranças

Como visto no tópico anterior, uma classe pode herdar outra sem problema nenhum, desde que a lógica estrutural e sintática esteja correta na hora de definir as mesmas.

Extrapolando um pouco, vale salientar que essa herança pode ocorrer em diversos níveis, na verdade, não há um limite para isto, porém é preciso tomar bastante cuidado para não herdar características desnecessárias que possam tornar o código ineficiente. Ex:

```
1  class Carro:
2      def __init__(self, nome, ano):
3          self.nome = nome
4          self.ano = ano
5
6  class Gasolina(Carro):
7      def __init__(self, tipogasolina=True, tipoalcool=False):
8          self.tipogasolina = tipogasolina
9          self.tipoalcool = tipoalcool
10
11 class Jeep(Gasolina):
12     pass
13
```

Repare que nesse exemplo inicialmente é criada uma classe Carro, dentro de si ela possui um método construtor onde é repassado como atributo de classe um nome e um ano.

Em seguida é criada uma classe de nome Gasolina que herda toda estrutura de Carro e dentro de si define que o veículo desta categoria será do tipo gasolina.

Por fim é criada uma última classe se referindo a um carro marca Jeep, que herda toda estrutura de Gasolina e de Carro pela hierarquia entre classe e subclasses. Seria o mesmo que:

```
class Jeep:
    def carro():
        def __init__(self, nome, ano):
            self.nome = nome
            self.ano = ano

    def gasolina(self, tipogasolina=True, tipoalcool=False):
        self.tipogasolina = tipogasolina
        self.tipoalcool = tipoalcool

jeep = Jeep()

```

Código usual, porém, ineficiente. Assim como estático, de modo que quando houvesse a criação de outros veículos não seria possível reaproveitar nada de dentro dessa Classe a não ser por instâncias de fora da mesma, o que gera muitos caminhos de leitura para o interpretador, tornando a performance do código ruim.

Lembrando que a vantagem pontual que temos ao realizar heranças entre classes é a reutilização de seus componentes internos no lugar de blocos e mais blocos de código repetidos.

```
class Carro:
    def __init__(self, nome, ano):
        self.nome = nome
        self.ano = ano

class Gasolina(Carro):
    def __init__(self, tipogasolina=True, tipoalcool=False):
        self.tipogasolina = tipogasolina
        self.tipoalcool = tipoalcool

class Jeep(Gasolina):
    pass

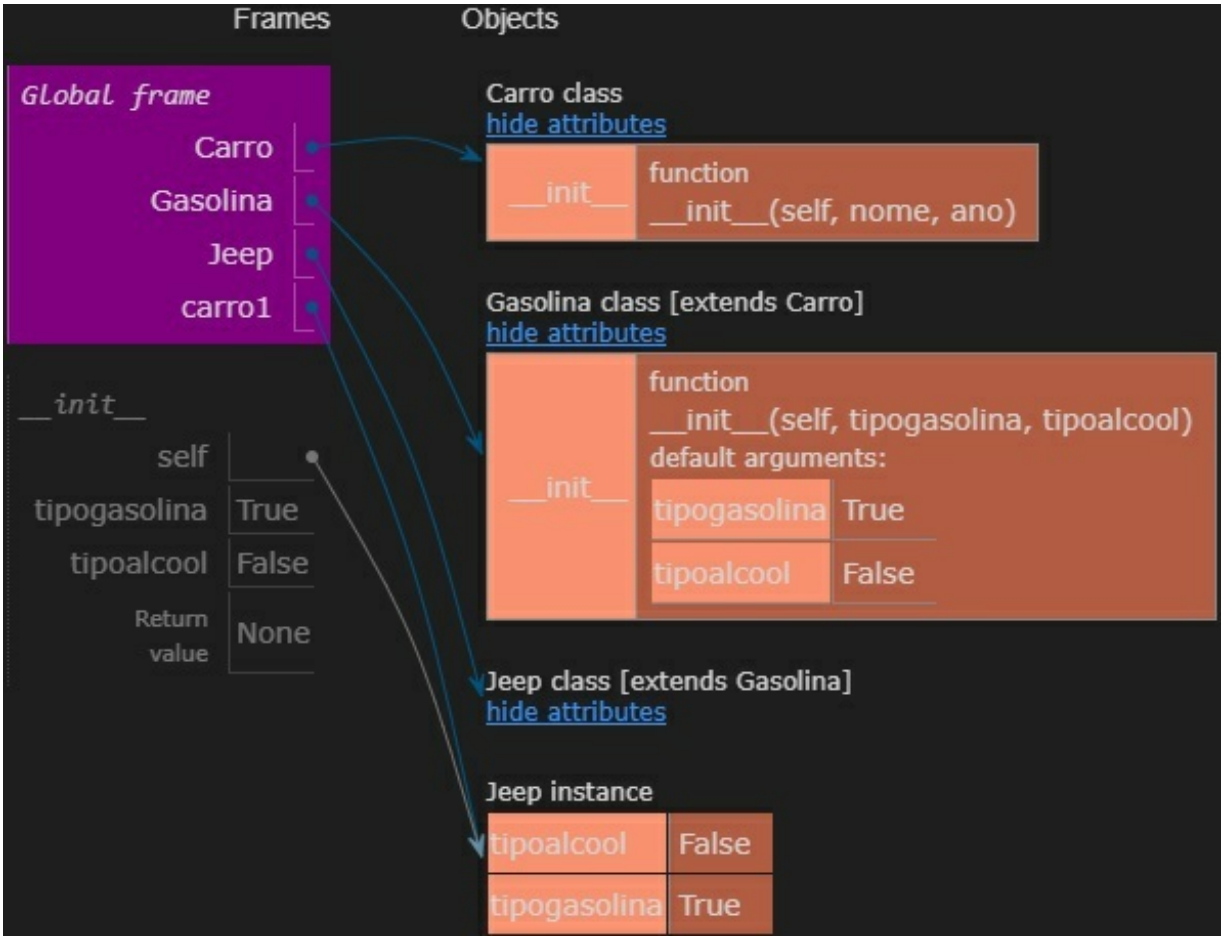
carro1 = Jeep()
print(carro1.tipogasolina)


```

```
True
```

Código otimizado via herança e cadeia de hierarquia simples.

Executando nossa função print( ) nesse caso o retorno será: True



Herança Múltipla

Entendida a lógica de como uma classe herda toda a estrutura de outra classe, podemos avançar um pouco mais fazendo o que é comumente chamado de herança múltipla. Raciocine que numa herança simples, criávamos uma determinada classe assim como todos seus métodos e atributos de classe, de forma que podíamos instanciar / passar ela como parâmetro de uma outra classe.

Respeitando a sintaxe, é possível realizar essa herança vinda de diferentes classes, importando dessa forma seus conteúdos. Por exemplo:

```
1  class Mercadoria:
2      def __init__(self, nome, preco):
3          self.nome = nome
4          self.preco = preco
5
6  class Carnes(Mercadoria):
7      def __init__(self, tipo, peso):
8          self.tipo = tipo
9          self.peso = peso
10
11 class Utensilios:
12     def __init__(self, espetos, carvao):
13         self.espetos = espetos
14         self.carvao = carvao
15
16 class KitChurrasco(Carnes, Utensilios):
17     pass
18
```

Para este exemplo, vamos supor que esse seria o background de um simples sistema de inventário de um mercado, onde as categorias de produtos possuem estrutura em classes distintas, porém que podem ser combinadas para montar “kits” de produtos.

Então repare que inicialmente é criada uma classe de nome

Mercadoria, que por sua vez possui um método construtor assim como nome e preco das mercadorias como atributos de classe.

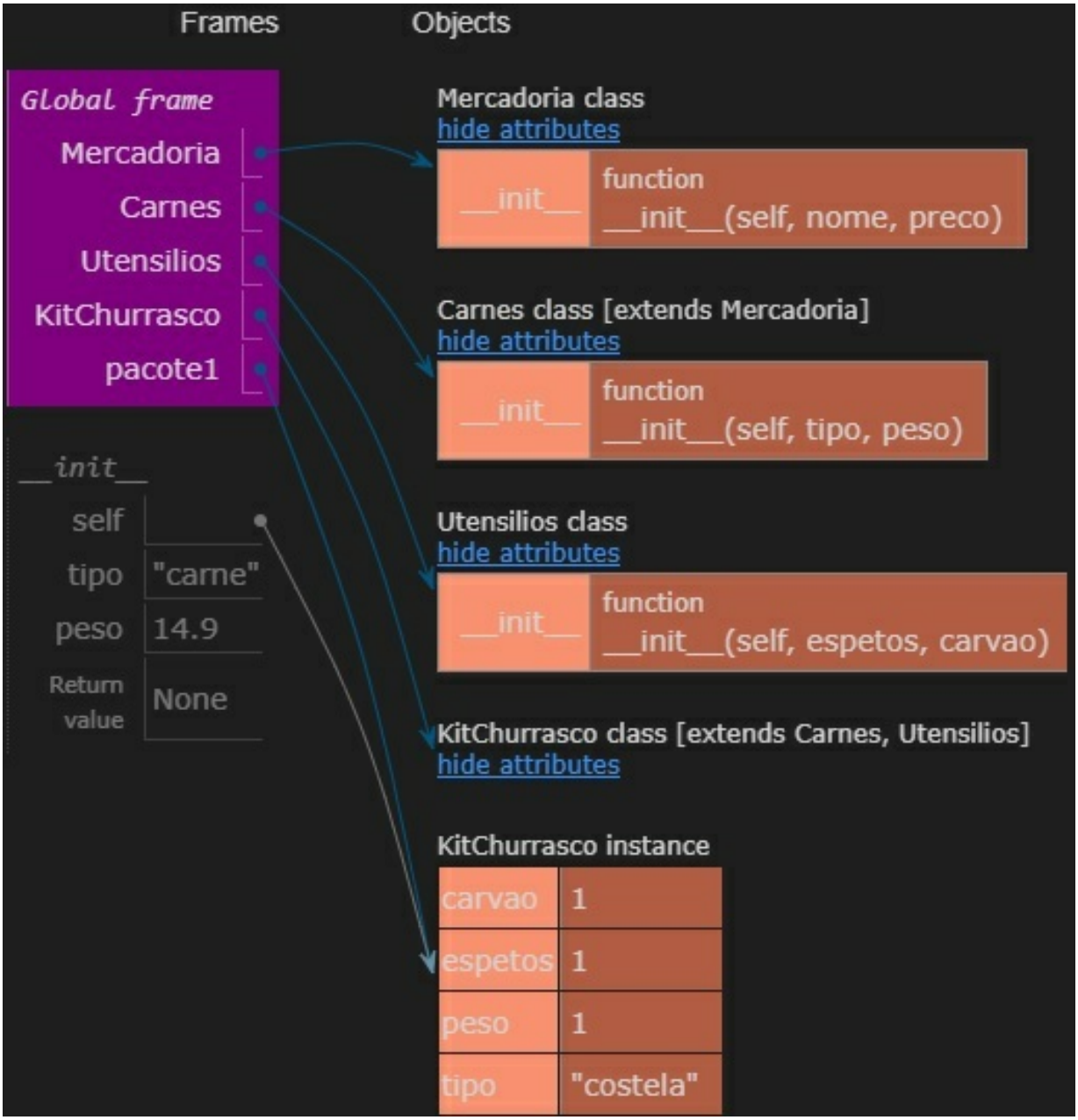
Em seguida é criada uma classe de nome Carnes que herda toda estrutura de Mercadoria e adiciona os atributos de classe referentes ao tipo de carne e seu peso.

Na sequência é criada uma classe de nome Utensilios que não herda nada das classes anteriores, e dentro de si possui estrutura de método construtor assim como seus atributos de classe.

Por fim, é criada uma classe de nome KitChurrasco, que por sua vez herda Carnes (que herda Mercadoria) e Utensilios com todo seu conteúdo.

```
1  class Mercadoria:
2      def __init__(self, nome, preco):
3          self.nome = nome
4          self.preco = preco
5
6  class Carnes(Mercadoria):
7      def __init__(self, tipo, peso):
8          self.tipo = tipo
9          self.peso = peso
10
11 class Utensilios:
12     def __init__(self, espetos, carvao):
13         self.espetos = espetos
14         self.carvao = carvao
15
16 class KitChurrasco(Carnes, Utensilios):
17     pass
18
19 pacote1 = KitChurrasco('carne', 14.90)
20 pacote1.tipo = 'costela'
21 pacote1.peso = 1
22 pacote1.espetos = 1
23 pacote1.carvao = 1
24
```

A partir disso, como nos exemplos anteriores, é possível instanciar objetos e definir dados/valores para os mesmos normalmente, uma vez que KitChurrasco agora, devido as características que herdou, possui campos para nome, preco, tipo, peso, espetos e carvao dentro de si.



Sobreposição de membros

Um dos cuidados que devemos ter ao manipular nossas classes em suas hierarquias e heranças é o fato de apenas realizarmos sobreposições quando de fato queremos alterar/sobrescrever um determinado dado/valor/método dentro da mesma.

Em outras palavras, anteriormente vimos que o interpretador do Python realiza a chamada leitura léxica do código, ou seja, ele lê linha por linha (de cima para baixo) e linha por linha (da esquerda para direita).

Dessa forma, podemos reprogramar alguma linha ou bloco de código para que na sequência de sua leitura/interpretação o código seja alterado. Por exemplo:

```
1  class Pessoa:
2      def __init__(self, nome):
3          self.nome = nome
4
5      def Acao1(self):
6          print(f'{self.nome} está dormindo')
7
8  class Jogador1(Pessoa):
9      def Acao2(self):
10         print(f'{self.nome} está comendo')
11
12 class SaveJogador1(Jogador1):
13     pass
14
```

Inicialmente criamos a classe Pessoa, dentro de si um método construtor que recebe um nome como atributo de classe. Também é definida uma Acao1 que por sua vez por meio da função print( ) exibe uma mensagem.

Em seguida é criada uma classe de nome Jogador1 que herda tudo de Pessoa e por sua vez, apenas tem um método Acao2 que exibe também

uma determinada mensagem. Por fim é criado uma classe SaveJogador1 que herda tudo de Jogador1 e de Pessoa.

```
class Pessoa:
    def __init__(self, nome):
        self.nome = nome

    def Acao1(self):
        print(f'{self.nome} está dormindo')

class Jogador1(Pessoa):
    def Acao2(self):
        print(f'{self.nome} está comendo')

class SaveJogador1(Jogador1):
    pass

p1 = SaveJogador1('Fernando')
print(p1.nome)

```

```
Fernando
```

Trabalhando sobre essa cadeia de classes, criamos uma variável de nome p1 que instancia SaveJogador1 e passa como parâmetro ‘Fernando’, pela hierarquia destas classes, esse parâmetro alimentará self.nome de Pessoa.

Por meio da função print( ) passando como parâmetros p1.nome o retorno é: Fernando

```python
class Pessoa:
    def __init__(self, nome):
        self.nome = nome

    def Acao1(self):
        print(f'{self.nome} está dormindo')

class Jogador1(Pessoa):
    def Acao2(self):
        print(f'{self.nome} está comendo')

class SaveJogador1(Jogador1):
    pass

p1 = SaveJogador1('Fernando')
print(p1.nome)

p1.Acao1()
p1.Acao2()

```

```
Fernando
Fernando está dormindo
Fernando está comendo
```

Da mesma forma, instanciando qualquer coisa de dentro de qualquer classe da hierarquia, se não houver nenhum erro de sintaxe tudo será executado normalmente.

Neste caso o retorno será:

Fernando está dormindo

Fernando está comendo

```
1  class Pessoa:
2      def __init__(self, nome):
3          self.nome = nome
4
5      def Acao1(self):
6          print(f'{self.nome} está dormindo')
7
8  class Jogador1(Pessoa):
9      def Acao2(self):
10         print(f'{self.nome} está comendo')
11
12 class SaveJogador1(Jogador1):
13     def Acao1(self):
14         print(f'{self.nome} está acordado')
15
```

Por fim, partindo para uma sobreposição de fato, note que agora em SaveJogador1 criamos um método de classe de nome Acao1, dentro de si uma função para exibir uma determinada mensagem.

Repare que Acao1 já existia em Pessoa, mas o que ocorre aqui é que pela cadeia de heranças SaveJogador1 criando um método Acao1 irá sobrepor esse método já criado anteriormente.

Em outras palavras, dada a hierarquia entre essas classes, o interpretador irá considerar pela sua leitura léxica a última alteração das mesmas, Acao1 presente em SaveJogador1 é a última modificação desse método de classe, logo, a função interna do mesmo irá ser executada no lugar de Acao1 de Pessoa, que será ignorada.

```python
class Pessoa:
    def __init__(self, nome):
        self.nome = nome

    def Acao1(self):
        print(f'{self.nome} está dormindo')

class Jogador1(Pessoa):
    def Acao2(self):
        print(f'{self.nome} está comendo')

class SaveJogador1(Jogador1):
    def Acao1(self):
        print(f'{self.nome} está acordado')

p1 = SaveJogador1('Fernando')
p1.Acao1()
p1.Acao2()

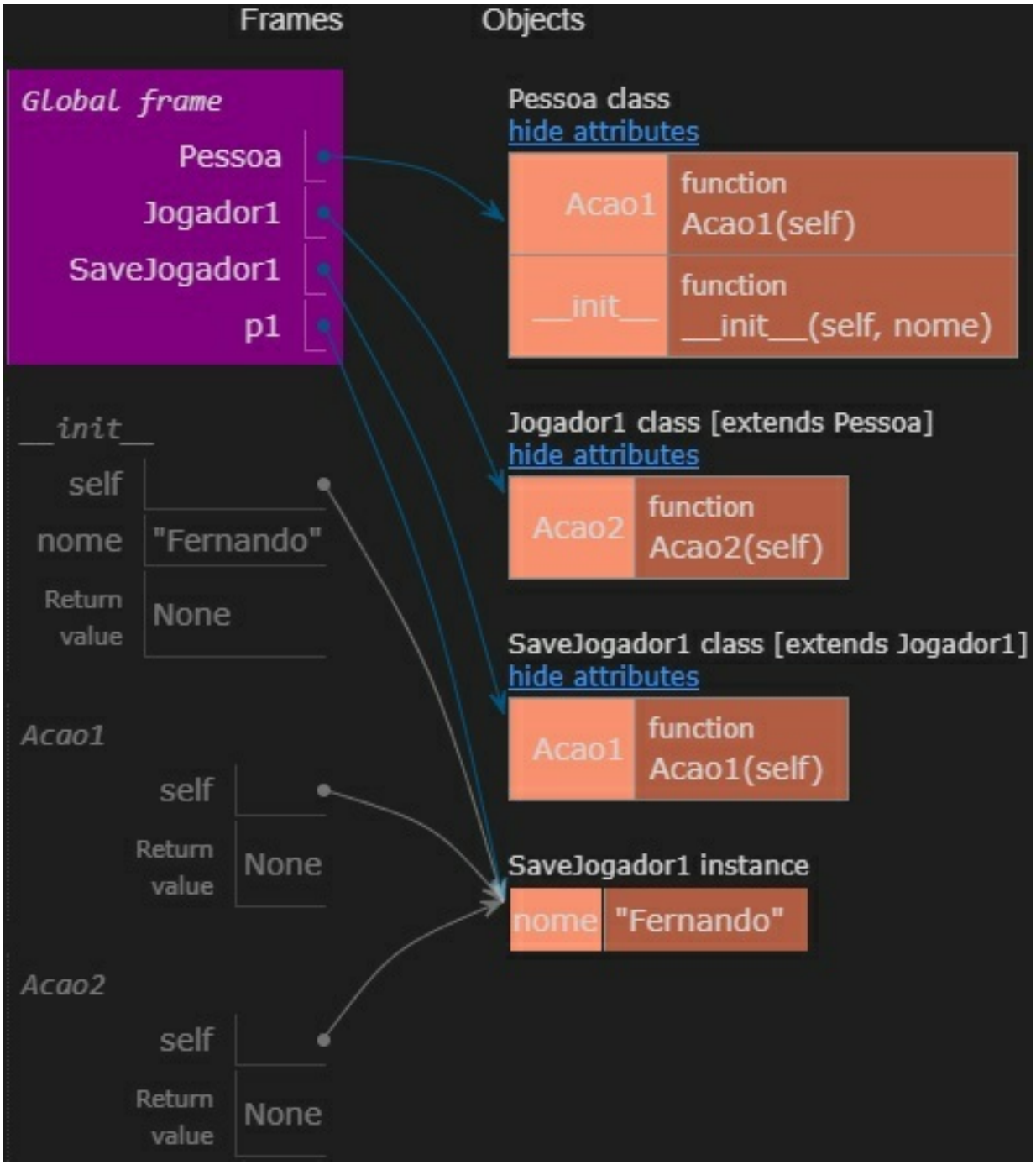
```

```
Fernando está acordado
Fernando está comendo
```

Nesse caso o retorno será:

Fernando está acordado (a última instrução atribuída a Acao1)

Fernando está comendo



Outra funcionalidade que eventualmente pode ser utilizada é a de, justamente executar uma determinada ação da classe mãe e posteriormente a sobrescrever, isso é possível por meio da função reservada super( ).

Esta função em orientação a objetos faz com que seja respeitada a hierarquia de classe mãe / super classe em primeiro lugar e posteriormente as

classes filhas / sub classes.

```python
class Pessoa:
    def __init__(self, nome):
        self.nome = nome

    def Acao1(self):
        print(f'{self.nome} está dormindo')

class Jogador1(Pessoa):
    def Acao2(self):
        print(f'{self.nome} está comendo')

class SaveJogador1(Jogador1):
    def Acao1(self):
        super().Acao1()
        print(f'{self.nome} está acordado')

p1 = SaveJogador1('Fernando')
p1.Acao1()
p1.Acao2()
class SaveJogador1(Jogador1):
    def Acao1(self):
        super().Acao1()
        print(f'{self.nome} está acordado')

```

```
Fernando está dormindo
Fernando está acordado
Fernando está comendo
```

Neste caso, primeiro será executada a Acao1 de Pessoa e em seguida ela será sobrescrita por Acao1 de SaveJogador1.

O retorno será:

Fernando está dormindo (Acao1 de Pessoa)

Fernando está acordado (Acao1 de SaveJogador1)

Fernando está comendo

Avançando com nossa linha de raciocínio, ao se trabalhar com heranças deve-se tomar muito cuidado com as hierarquias entre as classes dentro de sua ordem de leitura léxica.

Raciocine que o mesmo padrão usado no exemplo anterior vale para qualquer coisa, até mesmo um método construtor pode ser sobrescrito, logo, devemos tomar os devidos cuidados na hora de usar os mesmos.

Também é importante lembrar que por parte de sobreposições, quando se trata de heranças múltiplas, a ordem como as classes são passadas como parâmetro irá influenciar nos resultados. Ex:

```
class Pessoa:
    def acao(self):
        print('Inicializando o sistema')

class Acao1(Pessoa):
    def acao(self):
        print('Sistema pronto para uso')

class Acao2(Pessoa):
    def acao(self):
        print('Desligando o sistema')

class SaveJogador1(Acao1, Acao2):
    pass

p1 = SaveJogador1()
p1.acao()

```

```
Sistema pronto para uso
```

Apenas como exemplo, repare que foram criadas 4 classes, Pessoa, Acao1, Acao2 e SaveJogador1 respectivamente, dentro de si apenas existe um método de classe designado a exibir uma mensagem.

Para esse exemplo, o que importa é a forma como SaveJogador1 está herdando características de Acao1 e Acao2, como dito anteriormente, essa ordem ao qual as classes são passadas como parâmetro aqui influenciará os resultados da execução do código.

Nesse exemplo, SaveJogador1(Acao1, Acao2), o retorno será: Sistema pronto para uso

```
class Pessoa:
    def acao(self):
        print('Inicializando o sistema')

class Acao1(Pessoa):
    def acao(self):
        print('Sistema pronto para uso')

class Acao2(Pessoa):
    def acao(self):
        print('Desligando o sistema')

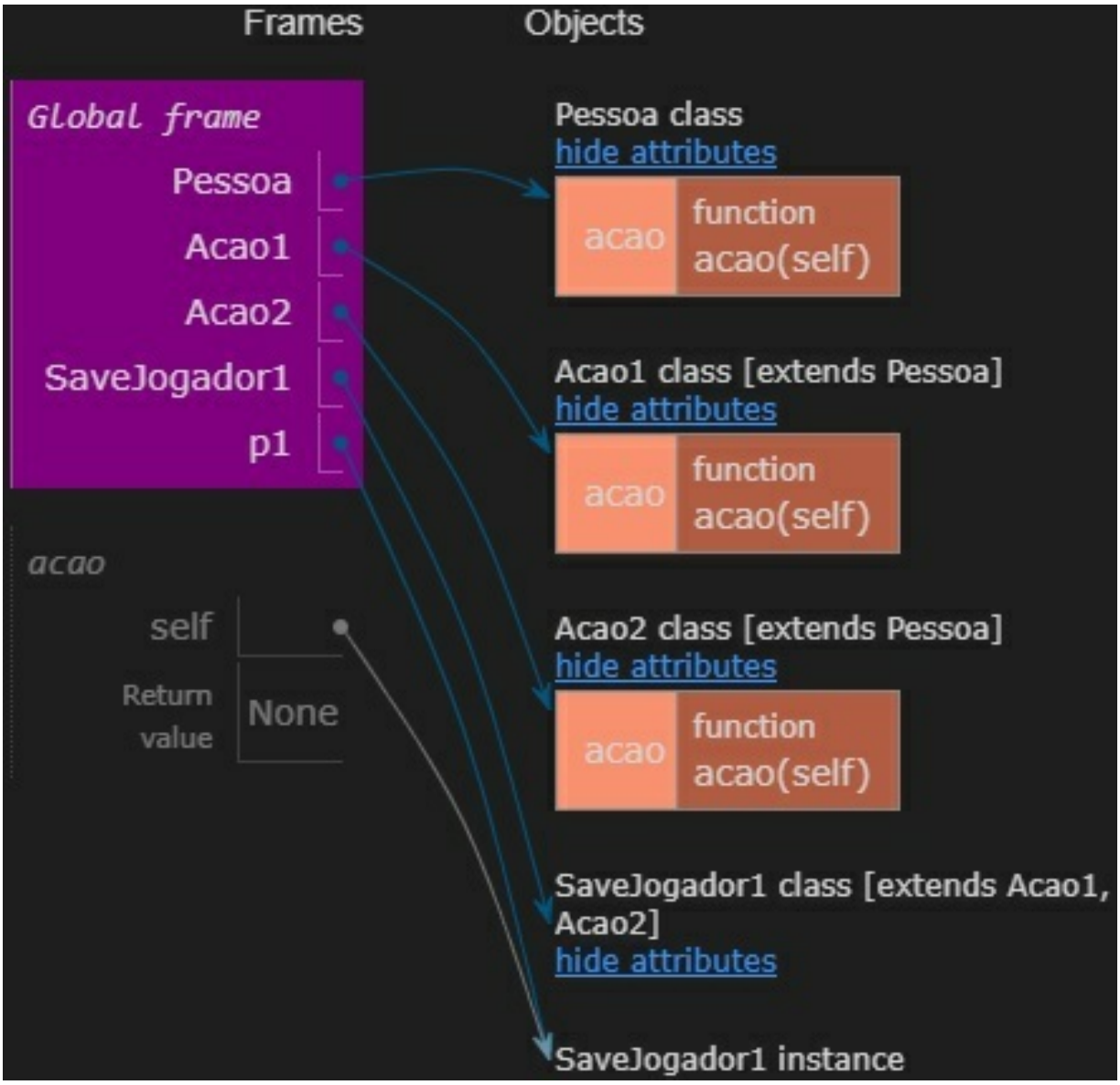
class SaveJogador1(Acao2, Acao1):
    pass


p1 = SaveJogador1()
p1.acao()

```

```
Desligando o sistema
```

Exatamente a mesma estrutura anterior, porém com ordem inversa, SaveJogador1(Acao2, Acao1), o retorno será: Desligando o sistema



Apenas fazendo um adendo, caso estes tópicos ainda gerem alguma dúvida para você. Uma forma de diferenciarmos a forma como tratamos as interações entre classes dentro do Python é a seguinte:

Associação: Se dá quando uma classe usa outro ou algum objeto de outra.

Agregação: Se dá quando um ou mais objetos de classe é compartilhado, usado por duas ou mais classes (para que não seja necessário programar esse atributo para cada uma delas).

Composição: Se dá quando uma classe é dona de outra pela interligação de seus atributos e a forma sequencial como uns dependem dos outros.

Herança: Se dá quando estruturalmente um objeto é outro objeto, no sentido de que ele literalmente herda todas características e funcionalidades do outro, para que o código fique mais enxuto.

Classes abstratas

Se você chegou até este tópico do livro certamente não terá nenhum problema para entender este modelo de classe, uma vez que já o utilizamos implicitamente em outros exemplos anteriores.

O fato é que, como dito desde o início, para alguns autores uma classe seria um molde a ser utilizado para se guardar todo e qualquer tipo de dado associado a um objeto de forma que possa ser utilizado livremente de forma eficiente ao longo do código.

Sendo assim, quando criamos uma classe vazia, ou bastante básica, que serve como base estrutural para outras classes, já estamos trabalhando com uma classe abstrata.

O que faremos agora é definir manualmente este tipo de classe, de forma que sua forma estrutural irá forçar as classes subsequentes a criar suas especializações a partir dela. Pode parecer um tanto confuso, mas com o exemplo tudo ficará mais claro.

Como dito anteriormente, na verdade já trabalhamos com este tipo de classe, mas de forma implícita e assim o interpretador do Python não gerava nenhuma barreira quanto a execução de uma classe quando a mesma era abstrata.

Agora, usaremos para este exemplo, um módulo do Python que nos obrigará a abstrair manualmente as classes a partir de uma sintaxe própria.

Sendo assim, inicialmente precisamos importar os módulos ABC e abstractmethod da biblioteca abc.

```
1  from abc import ABC, abstractclassmethod
2
```

Realizadas as devidas importações, podemos prosseguir com o exemplo:

```python
from abc import ABC, abstractclassmethod

class Pessoa(ABC):
    @abstractclassmethod
    def logar(self):
        pass

class Usuario(Pessoa):
    def logar(self):
        print('Usuario logado no sistema')

```

Inicialmente criamos uma classe Pessoa, que já herda ABC em sua declaração, dentro de si, note que é criado um decorador @abstractclassmethod, que por sua vez irá sinalizar ao interpretador que todos os blocos de código que vierem na sequência dessa classe, devem ser sobrescritas em suas respectivas classes filhas dessa hierarquia.

Em outras palavras, Pessoa no momento dessa declaração, possui um método chamado logar que obrigatoriamente deverá ser criado em uma nova classe herdeira, para que possa ser instanciada e realizar suas devidas funções.

Dando sequência, repare que em seguida criamos uma classe Usuario, que herda Pessoa, e dentro de si cria o método logar para sobrepor/sobrescrever o método logar de Pessoa. Este, por sua vez, exibe uma mensagem pré-definida por meio da função print( ).

```python
from abc import ABC, abstractclassmethod

class Pessoa(ABC):
    @abstractclassmethod
    def logar(self):
        pass

class Usuario(Pessoa):
    def logar(self):
        print('Usuario logado no sistema')

user1 = Pessoa()
user1.logar()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call l
<ipython-input-85-10a39ece0069> in <module>()
     10         print('Usuario logado no sistema')
     11
---> 12 user1 = Pessoa()
     13 user1.logar()

TypeError: Can't instantiate abstract class Pessoa with abstract method
```

A partir do momento que Pessoa é identificada como uma classe abstrata por nosso decorador, a mesma passa a ser literalmente apenas um molde.

Seguindo com o exemplo, tentando fazer a associação que estamos acostumados a fazer, atribuindo essa classe a um objeto qualquer, ao tentar executar a mesma será gerado um erro de interpretação.

Mensagem de erro, em tradução livre: Não é possível inicializar a classe abstrata Pessoa com o método abstrato logar.

```python
from abc import ABC, abstractclassmethod

class Pessoa(ABC):
    @abstractclassmethod
    def logar(self):
        pass

class Usuario(Pessoa):
    def logar(self):
        print('Usuario logado no sistema')

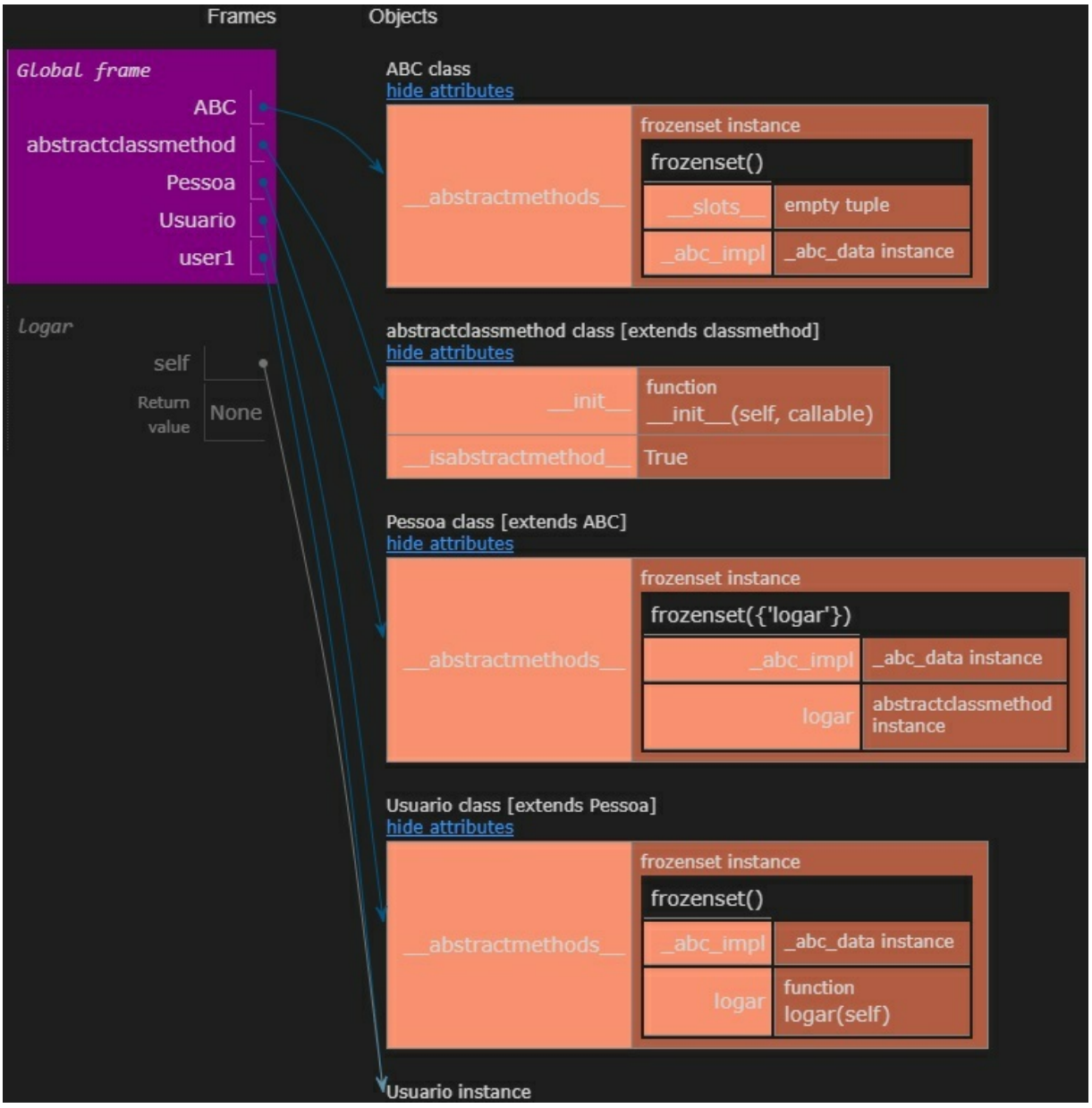
user1 = Usuario()
user1.logar()

```

```
Usuario logado no sistema
```

Porém instanciando Usuario, que herda Pessoa, ao tentar executar o método de classe logar que consta no corpo do mesmo, este funciona perfeitamente.

Nesse caso o retorno será: Usuario logado no sistema



Sendo assim, o que precisa ficar bem claro, inicialmente, é que uma das formas que temos de proteger uma classe que apenas nos serve de molde, para que essa seja herdada, mas não sobrescrita, é definindo a mesma manualmente como uma classe abstrata.

Polimorfismo

Avançando mais um pouco com nossos estudos, hora de abordar um princípio da programação orientada a objetos chamada polimorfismo.

Como o próprio nome já sugere, polimorfismo significa que algo possui muitas formas (ou ao menos mais que duas) em sua estrutura. Raciocine que é perfeitamente possível termos classes derivadas de uma determinada classe mãe / super classe que possuem métodos iguais, porém comportamentos diferentes dependendo sua aplicação no código.

Alguns autores costumam chamar esta característica de classe de “assinatura”, quando se refere a mesma quantidade e tipos de parâmetros, porém com fins diferentes.

Último ponto a destacar antes de partirmos para o código é que, polimorfismo normalmente está fortemente ligado a classes abstratas, sendo assim, é de suma importância que a lógica deste tópico visto no capítulo anterior esteja bem clara e entendida, caso contrário poderá haver confusão na hora de identificar tais estruturas de código para criar suas devidas associações.

Entenda que uma classe abstrata, em polimorfismo, obrigatoriamente será um molde para criação das demais classes onde será obrigatório realizar a sobreposição do(s) método(s) de classe que a classe mãe possuir.

```python
from abc import ABC, abstractclassmethod

class Pessoa(ABC):
    @abstractclassmethod
    def logar(self, chaveseguranca):
        pass

class Usuario(Pessoa):
    def logar(self, chaveseguranca):
        print('Usuario logado no sistema')

class Bot(Pessoa):
    def logar(self, chaveseguranca):
        print('Sistema rodando em segundo plano')

```

Seguindo com um exemplo muito próximo ao anterior, apenas para melhor identificação, note que inicialmente são feitas as importações dos módulos ABC e abstractclassmethod da biblioteca abc.

Em seguida é criada uma classe de nome Pessoa que herda ABC, dentro de si existe um decorador e um método de classe que a caracteriza como uma classe abstrata.

Em outras palavras, toda classe que herdar a estrutura de Pessoa terá de redefinir o método logar, fornecendo uma chavedeseguranca que aqui não está associada a nada, mas na prática seria parte de um sistema de login.

Na sequência é criada a classe Usuário que herda Pessoa, e pela sua estrutura, repare que ela é polimórfica, possuindo a mesma assinatura de objetos instanciados ao método logar, porém há uma função personalizada que está programada para exibir uma mensagem.

O mesmo é feito com uma terceira classe chamada Bot que herda Pessoa, que por sua vez herda ABC, tudo sob a mesma “assinatura”.

```python
from abc import ABC, abstractclassmethod

class Pessoa(ABC):
    @abstractclassmethod
    def logar(self, chaveseguranca):
        pass

class Usuario(Pessoa):
    def logar(self, chaveseguranca):
        print('Usuario logado no sistema')

class Bot(Pessoa):
    def logar(self, chaveseguranca):
        print('Sistema rodando em segundo plano')

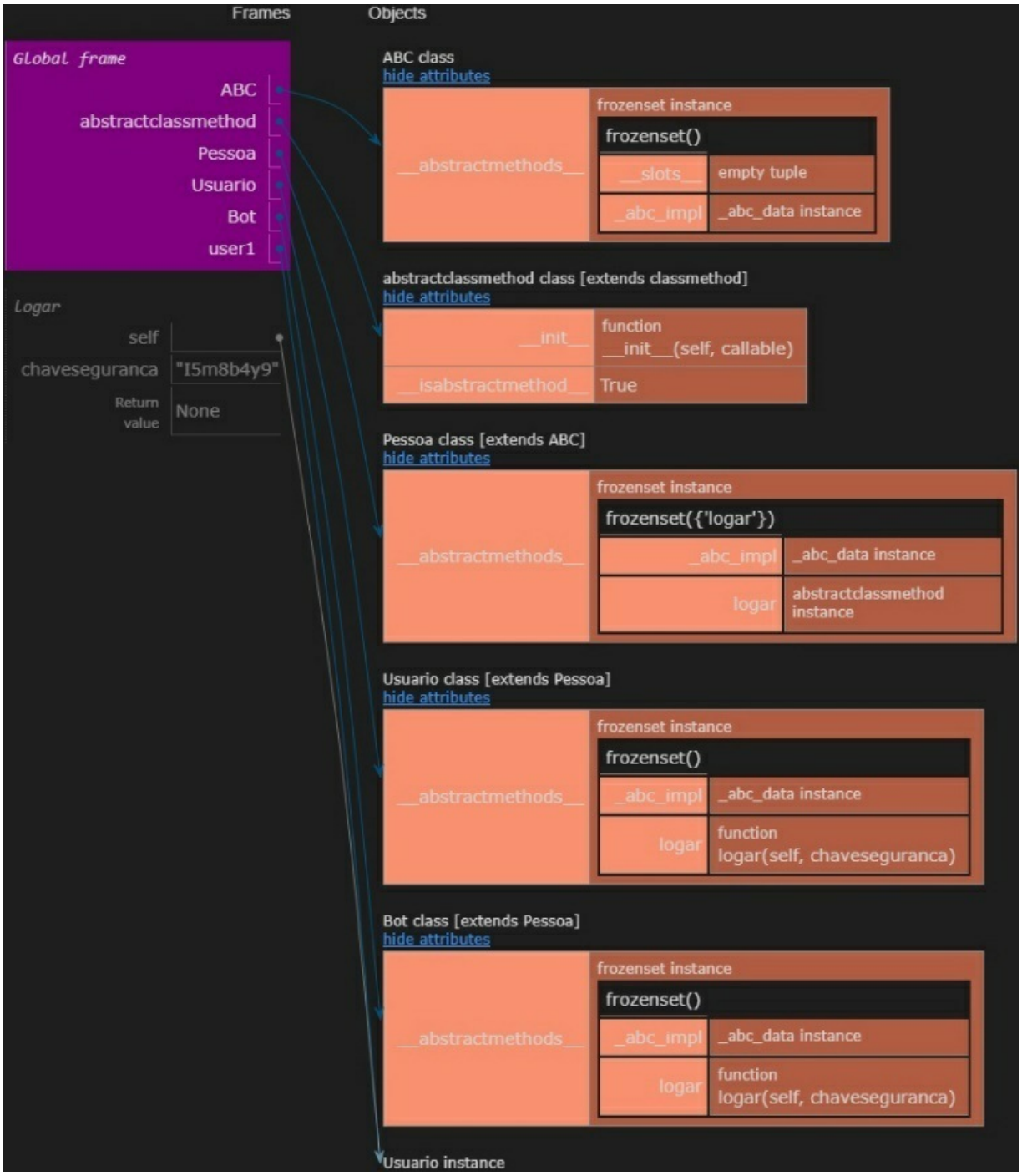
user1 = Usuario()
user1.logar('I5m8b4y9')

```

```
Usuario logado no sistema
```

Instanciando Usuario a um objeto qualquer e aplicando sobre si parâmetros (neste caso, fornecendo uma senha para chaveseguranca), ocorrerá a execução da respectiva função.

Neste caso o retorno será: Usuario logado no sistema


Frames
Objects
Global frame
ABC
abstractclassmethod
Pessoa
Usuario
Bot
user1
Logar
self
chaveseguranca
"I5m8b4y9"
Return value
None
ABC class
hide attributes
__abstractmethods__
frozenset instance
frozenset()
__slots__
empty tuple
_abc_impl
_abc_data instance
abstractclassmethod class [extends classmethod]
hide attributes
__init__
function __init__(self, callable)
__isabstractmethod__
True
Pessoa class [extends ABC]
hide attributes
__abstractmethods__
frozenset instance
frozenset({'logar'})
_abc_impl
_abc_data instance
logar
abstractclassmethod instance
Usuario class [extends Pessoa]
hide attributes
__abstractmethods__
frozenset instance
frozenset()
_abc_impl
_abc_data instance
logar
function logar(self, chaveseguranca)
Bot class [extends Pessoa]
hide attributes
__abstractmethods__
frozenset instance
frozenset()
_abc_impl
_abc_data instance
logar
function logar(self, chaveseguranca)
Usuario instance

Sobrecarga de operadores

Quando damos nossos primeiros passos no aprendizado do Python, em sua forma básica, programando de forma estruturada, um dos primeiros tópicos que nos é apresentado são os operadores.

Estes por sua vez são símbolos reservados ao sistema que servem para fazer operações lógicas ou aritméticas, de forma que não precisamos programar do zero tais ações, pela sintaxe, basta usar o operador com os tipos de dados aos quais queremos a interação e o interpretador fará a leitura dos mesmos normalmente.

Em outras palavras, se declaramos duas variáveis com números do tipo inteiro, podemos por exemplo, via operador de soma + realizar a soma destes valores, sem a necessidade de programar do zero o que seria uma função que soma dois valores.

Se tratando da programação orientada a objetos, o que ocorre é que quando criamos uma classe com variáveis/objetos dentro dela, estes dados passam a ser um tipo de dado novo, independente, apenas reconhecido pelo interpretador como um objeto.

Para ficar mais claro, raciocine que uma variável normal, com valor atribuído de 4 por exemplo, é automaticamente interpretado por nosso interpretador como int (número inteiro, sem casas decimais), porém a mesma variável, exatamente igual, mas declarada dentro de uma classe, passa a ser apenas um objeto geral para o interpretador.

Números inteiros podem ser somados, subtraídos, multiplicados, elevados a uma determinada potência, etc... um objeto dentro de uma classe não possui essas funcionalidades a não ser que manualmente realizemos a chamada sobrecarga de operadores. Por exemplo:

```
1  class Caixa:
2      def __init__(self, largura, altura):
3          self.largura = largura
4          self.altura = altura
5
6  caixa1 = Caixa(10,10)
7  caixa2 = Caixa(10,20)
8
```

Aqui propositalmente simulando este erro lógico, inicialmente criamos uma classe de nome Caixa, dentro dela um método construtor que recebe um valor para largura e um para altura, atribuindo esses valores a objetos internos.

No corpo de nosso código, criando variáveis que instanciam nossa classe Caixa e atribuem valores para largura e altura, seria normal raciocinar que seria possível, por exemplo, realizar a soma destes valores.

```
 1  class Caixa:
 2      def __init__(self, largura, altura):
 3          self.largura = largura
 4          self.altura = altura
 5
 6  caixa1 = Caixa(10,10)
 7  caixa2 = Caixa(10,20)
 8
 9  print(caixa1 + caixa2)
10
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call
<ipython-input-90-b5b182d8e6c6> in <module>()
      7 caixa2 = Caixa(10,20)
      8
----> 9 print(caixa1 + caixa2)

TypeError: unsupported operand type(s) for +: 'Caixa' and 'Caixa'
```

Porém o que ocorre ao tentar realizar uma simples operação de soma entre tais valores o retorno que temos é um traceback.

Em tradução livre: Erro de Tipo: tipo de operador não suportado para +: Caixa e Caixa.

Repare que o erro em si é que o operador de soma não foi reconhecido como tal, pois esse símbolo de + está fora de contexto uma vez que estamos trabalhando com orientação a objetos.

Sendo assim, o que teremos de fazer é simplesmente buscar os operadores que sejam reconhecidos e processados neste tipo de programação.

Segue uma lista com os operadores mais utilizados, assim como o seu método correspondente para uso em programação orientada a objetos.

Operador	Método	Operação
+	__add__	Adição
-	__sub__	Subtração
*	__mul__	Multiplicação
/	__div__	Divisão
//	__floordiv__	Divisão inteira
%	__mod__	Módulo
**	__pow__	Potência
<	__lt__	Menor que
>	__gt__	Maior que
<=	__le__	Menor ou igual a
>=	__ge__	Maior ou igual a
==	__eq__	Igual a
!=	__ne__	Diferente de

Dando continuidade a nosso entendimento, uma vez que descobrimos que para cada tipo de operador lógico/aritmético temos um

método de classe correspondente, tudo o que teremos de fazer é de fato criar este método dentro de nossa classe, para forçar o interpretador a reconhecer que, nesse exemplo, é possível realizar a soma dos dados/valores atribuídos aos objetos ali instanciados.

```
1  class Caixa:
2      def __init__(self, largura, altura):
3          self.largura = largura
4          self.altura = altura
5
6      def __add__(self, other):
7          pass
8
```

Retornando ao código, note que simplesmente é criado um método de classe de nome __add__ que recebe como atributos um valor para si e um valor a ser somado neste caso. __add__ é uma palavra reservada ao sistema, logo, o interpretador sabe sua função e a executará normalmente dentro desse bloco de código.

```
 1  class Caixa:
 2      def __init__(self, largura, altura):
 3          self.largura = largura
 4          self.altura = altura
 5
 6      def __add__(self, other):
 7          largura1 = self.largura + other.largura
 8          altura1 = self.altura + other.altura
 9          return Caixa(largura1, altura1)
10
11      def __repr__(self):
12          return f"<class 'Caixa({self.largura}, {self.altura})'>"
13
```

Dessa forma, podemos criar normalmente uma função de soma personalizada, note que no método __add__ foi criada uma variável de nome largura1 que guardará o valor da soma entre as larguras dos objetos.

Da mesma forma, altura1 receberá o valor da soma das alturas, tudo

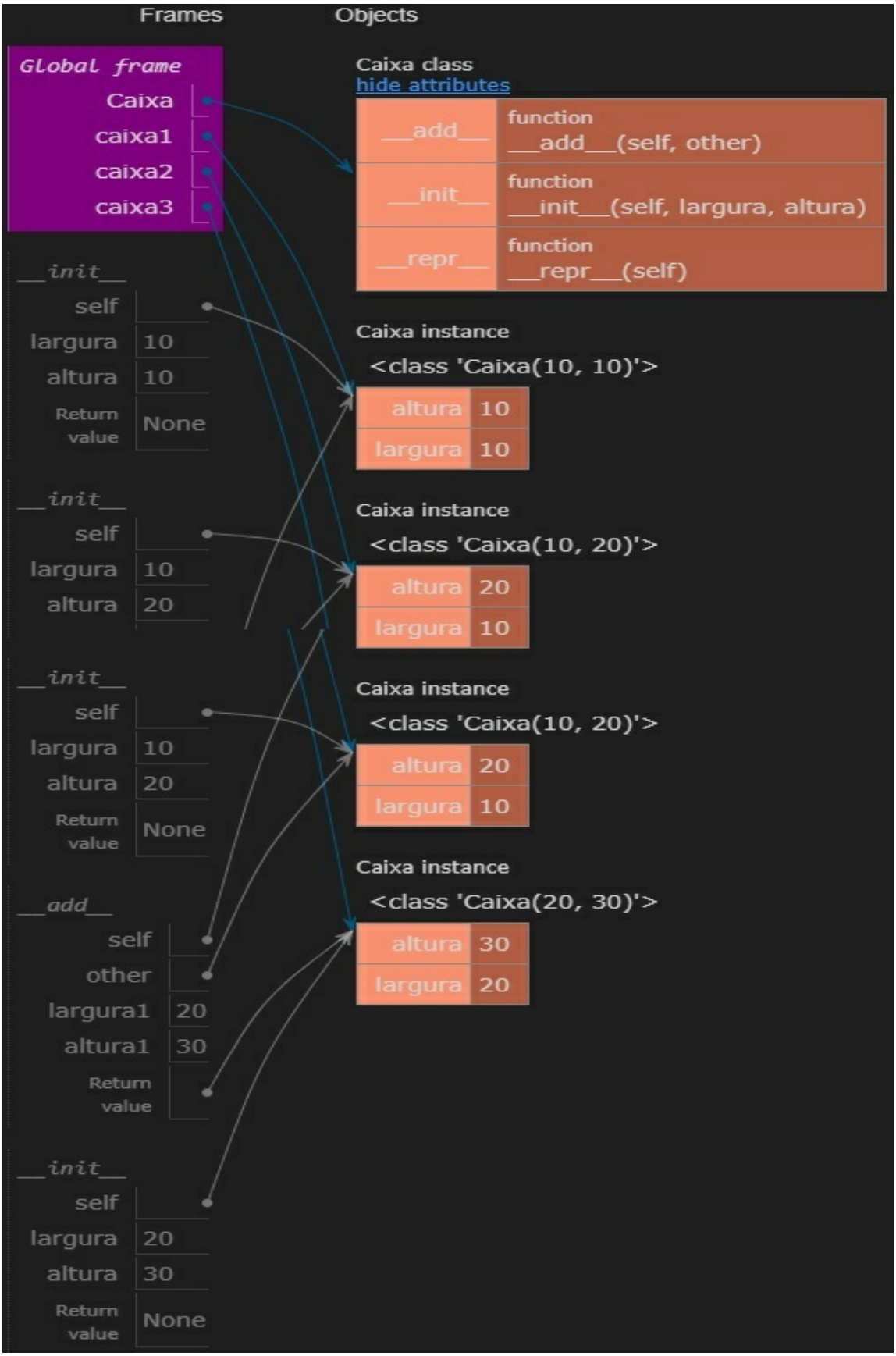
isto retornando para a classe tais valores. Na sequência, apenas para facilitar a visualização, foi criada um método de classe __repr__ (também reservado ao sistema) que irá retornar os valores das somas explicitamente ao usuário (lembre-se de que como estamos trabalhando dentro de uma classe, por padrão esta soma seria uma mecanismo interno, se você tentar printar esses valores sem esta última função, você terá como retorno apenas o objeto, e não seu valor).

```
1  class Caixa:
2      def __init__(self, largura, altura):
3          self.largura = largura
4          self.altura = altura
5
6      def __add__(self, other):
7          largura1 = self.largura + other.largura
8          altura1 = self.altura + other.altura
9          return Caixa(largura1, altura1)
10
11     def __repr__(self):
12         return f"<class 'Caixa({self.largura}, {self.altura})'>"
13
14 caixa1 = Caixa(10,10)
15 caixa2 = Caixa(10,20)
16 caixa3 = caixa1 + caixa2
17
18 print(caixa3)
19
```

```
<class 'Caixa(20, 30)'>
```

Por fim, agora instanciando a classe, atribuindo seus respectivos valores, finalmente é possível realizar normalmente a operação de soma entre as variáveis que instanciam essa classe.

Nesse exemplo, criando uma variável de nome caixa3 que recebe como atributo a soma dos valores de largura e altura de caixa1 e caixa2, o retorno será: <class 'Caixa(20, 30)'>


Frames
Objects
Global frame
Caixa
caixa1
caixa2
caixa3
Caixa class
hide attributes
__add__
function
__add__(self, other)
__init__
function
__init__(self, largura, altura)
__repr__
function
__repr__(self)
__init__
self
largura 10
altura 10
Return value
None
Caixa instance
<class 'Caixa(10, 10)'>
altura 10
largura 10
__init__
self
largura 10
altura 20
Caixa instance
<class 'Caixa(10, 20)'>
altura 20
largura 10
__init__
self
largura 10
altura 20
Return value
None
Caixa instance
<class 'Caixa(10, 20)'>
altura 20
largura 10
__add__
self
other
largura1 20
altura1 30
Return value
Caixa instance
<class 'Caixa(20, 30)'>
altura 30
largura 20
__init__
self
largura 20
altura 30
Return value
None

Encerrando, apenas tenha em mente que, de acordo com a funcionalidade que quer usar em seu código, deve procurar o método equivalente e o declarar dentro da classe como um método de classe qualquer, para “substituir” o operador lógico/aritmético de forma a ser lido e interpretador por nosso interpretador.

Tratando exceções

Tanto na programação comum estruturada quanto na orientada a objetos, uma prática comum é, na medida do possível, tentar prever os erros que o usuário pode cometer ao interagir com nosso programa e tratar isso como exceções, de forma que o programa não trave, nem feche inesperadamente, mas acuse o erro que o usuário está cometendo para que o mesmo não o repita, ou ao menos, fique ciente que aquela funcionalidade não está disponível.

Na programação estruturada temos aquela estrutura básica de declarar uma exceção e por meio dela, via try e except, tentar fazer com que o interpretador redirecione as tentativas sem sucesso de executar uma determinada ação para alguma mensagem de erro ao usuário ou para alguma alternativa que retorne o programa ao seu estado inicial. Na programação orientada a objetos esta lógica e sintaxe pode ser perfeitamente usada da mesma forma. Ex:

```
 1  class Erro001(Exception):
 2      pass
 3
 4  def erro():
 5      raise Erro001('Ação não permitida!')
 6
 7  try:
 8      erro()
 9  except Erro001 as msgerro:
10      print(f'ERRO: {msgerro}')
11
```

```
ERRO: Ação não permitida!
```

Repare na estrutura, é normalmente suportado pelo Python, que exceções sejam criadas também como classe. Inicialmente criamos uma classe de nome Erro001 que herda a função reservada ao sistema Exception.

A partir disto, criamos uma função que instancia uma variável para

Erro001 atribuindo uma string onde consta uma mensagem de erro pré-programada ao usuário (lembrando que a ideia é justamente criar mensagens de erro que orientem o usuário sobre o mesmo, e não as mensagens padrão de erro que o interpretador gera em função de lógica ou sintaxe errada).

Em seguida, exatamente igual ao modelo estruturado, criamos as funções try e except, onde realizamos as devidas associações para que caso o usuário cometa um determinado erro seja chamada a mensagem de exceção personalizada. Neste caso o retorno será: ERRO: Ação não permitida!

AVANÇANDO COM FUNÇÕES

Quando estamos iniciando nossos estudos de Python, assim que dominados os fundamentos da linguagem, o primeiro grande degrau a ser superado é quando começamos a entender e criar nossas próprias funções.

Aprendemos superficialmente a sintaxe básica de uma função, também aprendemos que a mesma pode ou não ter parâmetros, pode ou não de fato realizar alguma função, retornar ou não algum dado/valor, etc... E a partir deste ponto podemos começar a entender algumas particularidades que farão de nossos códigos algo mais produtivo, haja visto que em todo e qualquer programa o propósito inicial é realizar funções.

Parâmetros por Justaposição ou Nomeados

Uma das primeiras particularidades a serem consolidadas quando estamos avançando com nossos entendimentos sobre funções são sobre seus parâmetros, no sentido de que os mesmos podem ser atribuídos e manipulados por justaposição ou de forma nomeada conforme a necessidade.

Quando estamos aprendendo Python normalmente desconhecemos essas características pois em Python tudo é muito automatizado, de forma que quando criamos nossas primeiras funções parametrizadas o interpretador, em sua leitura léxica, automaticamente trata dos parâmetros conforme o contexto da função.

```
1                    1º    2º     3º
2  def pessoa1(nome, idade, funcao):
3      print(f'{nome} tem {idade} anos, função: {funcao}')
4
```

Nada melhor do que partir para prática para entender tais conceitos. Então, inicialmente, quando estamos falando de justaposição dos parâmetros de uma função estamos simplesmente falando da ordem aos quais estes parâmetros foram declarados.

Pela sintaxe Python toda função tem um nome com o prefixo def e um campo representado por parênteses onde podemos ou não definir parâmetros para nossa função.

Quando estamos criando nossas próprias funções temos de prezar pela legibilidade do código, contextualizando, uma boa prática de programação é justamente criar funções com nomes intuitivos, assim como quando houverem parâmetros, que os mesmos possuam nomes lógicos de fácil entendimento.

Voltando ao âmbito da justaposição em nosso exemplo, note que criamos uma função pessoa1( ) que recebe como parâmetros nome, idade e função, respectivamente.

A chamada justaposição se dá porque da maneira com que foram declarados estes parâmetros, seguindo uma ordem sequencial lógica, no momento de você atribuir algum dado/valor aos mesmos, a ordem de atribuição irá ser respeitada pelo interpretador.

Em outras palavras, ao criar uma variável que chame essa função, por padrão o primeiro dado que você repassar substituirá o primeiro atributo da função, neste caso: nome, o segundo atributo substituirá idade e logicamente o terceiro atributo repassado para essa função substituirá o terceiro atributo funcao.

Isto se dá porque o interpretador associará a ordem dos espaços alocados para parametrização com a ordem dos dados repassados parametrizando a função. Porém, é perfeitamente possível alterar essa forma de parametrizar uma determinada função, como veremos nos tópicos subsequentes.

```
1  def pessoa1(nome, idade = 18, funcao = 'técnico'):
2      print(f'{nome} tem {idade} anos, função: {funcao}')
3
```

Outra possibilidade que temos é trabalhar com os chamados parâmetros nomeados, onde no momento da declaração de tais parâmetros já atribuímos um dado padrão, que pode ou não ser substituído pelo usuário.

Raciocine que todos parâmetros que criamos para uma função devem ser substituídos por algum dado/valor e ser funcionais.

Não é recomendado se criar parâmetros além dos que serão realmente usados pela função, usar dessa prática pode levantar algumas exceções ou gerar erros de interpretação.

Pensando nisso, uma prática comum é, quando temos um determinado parâmetro que possa ser alterado ou não pelo usuário, já no momento da declaração estipular um dado/valor default para o mesmo.

Repare no exemplo, novamente criamos a função pessoa( ), onde seu primeiro parâmetro é nome, seu segundo parâmetro é idade = 18 e seu terceiro parâmetro agora é funcao = ‘técnico’.

A partir do momento que temos essa função criada dessa forma, supondo que o usuário fornecesse apenas um dado para o primeiro parâmetro

nome, os demais iriam usar dos dados padrão declarados, 18 e ‘técnico’, respectivamente repassados nas máscaras de substituição de idade e funcao.

```
1  def pessoa1(nome, idade = 18, funcao = 'técnico'):
2    print(f'{nome} tem {idade} anos, função: {funcao}')
3
4  p1 = pessoa1('Fernando', funcao = 'gerente')
5
```

Concluindo essa linha de raciocínio, usando da mesma função do exemplo anterior, vamos realizar uma parametrização tanto nomeada quanto justaposta.

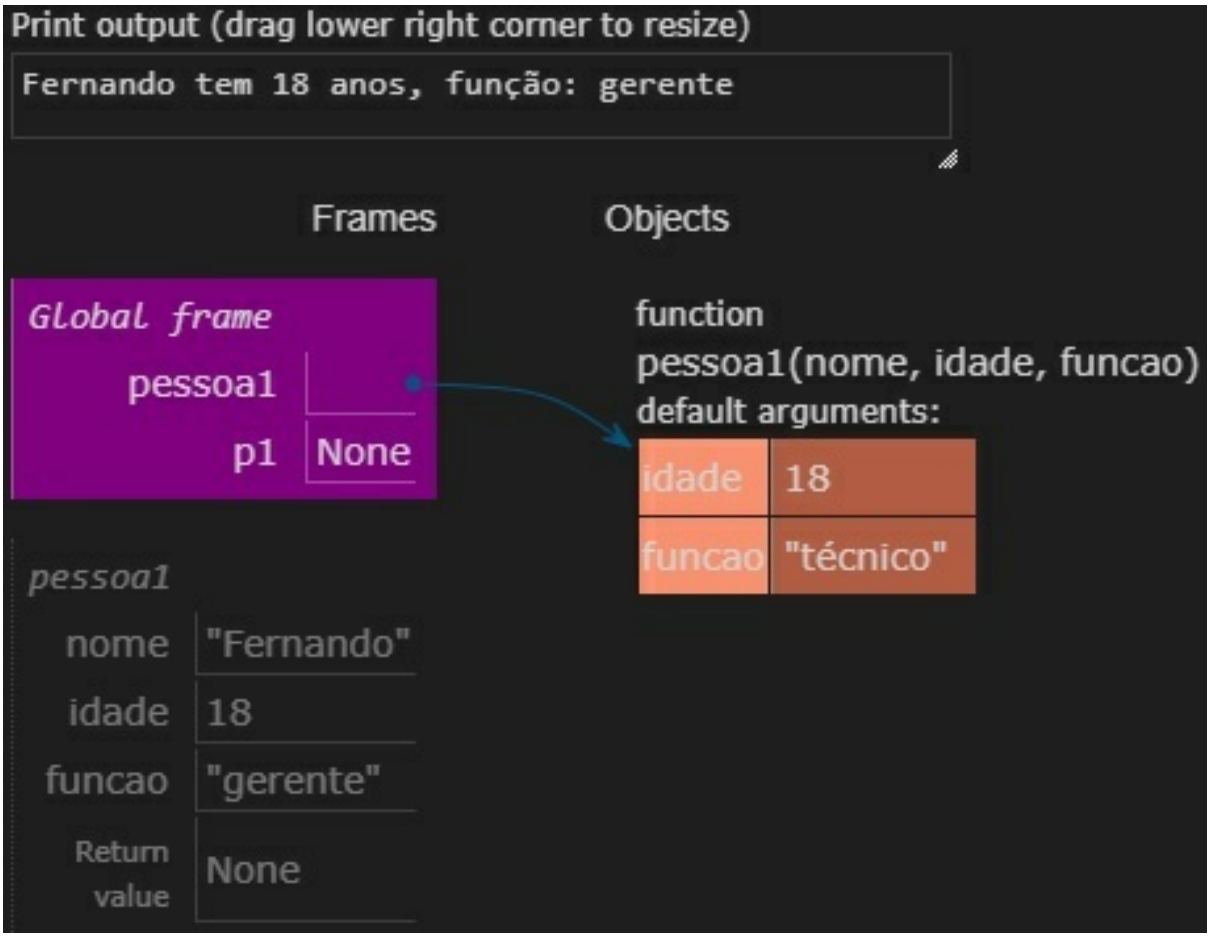
Note que para isso é criada uma variável p1 que chama a função pessoa1( ), onde o primeiro elemento colocado entre os parênteses é a string ‘Fernando’, que por justaposição irá substituir qualquer dado alocado ou existente em nome.

Em seguida declaramos que funcao agora tem como atributo a string ‘gerente’.

Por fim, como idade não foi parametrizada, no momento de seu uso ela fará uso de seu dado padrão 18.

```
Fernando tem 18 anos, função: gerente
```

Rodando essa função as devidas máscaras de substituição terão seus dados sobrescritos pelos parâmetros repassados, e como esperado, é exibida uma mensagem pela função print( ) do corpo da função.



Apenas revisando tais conceitos, quando criamos uma determinada função, criamos um número específico de parâmetros conforme a necessidade.

Tanto o nome da função quando de seus parâmetros deve ser um nome lógico e intuitivo, de fácil localização e entendimento por parte do desenvolvedor.

A ordem em que repassamos dados como parâmetros para nossa função, por padrão a parametrizará na mesma ordem de declaração.

É possível no momento da declaração dos parâmetros de uma função já declarar algum dado manualmente que pode ou não ser substituído conforme o contexto.

Por fim ao chamar uma determinada função é possível manualmente especificar qual parâmetro receberá um determinado dado, apenas fazendo referência ao seu nome.

*args e **kwargs

No exemplo anterior criamos uma simples função com apenas 3 parâmetros justapostos e/ou nomeados, porém uma situação bastante recorrente é não termos um número inicial de parâmetros bem definido.

Imagine que você está criando uma determinada de função em que seu corpo haverão uma série de subfunções cada uma com seus respectivos parâmetros, isso acarretaria em um código de legibilidade ruim e confusa.

Em Python é perfeitamente possível contornar essa situação fazendo o uso de *args e **kwargs.

Primeiramente repare na sintaxe, o uso de * asterisco faz com que estas palavras sejam reservadas ao sistema, é simplesmente impossível criar uma variável que seu nome comece com asterisco.

Toda vez que o interpretador em sua leitura léxica encontra um asterisco no campo de parâmetros de uma função ou em alguma comprehension, ele deduz que ali é um espaço alocado onde dados serão repassados.

Outro ponto interessante é que a nomenclatura *args e **kwargs é uma simples convenção, podendo ser criada como qualquer outro tipo de nome como *parametros e **parametros_nomeados por exemplo.

Aproveitando também esta linha de raciocínio, alguns autores costumam diferenciar *args de **kwargs de acordo com o tipo de parâmetro que será usado ou sua aplicação, recomendando o uso de *args sempre que estiver alocando espaço para parâmetros justapostos e **kwargs quando estiver reservando espaço para parâmetros nomeados.

Isto se dá apenas por uma questão de organização e convenção como boa prática de programação, uma vez que na prática não existe tal distinção.

Sendo assim, toda vez que tivermos uma situação onde inicialmente temos um número indefinido de parâmetros, podemos fazer uso de *args e **kwargs para tornar a legibilidade do código de nossas funções mais

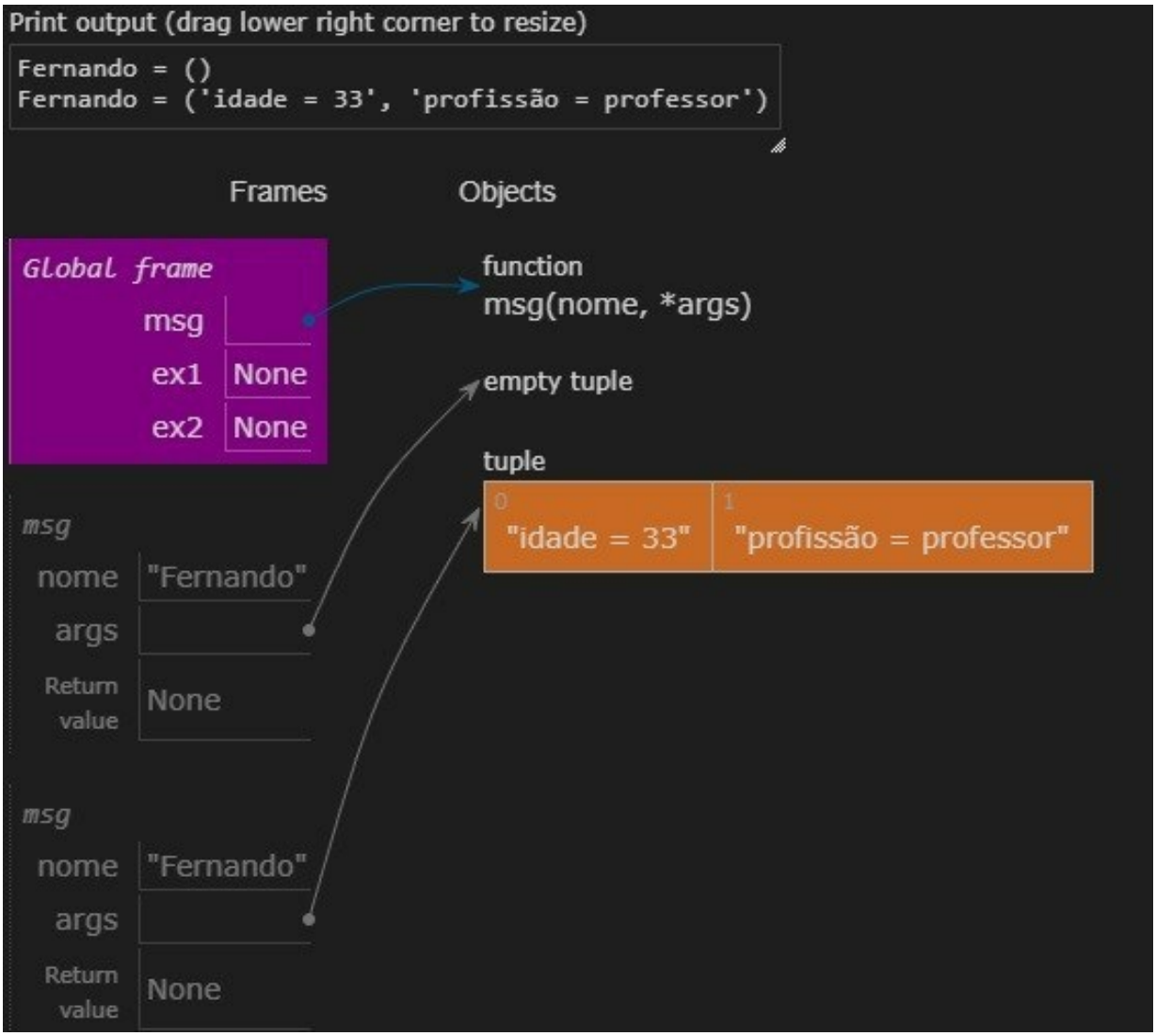
intuitivo.

```
1  def msg(nome, *args):
2    print(f'{nome} = {args}')
3
4  ex1 = msg('Fernando')
5  ex2 = msg('Fernando', 'idade = 33', 'profissão = professor')
6
```

Partindo para prática, note que é criada uma função de nome msg( ), que tem como parâmetros nome e *args. Dentro de seu corpo nesse exemplo é simplesmente criada uma mensagem onde via f’strings nas respectivas máscaras de substituição estarão os dados ‘Fernando’ para nome, assim como todos os outros dados dispostos sequencialmente conforme a declaração para *args.

```
Fernando = ()
Fernando = ('idade = 33', 'profissão = professor')
```

E acima temos o retorno esperado.


Print output (drag lower right corner to resize)
Fernando = ()
Fernando = ('idade = 33', 'profissão = professor')
Frames
Objects
Global frame
msg
ex1
None
ex2
None
function
msg(nome, *args)
empty tuple
tuple
0
"idade = 33"
1
"profissão = professor"
msg
nome
"Fernando"
args
Return value
None
msg
nome
"Fernando"
args
Return value
None

Empacotamento e Desempacotamento

Apesar do nome um pouco estranho nesse contexto, aqui simplesmente estamos trabalhando com a ideia de repassar múltiplos dados/valores como parâmetro de uma função.

Em outros momentos aprendemos que é possível adicionar mais de um tipo de dado/valor como atributo de uma variável, o que era feito normalmente por meio de listas, tuplas, conjuntos e dicionários, mas isso também se estende quando estamos criando funções.

Até o momento, gradualmente fomos aprendendo sobre diferentes possibilidades dentro de funções, porém até o momento tudo estava muito restrito a um dado/valor como parâmetro de uma função.

Porém, fazendo o uso de *args e **kwargs podemos extrapolar essa lógica e atribuir, por exemplo, todo o conteúdo de um dicionário como um parâmetro de uma função, e isto é convencionalmente conhecido como empacotamento e desempacotamento de dados.

```
1  numeros = (33, 1987, 2020, 19.90, 100000000)
2  dados = {'Nome':'Fernando', 'Profissão':'Engenheiro'}
3
4  def identificacao(*args, **kwargs):
5      print(args)
6      print(kwargs)
7
8  identificacao(*numeros, **dados)
9
```

Visualizando estes conceitos em um exemplo, repare que aqui inicialmente criamos uma tupla de nome numeros com alguns dados numéricos (int e float) em sua composição, da mesma forma criamos um simples dicionário chamado dados com apenas dois conjuntos de chaves e valores.

A partir deste ponto criamos uma função de nome identificacao( ) parametrizada com *args e **kwargs. No corpo dessa função simplesmente

incluímos duas funções print( ) responsáveis por exibir em tela o conteúdo que estiver atribuído para *args e para kwargs respectivamente.

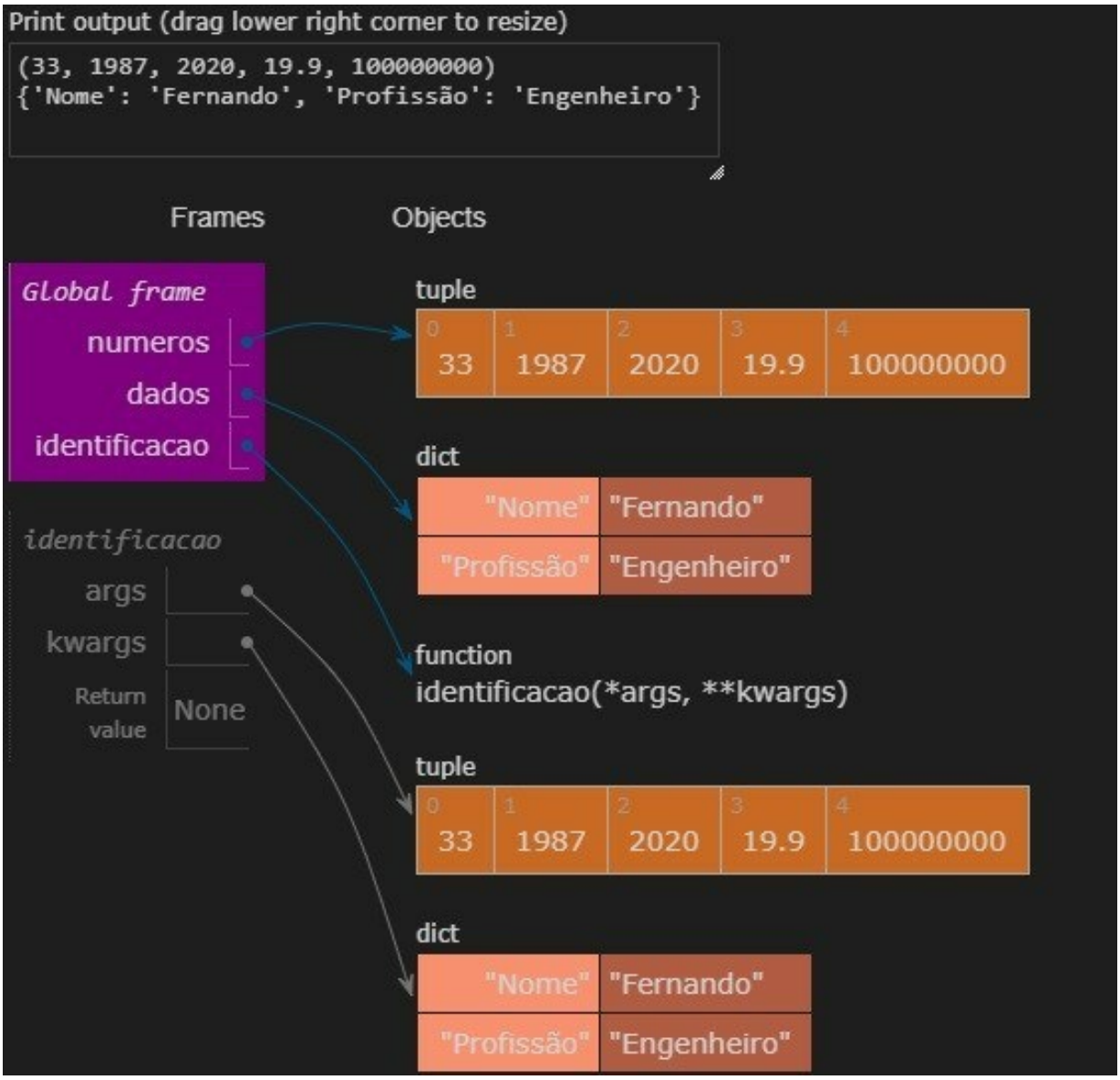
Por fim, repare que chamamos a função identificacao( ) parametrizando a mesma com *numeros no lugar de *args e dados no lugar de *kwargs.

Como visto nos capítulos anteriores, “args” e “kwargs” são normalmente usados apenas como convenção, neste exemplo fica claro que tais nomes podem ser alterados livremente.

Nesse caso, *args e *kwargs levam os nomes das próprias variáveis que contém os dados a serem usados como parâmetro de nossa função.

```
(33, 1987, 2020, 19.9, 100000000)
{'Nome': 'Fernando', 'Profissão': 'Engenheiro'}
```

Como esperado, o retorno das funções print( ) são os respectivos dados oriundos de numeros e de dados, parâmetros da função identificacao( ).


Print output (drag lower right corner to resize)
(33, 1987, 2020, 19.9, 100000000)
{'Nome': 'Fernando', 'Profissão': 'Engenheiro'}
Frames
Objects
Global frame
numeros
dados
identificacao
identificacao
args
kwargs
Return value
None
tuple
0 33
1 1987
2 2020
3 19.9
4 100000000
dict
"Nome" "Fernando"
"Profissão" "Engenheiro"
function
identificacao(*args, **kwargs)
tuple
0 33
1 1987
2 2020
3 19.9
4 100000000
dict
"Nome" "Fernando"
"Profissão" "Engenheiro"

Expressões Lambda

Em Python, expressões lambda, também conhecidas como funções anônimas, nada mais são do que funções simples, que realizam tarefas simples, normalmente associadas a uma variável, de forma que sua função é executada apenas naquele bloco de código, sem ser por padrão reutilizáveis.

```
1  variavel1 = lambda n1, n2: n1 * n2
2
```

Vamos ao exemplo, e, como de costume, usando de um exemplo simples para melhor entendimento. Inicialmente criamos uma variável de nome variavel1 que recebe lambda seguido de n1, n2: n1 * n2.

Vamos entender ponto a ponto, nossa expressão lambda está diretamente atribuída a nossa variável variavel1, repare que lambda é uma palavra reservada ao sistema, e em sua sequência temos dois blocos de elementos, n1 e n2 são os parâmetros da função lambda, após os “ : “ dois pontos está o que a função em si realiza, que neste simples caso é a multiplicação de n1 por n2.

Sendo assim, podemos facilmente entender que a sintaxe de uma expressão lambda é “lambda parametro_1 , parametro_2 : função lógica ou aritmética a ser realizada”.

```
1  def multiplicacao(n1, n2):
2      return n1 * n2
3
4  ou
5
6  multiplicacao = n1 * n2
7
```

Você muito provavelmente está habituado a ter de criar uma função da maneira convencional, com declaração e corpo, ou ter que fazer uso de uma variável acessória que realiza a operação e guarda o resultado em si, porém, como dito anteriormente, quando necessário o uso de uma função bastante simples como a de nosso exemplo, a mesma pode ser uma expressão

lambda.

```
1  variavel1 = lambda n1, n2: n1 * n2
2
3  n1 = int(input('Digite um número:'))
4  n2 = int(input('Digite outro número:'))
5
6  print(n1)
7  print(n2)
8
9  print(variavel1(n1,n2))
10
```

A partir do momento da criação de nossa função de multiplicação de dois elementos em formato de expressão lambda, podemos realizar normalmente as demais interações com o usuário assim como a exibição do resultado em tela.

Apenas um pequeno adendo, uma vez que uma variável possui como atributo uma expressão lambda, sempre que você instanciar tal variável ela irá executar a expressão/função atribuída para si.

```
Digite um número:6
Digite outro número:7
6
7
42
```

Como esperado, supondo que o usuário digitou 6 e 7 respectivamente, na última linha do retorno é exibido o resultado da função expressão lambda.



Apenas por curiosidade, repare nesta representação gerada pelo Python Preview, diferente de outros códigos apresentados até o momento, aqui não existe uma função com um espaço alocado em memória exclusivo para tal, já que nossa expressão lambda é sim uma função, mas ela é lida

como um simples atributo de variável.

RECURSIVIDADE

Na programação, tanto estruturada quanto orientada a objetos, dada a complexidade de nosso programa, boa parte das vezes criamos funções bastante simples e com característica de ser executada apenas uma vez para um determinado fim em nosso programa, sendo em seguida desconsiderada.

Porém, programas mais robustos podem exigir que determinadas funções sejam executadas várias vezes, normalmente até alcançar um objetivo, isto na prática se chama recursividade.

Importante salientar que essa lógica aqui é bastante parecida com algumas estruturas condicionais, porém, trabalhando com recursividade aplicada nossas funções não somente usarão de estruturas condicionais internamente, mas irão se comportar como uma à medida que suas funcionalidades são requisitadas pelo interpretador.

Diferentemente das estruturas for e while, uma função recursiva pode ser reutilizada inúmeras vezes conforme a demanda, assim como possui uma característica padrão de que quando construída, em algum momento ela retornará ela mesma reprocessando os dados.

Pode parecer um pouco confuso, mas raciocine que neste tipo de arquitetura de dado, podemos dentro da função chamar ela mesma para que funcione em loop até uma certa condição ser validade.

Também é uma prática comum uma função recursiva ser criada de modo que gere um retorno atribuído a uma variável externa, pois ao final de sua execução ela pode ser chamada por outra variável, com outros parâmetros, sem interferir nos retornos gerados anteriormente.

Em resumo, em Python é perfeitamente permitido que tenhamos estruturas de dados aninhadas, como listas dentro de listas, dicionários dentro de dicionários, estruturas condicionais dentro de estruturas condicionais, sendo assim por quê não, funções dentro de funções.

Outro ponto a ser considerado é que no processo de criação de todo o bloco de uma função recursiva, parte do código será dedicado a

funcionalidade que retornará algum dado/valor para alguma variável, assim como parte do código será dedicado a chamar a própria função em loop, o que para alguns autores é chamado como caso base e caso recursivo, respectivamente.

Desde que respeitadas a sintaxe e a estrutura lógica da função em si, esta prática funcionará perfeitamente.

```
1  def fatorial(num: int) -> int:
2      if num == 1:
3          return 1
4
5      return num * fatorial(num - 1)
6
```

Partindo para prática, vamos entender a lógica de uma função recursiva sobre um exemplo onde estamos criando uma função que recebe um número e retorna seu fatorial (o mesmo número multiplicado por todos seus antecessores).

Inicialmente declaramos a função fatorial( ) que recebe como parâmetros um num já especificado que deve ser do tipo int e que deve retornar um número do tipo int.

Em seguida é criada uma estrutura condicional onde quando o valor de num for igual a 1, retorne 1. Isso diz respeito a própria lógica de um fatorial, o último expoente pelo qual o número será multiplicado é 1. Note que este bloco é o caso base, onde consta as linhas referentes a funcionalidade da função em si.

Logo após, fora da estrutura condicional existe um retorno bem peculiar, preste bastante atenção na forma com que esse retorno é declarado. Aqui, no chamado caso recursivo, o retorno chama a própria função fatorial( ) onde ele mesmo está inserido.

Nesse retorno o valor atribuído a num é multiplicado pelo fatorial parametrizado com o próprio num subtraindo seu valor final em 1. Em outras palavras, estamos criando uma cadeia de execução em loop, onde essa função será executada novamente se retroalimentando com o valor de num, até que o mesmo seja igual a 1.

```
7  fator = fatorial(12)
8  print(fator)
9
```

Na sequência é criada uma variável de nome fator, que chama a função fatorial( ) parametrizando a mesma com 12.

```
479001600
```

E como esperado, o resultado fatorial de 12 é 479001600

Código Completo:

```
[ ]  1  def fatorial(num: int) -> int:
     2      if num == 1:
     3          return 1
     4
     5      return num * fatorial(num - 1)
     6
     7  fator = fatorial(12)
     8  print(fator)
     9

479001600
```

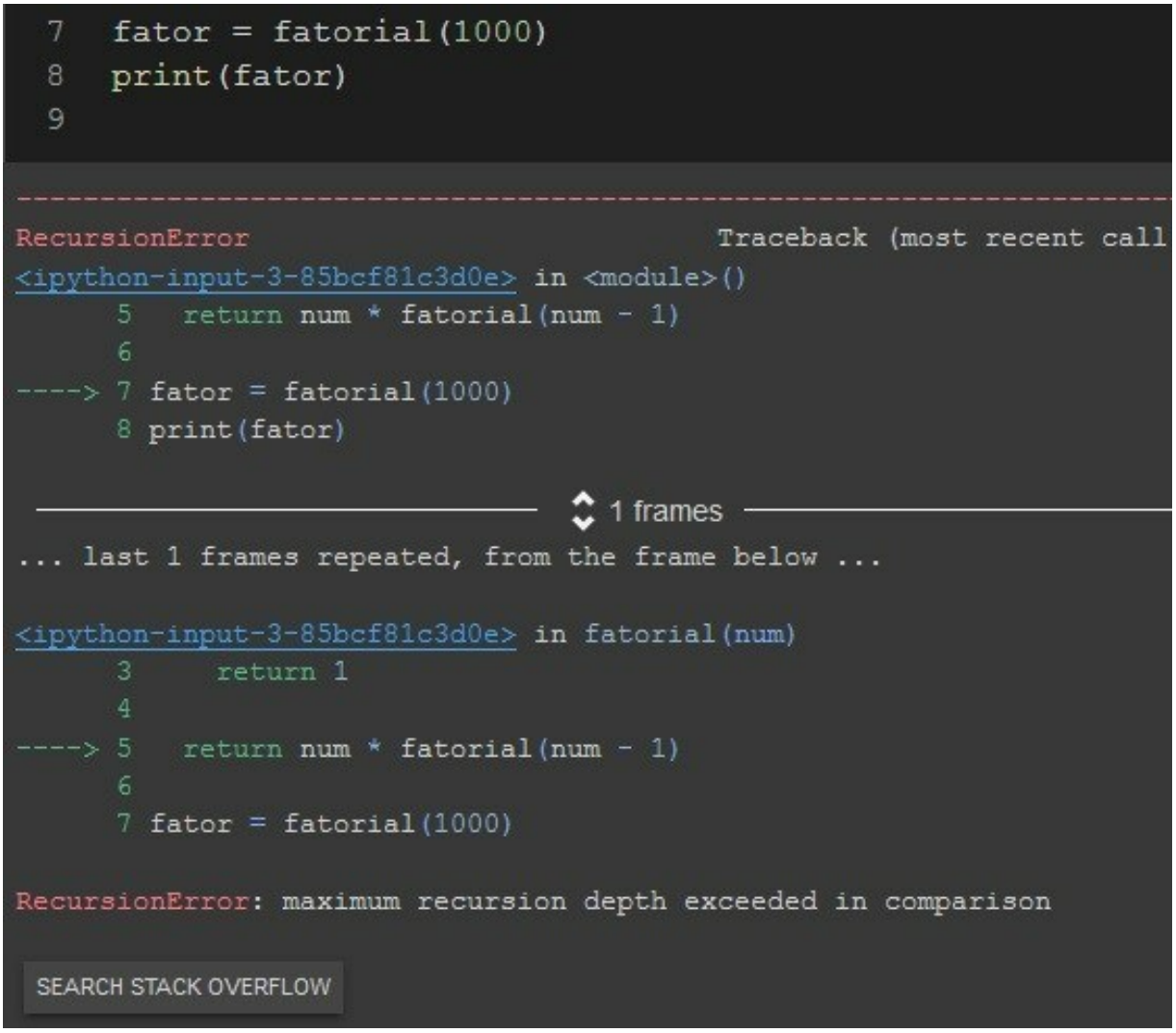
Se você quiser realizar o debug do código, perceberá que até o final do processo, a função fatorial( ) será executada 12 vezes.

Lembrando que cada execução dessas estará em um espaço de memória alocado, sendo assim, dependendo o número a ser fatorado o retorno pode ser enorme, proporcional ao número de recursões, inclusive excedendo os limites de memória pré-definidos automaticamente por segurança.

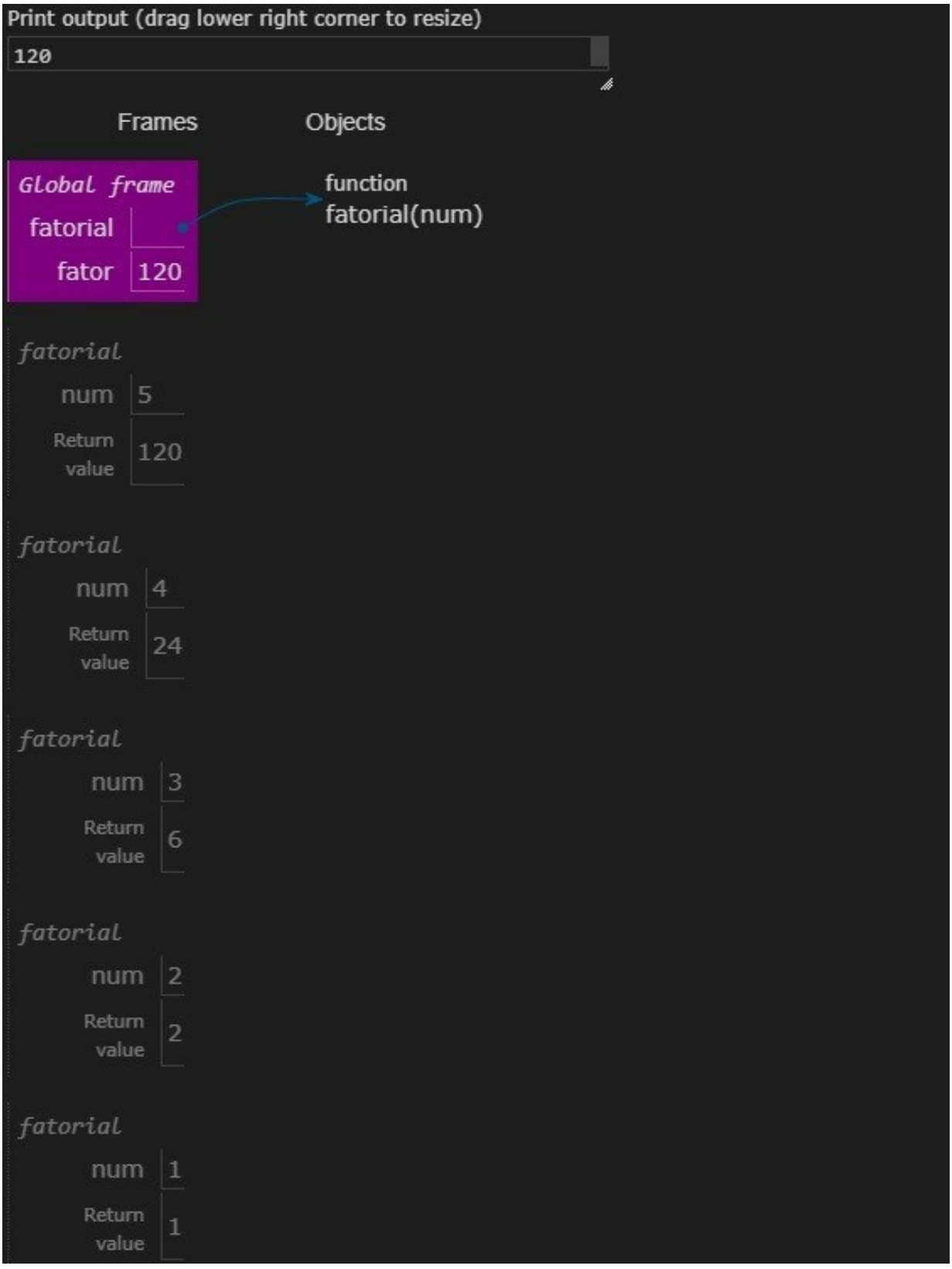
```
7  fator = fatorial(500)
8  print(fator)
9

1220136825991110068701238785423046926253574342803192842192413588 3
```

Apenas como exemplo, fatorando o número 500 é gerado um retorno com mais de 1000 dígitos.



Tentando realizar a fatoração de 1000 é gerado um traceback. Esta conta seria possível, mas excede o limite de memória disponível para este tipo de função.



Fatorial de 5.

Sendo assim, apenas como curiosidade, sempre que houver a possibilidade de um retorno extrapolar a memória, ou de uma função recursiva entrar em um loop infinito, será gerado um traceback.

Porém, supondo que você realmente tenha uma aplicação que necessite suporte a números muito grandes, você pode configurar manualmente um limite por meio dos comandos import sys e sys.setrecursionlimit(5000), aqui parametrizado com 5000 apenas como exemplo.

Dessa forma o interpretador do Python irá obedecer a este limite estipulado.

EXPRESSÕES REGULARES

Quando estamos trabalhando diretamente com dados do tipo string, estamos habituados a aplicar sobre os mesmos uma série de funções nativas do Python a fim de manipular os elementos da mesma.

Porém, a forma básica como manipulamos elementos de uma string costuma ser muito engessada em conceitos que funcionam para qualquer tipo de dado na verdade. Pensando nisso, foram criadas uma vasta gama de bibliotecas que adicionaram novas possibilidades no que diz respeito a manipulação de strings.

Uma das formas mais adotadas é a chamada Expressões Regulares, onde por meio da biblioteca homônima, conseguimos ter ferramentas adicionais para encontrar e realizar o tratamento de certos padrões em uma string.

r’Strings e Metacaracteres

Um dos conceitos iniciais que veremos na prática sua lógica e funcionamento são as chamadas r’Strings.

Se você chegou a esse ponto certamente conhece as chamadas f’Strings, sintaxe moderna implementada no Python 3 para tratar máscaras de substituição de forma mais reduzida e intuitiva do que o padrão “.format”.

Pois bem, em f’Strings temos em meio nossas strings máscaras de substituição que suportam todo e qualquer tipo de dado, assim como operações lógicas e aritméticas.

Não muito diferente disso teremos nossas r’Strings, onde trabalhando especificamente com dados em formato de texto podemos adicionar algumas expressões em meio aos próprios caracteres da string, chamando funções internas para serem aplicadas naquele ponto.

Diferentemente de f’Strings e suas máscaras de substituição representadas por { }, em r’Strings teremos o que chamamos de metacaracteres, que naquele contexto serão lidos e interpretados pelo interpretador como marcadores para alguma função.

Raciocine que por parte do interpretador em sua leitura léxica, quando o mesmo encontrava o marcador f’ já esperava que em seguida houvesse uma string composta de caracteres e máscaras de substituição.

O mesmo ocorre aqui, quando o interpretador carrega a biblioteca re e encontra ao longo do código um marcador r’, ele já presume que na sequência existe uma string composta de caracteres e metacaracteres.

Metacaracteres

Um dos primeiros e mais básicos metacaracteres a ser entendido é o metacaracter “ . “ (ponto). De modo convencional o símbolo de ponto em uma string nada mais é do que mais um caractere inserido ali para dar o contexto de encerramento de uma sentença/frase. Agora, tratando-se de ponto como um metacaractere ele assume um papel bastante diferente do usual.

O metacaractere ponto é usado em casos onde estaremos realizando a leitura de uma string simples, normalmente uma palavra, onde um determinado caractere pode ser representado por vários elementos alterando seu contexto.

```
1  import re
2
3  minha_string = 'Fernando chegará a meia noite.'
4
5  print(re.findall(r'Fernand.', minha_string))
6
```

Para ficar mais claro tais conceitos, vamos partir para a prática.

Inicialmente é importada a biblioteca re por meio do comando import re, na sequência é criada uma variável de nome minha_string que recebe como atributo ‘Fernando chegará a meia noite’.

Por fim, por meio da função print( ), chamamos a função re.findall( )*, passando dois parâmetros para a mesma, uma r’String e um objeto. Em nossa r’String, supondo que pudéssemos ter tanto a string ‘Fernando’ quanto ‘Fernanda’, note que o último caractere é a diferença das mesmas.

Sendo assim, poderíamos consultar se Fernando ou Fernanda constam no corpo daquela string, usando do metacaractere “ . “ no lugar do o ou a em questão.

Apenas finalizando, como segundo parâmetro para findall( ) passamos o objeto como referência de onde serão buscadas tais strings, nesse caso, minha_string.

```
['Fernando']
```

O resultado como esperado, é a string ‘Fernando’.

*Iremos ver as demais funções da biblioteca regex na sequência.

Poderíamos perfeitamente duplicar o código referenciando tanto ‘Fernando’ quanto ‘Fernanda’, mas isso não é uma boa prática de programação.

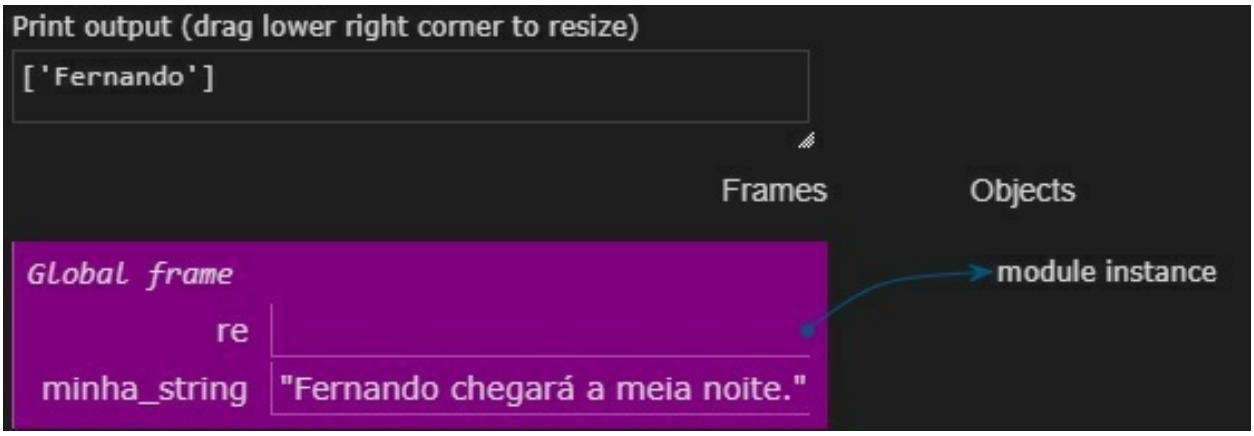
```
1  import re
2
3  minha_string = 'Fernando chegará a meia noite.'
4  print(re.findall(r'Fernand.', minha_string))
5
6  minha_string2 = 'Fernanda chegará a meia noite.'
7  print(re.findall(r'Fernand.', minha_string2))
8
```

Dois blocos de código realizando a mesma função significa o dobro de processamento. Sendo assim, uma maneira de buscarmos mais de uma possível palavra em uma string é fazendo o uso do metacaractere “ | “.

No contexto de metacaracteres, o símbolo | (pipe) representa “ou”, contextualizando para nosso exemplo, estamos buscando duas variações do mesmo nome, ou ‘Fernando’ ou ‘Fernanda’ e isso pode perfeitamente ser feito em nossa r’String.

```
1  minha_string = 'Fernando chegará a meia noite.'
2  print(re.findall(r'Fernando|Fernanda', minha_string))
3
```

Repare que agora em nossa r’String temos a expressão ‘Fernando|Fernanda’, em outras palavras, caso existe uma ou outra dessas palavras, a mesma será retornada por nossa função print( ).




```
1  texto1 = '''No dia trinta de março de dois mil e vinte
2  foi inaugurado o novo ginásio de esportes da escola
3  estatual Professor Annes Dias, na cidade de Cruz Alta,
4  no estado do Rio Grande do Sul. A obra inicialmente
5  orçada em um milhão de, reais acabou não utilizando de
6  todo o recurso, uma vez que dois grandes empresários da
7  cidade, João Fagundes e Maria Terres doaram juntos em
8  torno de duzentos mil reais. João Fagundes se manifestou
9  dizendo que apoiou a obra pois acredita no desenvolvimento
10 da cidade, assim investe regularmente para hospitais e
11 escolas da mesma. maria Terres não quis se pronunciar
12 sobre o assunto.'''
13
```

Partindo para um exemplo um pouco mais elaborado, inicialmente criamos uma variável de nome texto1 que recebe literalmente um texto em forma de string.

A partir da mesma vamos explorar algumas outras possibilidades.

```
14  print(re.findall(r'.oão', texto1))
15
```

Como já visto no exemplo inicial, usando do metacaractere . podemos em seu lugar ter qualquer tipo de caractere.

```
14  print(re.findall(r'.oão', texto1))
15  print(re.findall(r'.oão|.aria', texto1))
16
```

Usando dos dois metacaracteres apresentados até o momento, note que acabamos por contornar um possível problema de nossa string.

Se houvéssemos buscado especificamente por ‘Maria’, o interpretador iria literalmente ignorar ‘maria’ que está em outra linha do texto.

O metacaracter . serve inclusive para esse propósito, pois o mesmo recebe qualquer caractere e no caso de alfabéticos não diferencia maiúsculas de minúsculas.

```
['João', 'João']
['João', 'Maria', 'João', 'maria']
```

Como esperado, a função findall( ) encontrou as devidas strings de nossa variável.

```
print(re.findall(r'.oão|.aria', texto1))
print(re.findall(r'.oão|.aria|mil', texto1))

```

Apenas concluindo esta linha de raciocínio, o metacaractere | pode ser usado quantas vezes forem necessárias, assim como a combinação entre | e . são perfeitamente aceitáveis para qualquer r’string.

```
['João', 'Maria', 'João', 'maria']
['mil', 'mil', 'João', 'Maria', 'mil', 'João', 'maria']
```

Como esperado, é nos retornadas as strings parametrizadas para findall( ).

Outro metacaractere muito comum de ser usado é o [], cuidado para não confundir com a sintaxe padrão de uma lista em Python, quando estamos falando especificamente de expressões regulares, o símbolo [] significa um conjunto de elementos.

```
print(re.findall(r'[Jj]oão|.aria', texto1))

```

Usando ainda do exemplo anterior, note que no caso de ‘João” sabemos que as possíveis alterações nessa string seriam apenas quanto ao caractere inicial ser maiúsculo ou minúsculo, dessa forma, podemos especificar via [ ]

que naquela posição da string os possíveis caracteres são J ou j.

```
1  print(re.findall(r'[AaBbCcDdEeFfGgHhIiJj]oão', texto1))
2
```

Apenas para fins de exemplo, quando estamos definindo uma expressão regular de um conjunto de possíveis caracteres, não há limite sobre quantos ou quais caracteres poderão estar ocupando aquela posição na string.

```
3  print(re.findall(r'[a-z]oão|[A-Z]oão', texto1))
4  print(re.findall(r'[a-zA-Z]oão', texto1))
5
```

Outra possibilidade que temos quando trabalhamos com o metacaractere [ ] é definir um intervalo dentro do mesmo. Por exemplo, que naquele espaço possa conter qualquer caractere de a-z minúsculo ou de A-Z maiúsculo.

```
1  print(re.findall(r'jOãO|maRIA', texto1, flags = re.I))
2
```

Em certos casos podemos nos deparar com strings totalmente desconfiguradas quanto a seus caracteres estarem em letras maiúsculas ou minúsculas de forma aleatória.

Nesses casos é possível atribuir um terceiro parâmetro para findall( ) chamado flags que é definido como re.I, em outras palavras, essa sinalização para função findall( ) faz com que a mesma ignore o formato dos caracteres e os leia normalmente.

Metacaracteres Quantificadores

Entendida a lógica dos metacaracteres de substituição, podemos dar mais um passo, agora entendendo das expressões regulares os chamados metacaracteres quantificadores.

Anteriormente vimos que por meio de r'Strings podemos manipular o conteúdo de uma string de forma mais completa se comparado com as ferramentas nativas padrão do Python.

Porém todo exemplo usado até o momento era considerando uma string de gramática correta, mas na realidade boa parte do uso de expressões regulares serão para manipular strings levando em consideração os vícios de linguagem do usuário.

Uma boa prática de programação é tentar prever os erros que possam ser cometidos pelo usuário e tratar essas exceções evitando que o programa pare de funcionar.

Tratando-se de strings a lógica não é muito diferente disso, uma vez que boa parte dos textos gerados são com expressões, gírias, abreviações ou até mesmo erros de gramática, e tudo isso deve ser levado em consideração.

```
1  mensagem = 'Nããããããããããoooooooo esquecer de pagar a mensalidade, não deixar para depois'
2
```

Partindo para o exemplo, note que temos uma variável de nome mensagem onde atribuída para si tem uma string composta de uma frase de linguagem usual, não formal, pois existem vícios de linguagem como 'Nããããããããããoooooooo'.

```
3  print(re.findall(r'não', mensagem, flags = re.I))
4
```

Tentando usar dos conceitos entendidos até o momento, ao tentarmos buscar a string 'não' em mensagem por meio da expressão regular convencional, o retorno será um pouco diferente do esperado.

```
['não']
```

O grande problema neste caso é que existe uma interpretação literal por

parte do interpretador, em outras palavras, ele buscará por padrão exatamente a string ‘não’ em sua forma padrão de caracteres e ordem de declaração dos mesmos.

Para contornar este tipo de problema, uma possibilidade que temos é de, fazendo o uso de metacaracteres quantificadores, dizer para nosso interpretador que um ou mais caracteres naquela string podem se repetir, devendo serem lidos e não ignorados.

Complementar a nossos metacaracteres “ . “ e “ | “, agora teremos mais 4 metacaracteres que nos permitirão tratar elementos repetidos em uma string.

Basicamente, o metacaractere “ * “ significa que um determinado elemento da string pode se repetir 0 ou um número ilimitado de vezes. Da mesma forma, o metacaractere “ + “ significa que um certo elemento se repetirá 1 ou ilimitadas vezes.

Um pouco diferente dos anteriores, o metacaractere “ ? “ significa que um determinado elemento se repetirá 0 ou 1 vez na string.

Por fim o metacaractere “ {} “ significa que um elemento específico da string se repetirá dentro de um intervalo definido.

Diferente dos metacaracteres convencionais que ocupavam a posição exata do elemento a ser alterado, tratando-se dos metacaracteres quantificadores, os mesmos devem estar à frente do caractere em questão.

Vamos para prática para que tais conceitos fiquem melhor entendidos.

```
3  print(re.findall(r'não', mensagem, flags = re.I))
4
5  print(re.findall(r'Nã+o', mensagem))
6  print(re.findall(r'Nã+o+', mensagem))
7  print(re.findall(r'[Nn]ã+o+', mensagem))
8  print(re.findall(r'nã+o+', mensagem, flags = re.I))
9
```

Trabalhando em cima da string de nossa variável mensagem, como visto anteriormente, no primeiro print( ) usando da expressão regular convencional, o interpretador apenas fez a leitura de ‘não’, ignorando ‘Nããããããããooooooo’ completamente.

Agora usando de expressões regulares quantificadoras, podemos mostrar a nosso interpretador que para tal r'String existem caracteres repetidos a serem lidos.

Na linha 5, temos uma função print( ) parametrizada com a função findall( ) de re, que por sua vez recebe uma r'String.

Repare que aqui é declarada a string Nã+o, que em outras palavras significa que para o caractere "ã" podem haver a repetição desse caractere 1 vez ou um número ilimitado de vezes.

```
['não']
['Nãããããããããão']
```

Repare que no retorno obtido, solucionamos o problema de todos caracteres "ã" repetidos na string, porém, conforme o exemplo, o caractere "o" também possui repetições e foi ignorado.

Isto se dá porque cara metacaractere quantificador deve estar a frente do respectivo caractere em questão. Em nosso exemplo, via " + " nosso interpretador buscou corretamente todas repetições de "ã+".

Sendo assim, note que na linha 6 temos exatamente a mesma linha de código, porém, fazendo uso de dois metacaracteres " + ", um para cada caractere com possíveis repetições.

```
['não']
['Nãããããããããão']
['Nããããããããããooooooo']
```

O resultado, como esperado, é finalmente toda a string 'Nããããããããooooooo', porém, de acordo com nossa string, temos duas palavras "não", e até o momento tratamos da que estava escrita de forma usual.

Aprimorando um pouco mais nosso tratamento de string, na linha 7 especificamos que para o primeiro caractere de "não" pode haver um conjunto de elementos, nesse caso, sabemos que as únicas possibilidades são [Nn].

Logo, colocamos este metacaractere incorporado na string na sua posição correta.

```
['não']
['Nããããããããããão']
['Nãããããããããããoooooooo']
['Nãããããããããããoooooooo', 'não']
```

Eis que nosso interpretador finalmente encontrou as duas palavras “não”, para que possamos manipular as mesmas conforme a necessidade.

Apenas finalizando nossa linha de raciocínio, neste caso, onde estávamos a buscar uma palavra simples “não”, sabíamos que as únicas possibilidades de diferenciação era quanto ao uso de caractere inicial maiúsculo ou minúsculo “n”, sendo assim, poderíamos realizar o mesmo feito por meio de flags como já visto anteriormente.

```
['não']
['Nããããããããããão']
['Nãããããããããããoooooooo']
['Nãããããããããããoooooooo', 'não']
['Nãããããããããããoooooooo', 'não']
```

Como esperado, os retornos referentes as linhas 7 e 8 são exatamente os mesmos.

Apenas como um adendo, importante salientar que não existem restrições quanto aos tipos e número de metacaracteres usados dentro de uma expressão regular, mas temos de tomar cuidade e prezar pela legibilidade de nosso código sempre.

```
1  chamando_nome = 'Fernnnnnnnannnnndo'
2
3  print(re.findall(r'fer[a-z]+a[Nn]+do', chamando_nome, flags = re.I))
4
```

Repare nesse exemplo, inicialmente criamos uma variável de nome chamando_nome que recebe uma string como atributo.

Buscando a string via expressão regular e metacaracteres, note a quantidade de metacaracteres usados na r’String, exageros como esse comprometem a legibilidade do código.

Neste exemplo, existem tantos metacaracteres que fica difícil até mesmo ler que em meio a eles existe a string ‘fernando’. Sendo assim,

devemos prezar por fazer o uso de metacaracteres conforme realmente necessário.

Métodos de Expressões Regulares

Nos tópicos anteriores usamos em meio a nosso entendimento de metacaracteres a função findall( ), e não por acaso, escolhi usar dela em nossos exemplos pois na prática é a função de expressão regular mais usada.

Avançando um pouco com métodos de expressões regulares, podemos facilmente entender a função re.sub( ).

Uma vez que em expressões regulares estamos realizando um tratamento mais avançado de nossas strings, nada mais comum do que ter uma função dedicada a substituir elementos da mesma, e por meio da biblioteca regex, isso é feito via função sub( ).

```
1  texto1 = '''No dia trinta de março de dois mil e vinte
2  foi inaugurado o novo ginásio de esportes da escola
3  estatual Professor Annes Dias, na cidade de Cruz Alta,
4  no estado do Rio Grande do Sul. A obra inicialmente
5  orçada em um milhão de, reais acabou não utilizando de
6  todo o recurso, uma vez que dois grandes empresários da
7  cidade, João Fagundes e Maria Terres doaram juntos em
8  torno de duzentos mil reais. João Fagundes se manifestou
9  dizendo que apoiou a obra pois acredita no desenvolvimento
10 da cidade, assim investe regularmente para hospitais e
11 escolas da mesma. maria Terres não quis se pronunciar
12 sobre o assunto.'''
13
14 print(re.sub(r'jOãO', 'Carlos', texto1, flags = re.I))
15
```

Usando novamente de nossa variável texto1 e sua string, vimos anteriormente que um dos nomes citados ao longo do texto era João.

Supondo que quiséssemos substituir todas as instâncias de João no texto por outro nome, Carlos por exemplo, isso seria feito alterando a função findall( ) por sub( ).

Outra particularidade é que como parâmetro dessa função devemos na r'String especificar qual string será substituída e por qual, assim como o

nome da variável em questão onde consta toda a string.

```
No dia trinta de março de dois mil e vinte
foi inaugurado o novo ginásio de esportes da escola
estatual Professor Annes Dias, na cidade de Cruz Alta,
no estado do Rio Grande do Sul. A obra inicialmente
orçada em um milhão de, reais acabou não utilizando de
todo o recurso, uma vez que dois grandes empresários da
cidade, Carlos Fagundes e Maria Terres doaram juntos em
torno de duzentos mil reais. Carlos Fagundes se manifestou
dizendo que apoiou a obra pois acredita no desenvolvimento
da cidade, assim investe regularmente para hospitais e
escolas da mesma. maria Terres não quis se pronunciar
sobre o assunto.
```

Como esperado, é retornada toda a string de texto1, com todas as referências a “João” sobrescritas por “Carlos”.

```
1  minha_string = 'Fernando chegará a meia noite.'
2
3  print(re.match(r'Fernando', minha_string))
4  print(re.match(r'noite', minha_string))
5
```

Outra função da biblioteca regex é a match( ), ela basicamente tem funcionalidade parecida com a de findall( ) porém bem mais limitada.

Enquanto findall( ) como próprio nome sugere, busca em toda a string, match é usada apenas quando queremos verificar a primeira palavra de um texto.

```
<_sre.SRE_Match object; span=(0, 8), match='Fernando'>
None
```

Note que no exemplo, reaproveitando nossa variável minha_string de exemplos anteriores, ao usar da função match( ), parametrizando a r’String com ‘Fernando’, é retornado um objeto fazendo referência ao espaço de memória alocado para o mesmo.

Da mesma forma, tentando buscar outra palavra de outra posição do texto, o retorno sempre será None.

```
1  minha_string2 = 'Fernando chegará, a meia noite.'
2
3  print(re.split(r',', minha_string2))
4
```

Outra possibilidade interessante é a de separar em partes uma string via expressão regular. Para isso, da biblioteca regex usaremos da função split( ).

Diferente da função split( ) nativa do Python, que desmembra uma string caractere por caractere, a função split( ) da biblioteca regex recebe um marcador, normalmente um símbolo especial, para que separe a string em duas partes, uma antes e outra depois deste marcador.

Nesse exemplo, usando uma vírgula como separador para o texto, basta referenciar a mesma em nossa r'String.

```
['Fernando chegará', ' a meia noite.']
```

O resultado, como esperado, serão duas strings independentes.

Outros metacaracteres e funções podem ser encontradas diretamente na documentação da biblioteca regex, aqui, trouxe apenas as que de fato uso rotineiramente para que você tenha este conhecimento.

Link para documentação oficial da biblioteca regex:

https://docs.python.org/3/library/re.html

LIST COMPREHENSION

Avançando um pouco no que diz respeito a manipulação de dados a partir de uma lista, iremos otimizar nossos códigos fazendo o uso da chamada List Comprehension sobre nossas listas.

A linguagem Python, como já mencionado anteriormente, é uma linguagem dinâmica onde temos diversas possibilidades de codificar a mesma coisa, também não é nenhum segredo que a partir da versão 3 da mesma foram implementadas uma série de novas funcionalidades, assim como o aprimoramento de funcionalidades já existentes.

Quando dominamos a sintaxe básica da linguagem, tanto em sua forma estruturada quanto orientada a objetos, temos rotineiramente um grande uso de dados em formato de listas.

Pensando nesse quesito, ao longo das atualizações da linguagem Python foram implementadas diversas novas funcionalidades específicas para manipulação de dados a partir de listas.

Uma das formas mais modernas e ao mesmo tempo eficientes é a chamada List Comprehension.

```
1  lista1 = [1,2,3,4,5,6]
2
3  for i in lista1:
4      print(i * 2)
5
```

Como exemplo inicial, vamos simplesmente supor que temos uma lista qualquer com alguns elementos e precisamos multiplicar cada um desses elementos por 2.

O método convencional, adotado por boa parte dos estudantes de Python é criar um laço de repetição for que percorre cada um dos elementos da lista e retorna seu valor multiplicado.

```
1  lista1 = [1,2,3,4,5,6]
2
3  lista2 = [i * 2 for i in lista1]
4
```

Otimizando nosso código, repare que temos exatamente a mesma lista, porém agora criamos uma nova variável de nome lista2, que recebe como atributo uma list comprehension realizando a mesma função do exemplo anterior, porém de forma reduzida e otimizada via list comprehension.

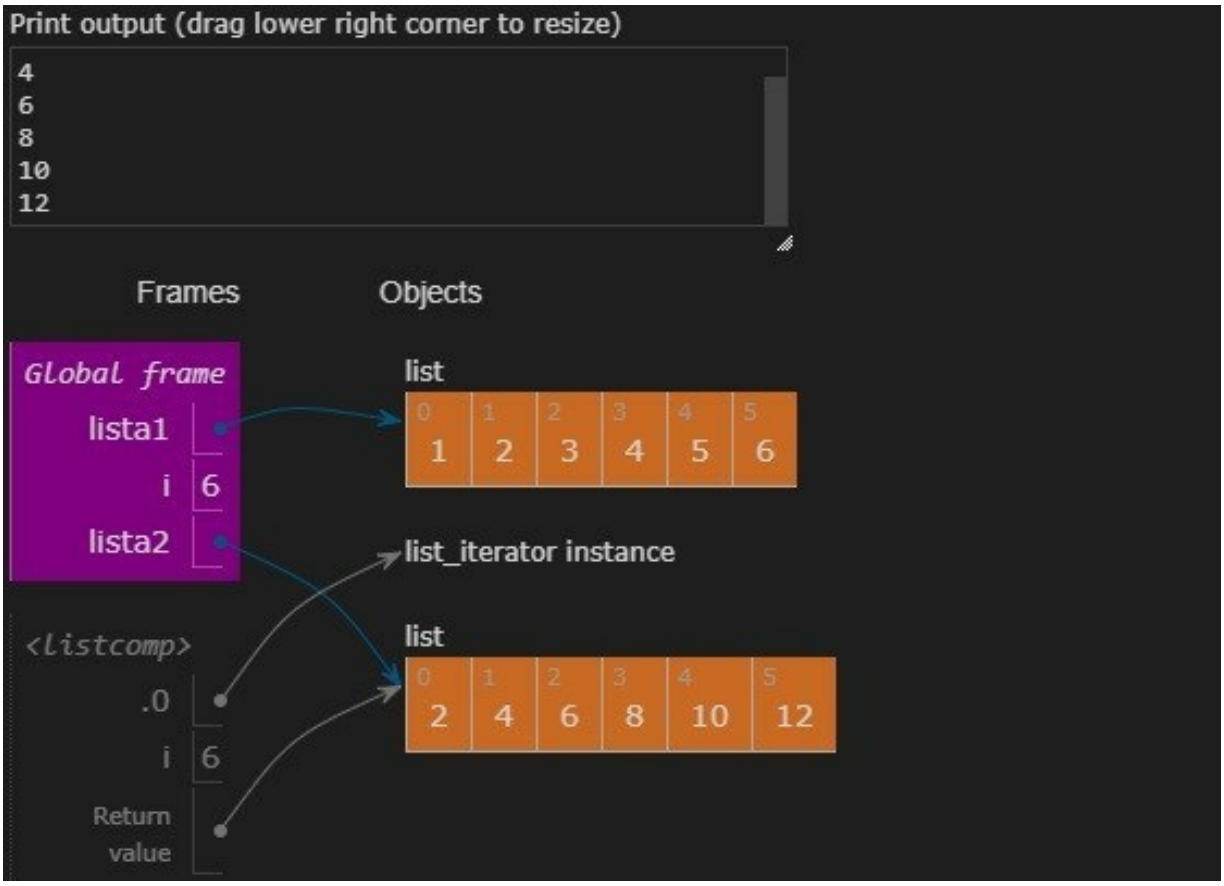
Vamos entender ponto a ponto o que está acontecendo aqui, preste muita atenção a lista atribuída para lista2, em formato de lista está uma expressão que realiza o mesmo procedimento do laço for feito anteriormente, importando os dados de lista1.

O que pode parecer um pouco estranho inicialmente é que pela conotação da sintaxe Python estamos acostumados a, quando criamos uma lista manualmente, abrir e fechar colchetes e já alimentar a lista com seus respectivos dados.

Na chamada List Comprehension podemos usar de expressões lógicas e aritméticas diretamente dentro dos colchetes da função, sem precisar nenhuma codificação adicional, uma vez que tal funcionalidade é nativa do Python 3.

```
2
4
6
8
10
12
[2, 4, 6, 8, 10, 12]
```

Como esperado, ambos métodos funcionam perfeitamente, na vertical o retorno do laço for manual, na horizontal, o retorno do laço for via list comprehension.


Print output (drag lower right corner to resize)
4
6
8
10
12
Frames
Objects
Global frame
lista1
i
6
lista2
list
0 1 2 3 4 5
1 2 3 4 5 6
list_iterator instance
<listcomp>
.0
i
6
Return value
list
0 1 2 3 4 5
2 4 6 8 10 12

Desmembrando strings via List Comprehension

Apesar da nomenclatura “List Comprehension”, tenha em mente que tal expressão vale para todo e qualquer tipo de dado em Python, o que difere na verdade é que a expressão “comprehension” é criada em formato de lista.

Uma das aplicações mais comuns de “comprehension” é quando estamos trabalhando com uma string e precisamos desmembrar a mesma elemento por elemento para algum fim.

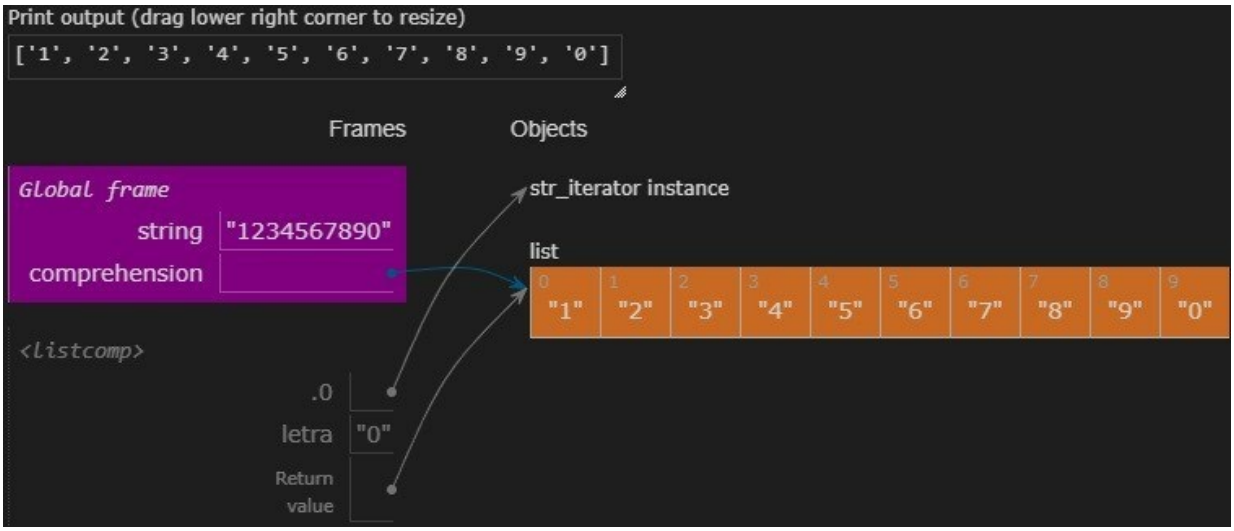
```
1  string = '1234567890'
2  comprehension = [letra for letra in string]
3
4  print(comprehension)
5
```

Como exemplo temos uma variável de nome string que, como é de se esperar, possui como atributo um dado em formato de string.

Na sequência criamos uma variável de nome comprehension que recebe como atributo uma list comprehension, na mesma lógica do exemplo anterior, realizando um laço for que percorre cada elemento da string e exibe este resultado.

```
['1', '2', '3', '4', '5', '6', '7', '8', '9', '0']
```

O resultado é uma lista onde cada elemento da string anterior agora está separada como elemento individual da lista.


Print output (drag lower right corner to resize)
['1', '2', '3', '4', '5', '6', '7', '8', '9', '0']
Frames
Objects
Global frame
string
"1234567890"
comprehension
str_iterator instance
list
0 "1"
1 "2"
2 "3"
3 "4"
4 "5"
5 "6"
6 "7"
7 "8"
8 "9"
9 "0"
<listcomp>
.0
letra
"0"
Return value

Manipulando partes de uma string via List Comprehension

```
1  string = '12345678901234567890592524952956 2'
2
3  print(string[0:10])
4  print(string[10:20])
5  print(string[20:30])
6
```

Quando estamos trabalhando com strings é bastante comum realizar o fatiamento das mesmas para fazer uso de apenas algumas partes, isso é comumente feito referenciando o intervalo onde tais caracteres da string se encontram.

Por exemplo, na imagem acima, temos uma variável de nome string que tem como atributo uma série de números em formato de string.

No primeiro print( ) estamos buscando os caracteres dentro do intervalo de índice 0 a 10, da mesma forma o segundo print( ) exibirá os caracteres do índice 10 ao 20 e por fim a terceira função print( ) exibirá os caracteres indexados de 20 a 30.

```
1234567890
1234567890
5925249529
```

Como esperado, temos as sequências de caracteres quanto ao seu fatiamento.

Podemos não somente otimizar este código como usar de outras funcionalidades via list comprehension.

```
1  string = '01234567890123456789012345678901 2'
2  n = 10
3  comp = [i for i in range(0, len(string), n)]
4
5  print(comp)
6
```

Usando da mesma variável do exemplo anterior, note que agora criamos uma variável de nome n que recebe 10 como atributo. Em seguida criamos uma variável de nome comp que recebe como atributo uma list comprehension.

Dentro da mesma temos um laço for que percorrerá toda a extensão da variável string, iniciando em zero, lendo todo seu comprimento, pulando os números em um intervalo de 10 em 10 de acordo com o parâmetro definido em n.

```
 7  comp2 = [(i, i + n) for i in range(0, len(string), n)]
 8
 9  print(comp2)
10
```

Da mesma forma podemos, como já estamos habituados, a verificar o tamanho desse intervalo. Em outras palavras, de quantos em quantos elementos serão feitos os fatiamentos.

```
11  comp3 = [string[i:i + n] for i in range(0, len(string), n)]
12
13  print(comp3)
14
```

Sabendo o tamanho da string e definidos os intervalos, podemos perfeitamente extrair os dados em si. Lembrando que na variável string temos uma série de caracteres numéricos, porém em formato de texto.

```
15  lista = [string[i:i + n] for i in range(0, len(string), n)]
16  retorno = '.'.join(lista)
17
18  print(retorno)
19
```

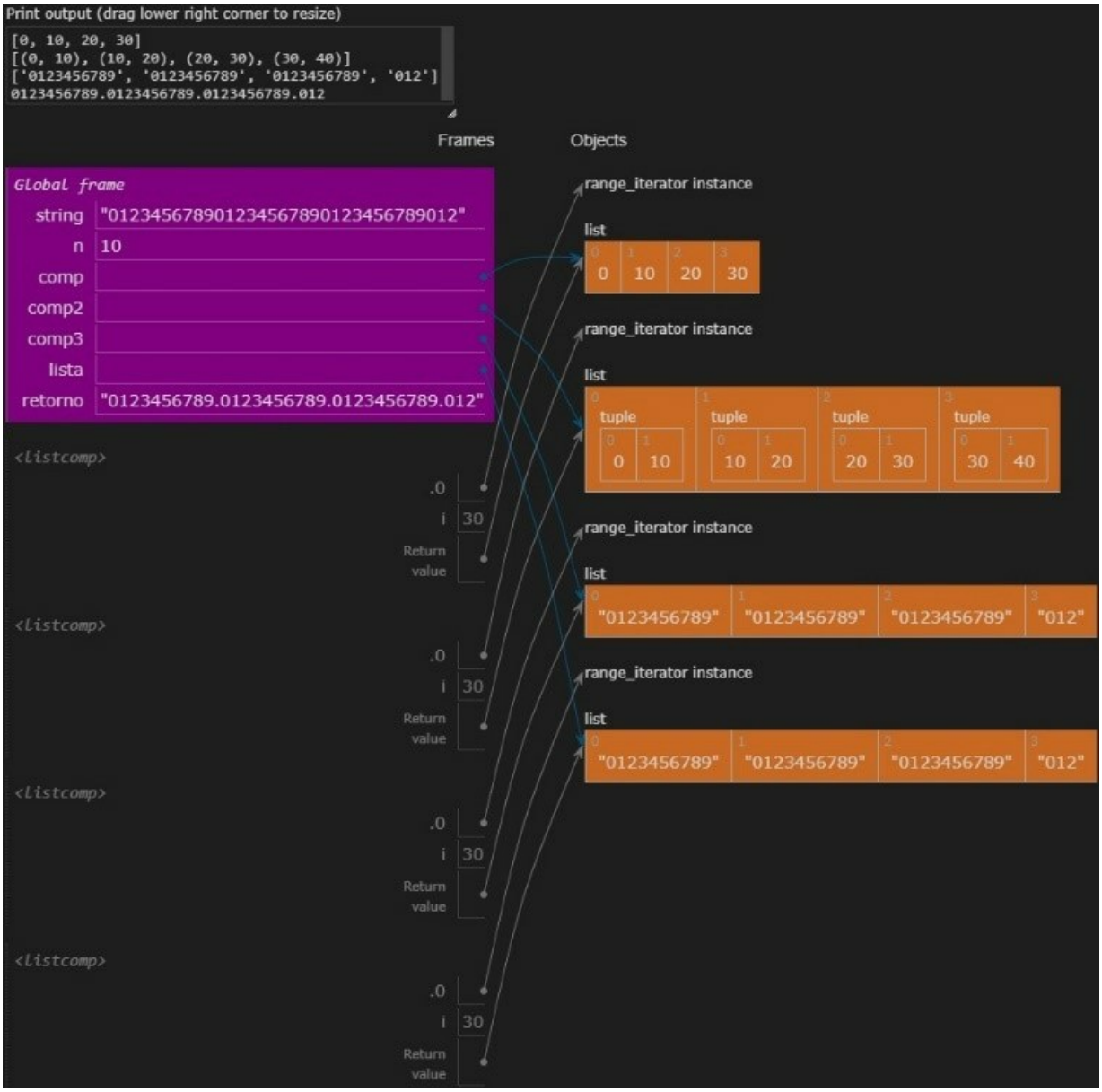
Por fim, podemos pegar tais elementos e concatená-los, atribuindo

os mesmos a uma nova variável. Nesse caso, criamos uma variável de nome lista que realiza o fatiamento dos dados lidos anteriormente.

Também é criada uma variável de nome retorno que via função join( ) pega tais dados, os concatena e atribui os mesmos para si.

```
[0, 10, 20, 30]
[(0, 10), (10, 20), (20, 30), (30, 40)]
['0123456789', '0123456789', '0123456789', '012']
0123456789.0123456789.0123456789.012
```

Como esperado, para cada linha temos o respectivo retorno de nossas funções print( ). A primeira delas mostrando o intervalo a ser lido, na segunda linha o intervalo dos elementos (de qual a qual), na terceira linha os elementos fatiados em si, por fim na quarta linha os elementos concatenados da variável retorno.



Como este tópico pode inicialmente ser um pouco confuso, vamos realizar a otimização via list comprehension por meio de um exemplo prático.

```
carrinho = []
carrinho.append(('Item 1', 30))
carrinho.append(('Item 2', 45))
carrinho.append(('Item 3', 22))
total = 0

for produto in carrinho:
    total = total + produto[1]

print('Método tradicional', total)

```

Supondo que temos um simples sistema de carrinho de compras, onde podemos adicionar novos itens, realizar a leitura de tais itens via laço for e por fim somar os valores para fechar o carrinho de compras.

O método convencional basicamente é como o da imagem acima, usando de um laço for e algumas variáveis acessórias para ler e somar os valores dos itens do carrinho.

Apenas ressaltando que este método não está errado, tudo o que estamos trabalhando aqui é no sentido de tornar a performance de nosso código melhor, reduzindo o mesmo e usando de expressões que internamente demandam menor processamento.

```
total2 = []
for produto in carrinho:
    total2.append(produto[1])
print('Usando funções internas', sum(total2))

```

Seguindo com nossa linha de raciocínio, o mesmo laço for usado anteriormente poderia ser otimizado fazendo o uso de algumas funções internas do Python como por exemplo a função sum( ) para a devida soma dos valores dos itens de carrinho.

Repare que aqui o que estamos fazendo é criando uma lista vazia na variável total2 onde o laço for percorre todos dados de carrinho, pegando apenas os elementos de índice 1, que no caso são os preços dos itens, adicionando para total2.

```
1  total3 = sum([y for x, y in carrinho])
2
3  print('Usando list comprehension', total3)
4
```

Aprimorando ainda mais o código, o mesmo pode ser reescrito por meio de uma list comprehension. Para isso criamos uma variável de nome total3 que recebe como atributo uma list comprehension.

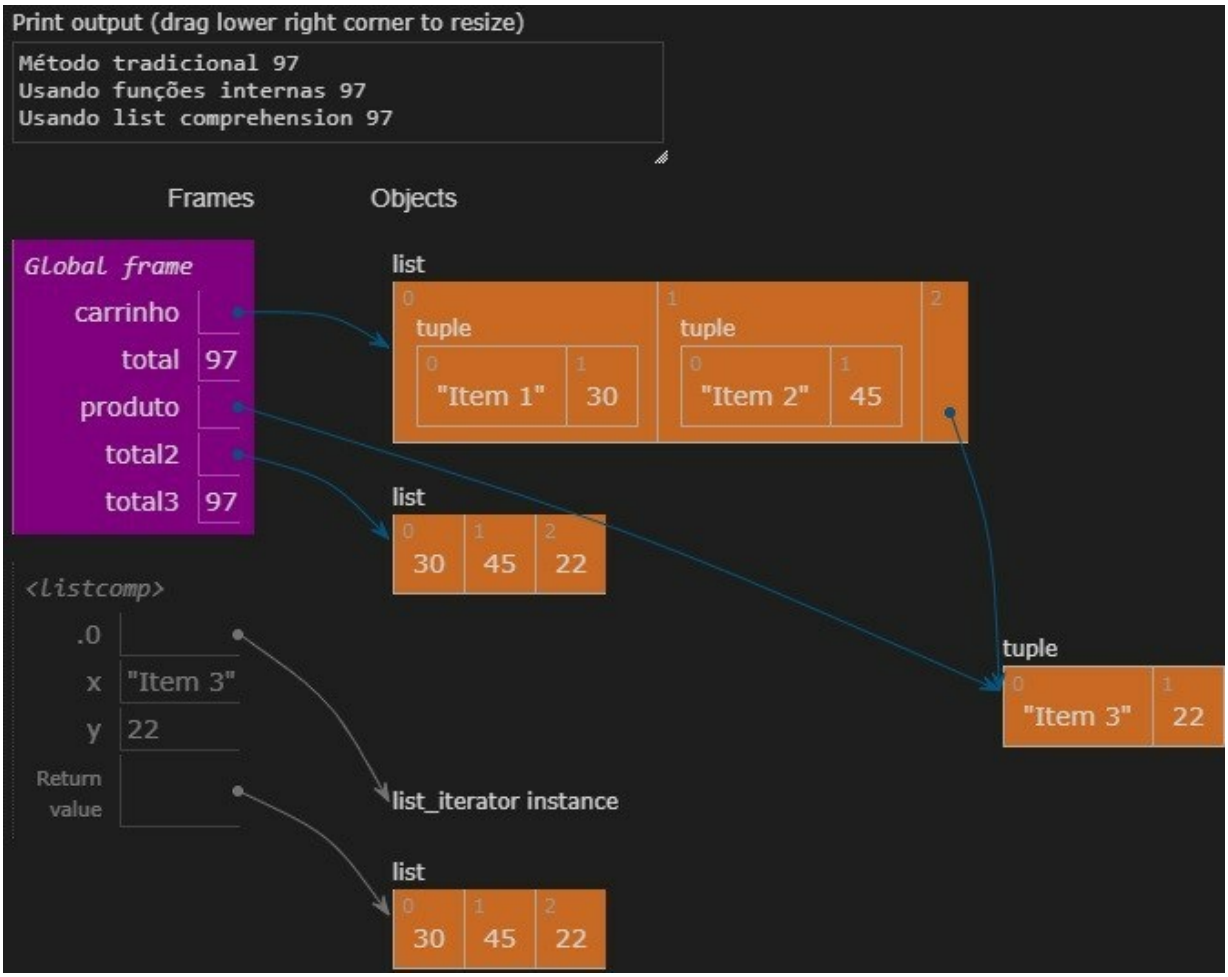
Note que dentro da mesma temos uma expressão onde é feita a leitura de cada elemento em sua posição y, retornando a soma destes elementos.

```
Método tradicional 97
Usando funções internas 97
Usando list comprehension 97
```

Executando esse código, novamente temos os retornos das respectivas funções print( ) em suas respectivas linhas.

Na primeira, de acordo com o método convencional, na segunda com o método otimizado padrão e na terceira com o método otimizado via list comprehension.

Os resultados, como esperado, são os valores das somas dos itens de carrinho. Novamente salientando, uma vez que temos aqui 3 formas diferentes de se chegar ao mesmo resultado, o ideal em uma aplicação real é fazer o uso do meio de melhor performance, nesse caso, via list comprehension.


Print output (drag lower right corner to resize)
Método tradicional 97
Usando funções internas 97
Usando list comprehension 97
Frames
Objects
Global frame
carrinho
total
97
produto
total2
total3
97
list
0
tuple
0
"Item 1"
1
30
1
tuple
0
"Item 2"
1
45
2
list
0
30
1
45
2
22
<listcomp>
.0
x
"Item 3"
y
22
Return value
tuple
0
"Item 3"
1
22
list_iterator instance
list
0
30
1
45
2
22

DICTIONARY COMPREHENSION

Uma vez entendida a lógica de list comprehension, o mesmo pode ser feito com dados em formato de dicionário, respeitando, claro, sua sintaxe.

Contextualizando, enquanto trabalhamos com dados em formato de lista, estamos acostumados a respeitar cada dado/valor como simples elemento da lista, já no âmbito de dicionários, estamos habituados a trabalhar com dados em formato de chave:valor.

Sendo assim, podemos facilmente adaptar nossas “comprehensions” de modo que iteramos sobre dois objetos, um para a chave e outro para valor, respectivamente.

```
1  lista = [('chave1','chave2'), ('chave2','valor2'),('chave3','valor3')]
2
3  dicionario = {x:y for x, y in lista}
4
```

Exemplificando, inicialmente temos uma variável de nome lista, em seu conteúdo, alguns pares de dados entre parênteses e com seus devidos separadores.

A partir disso, podemos realizar a conversão desta lista para um dicionário por meio de comprehension.

Sendo assim, criamos uma variável de nome dicionario que recebe como atributo uma expressão em forma de dicionário de acordo com a sintaxe. Na expressão em si, simplesmente criamos duas variáveis temporárias x e y que percorrerão separadamente via laço for os dados de lista.

```
{'chave1': 'chave2', 'chave2': 'valor2', 'chave3': 'valor3'}
```

Por meio da função print( ) passando como parâmetro o conteúdo de dicionario, é nos retornado um dicionário onde cada par de dados de lista foi adaptado em formato de chave:valor de dicionario.

A este ponto você deve estar se perguntando qual a justificativa lógica para um exemplo desses, haja visto que existe uma forma mais fácil de fazer tal conversão pelo próprio método dict( ) nativo do Python.

```
3  dicionario = {x:y for x, y in lista}
4  dicionario2 = dict(lista)
5
6  print(dicionario)
7  print(dicionario2)
8
```

Nesse caso, de fato apenas realizamos a conversão de um tipo de dado para outro, porém, lembre-se que a grande vantagem de usar comprehensions é reduzir o código, logo, o uso convencional de uma comprehension em um dicionário é fazendo o uso de expressões e operações no momento de sua criação/adaptação.

Tanto na linha 3 quanto na linha 4 o resultado será o mesmo, cabendo ao desenvolvedor optar por qual método julga mais eficiente.

```
{'chave1': 'chave2', 'chave2': 'valor2', 'chave3': 'valor3'}
{'chave1': 'chave2', 'chave2': 'valor2', 'chave3': 'valor3'}
```

Repare que nesse caso, ambos retornos da função print( ), para ambas variáveis dicionario e dicionario2 são exatamente iguais.

```
1  produtos = [('Caneta',1.99), ('Lápis',1.49), ('Caderno',8.99)]
2
3  produtos_com_imposto = {x:y * 1.6 for x, y in produtos}
4
5  print('Preços SEM Imposto: ', produtos)
6  print('Preços COM Imposto: ', produtos_com_imposto)
7
```

Partindo para uma aplicação real, inicialmente criamos uma variável de nome produtos, onde simplesmente seu conteúdo é uma lista , onde cada elemento da mesma é um par de dados simulando um determinado item com seu respectivo preço.

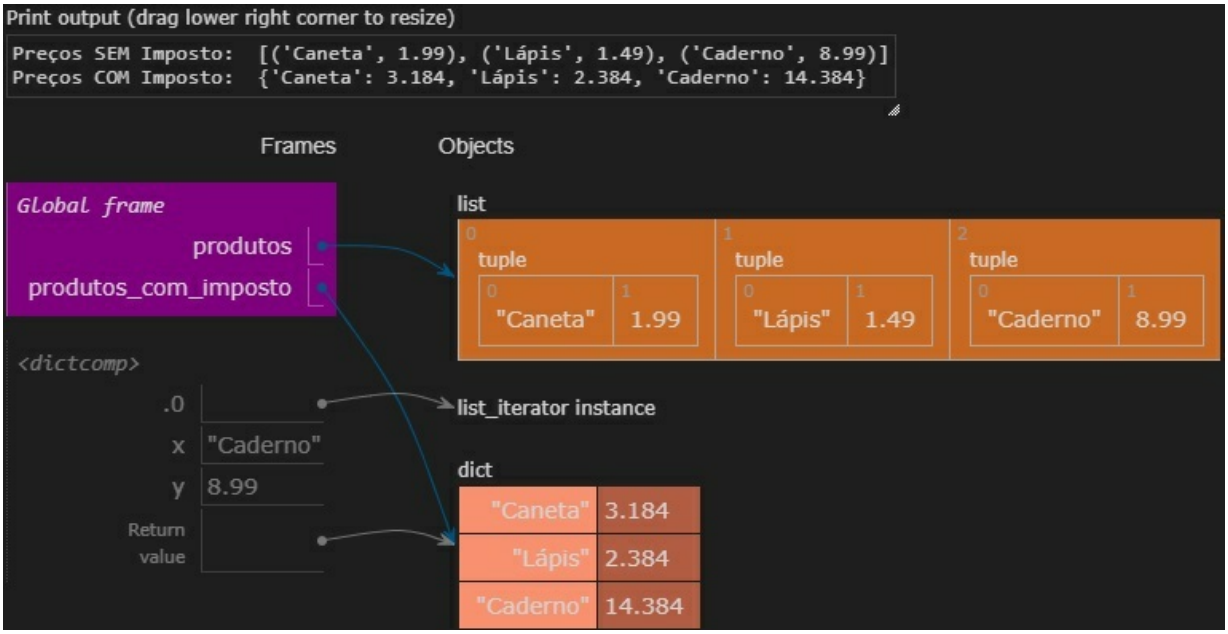
Supondo que tivéssemos de aplicar um determinado valor de imposto sobre os valores de tais itens, agora sim temos uma boa justificativa para o uso de dict comprehension.

Para esse fim é criada a variável produtos_com_imposto, que por sua vez recebe uma expressão que percorre cada elemento de produtos

multiplicando, quando possível, seu valor por 1.6.

```
Preços SEM Imposto:  [('Caneta', 1.99), ('Lápis', 1.49), ('Caderno', 8.99)]
Preços COM Imposto:  {'Caneta': 3.184, 'Lápis': 2.384, 'Caderno': 14.384}
```

Como retorno das funções print( ) temos os valores iniciais e os valores após a aplicação do imposto fictício. Note que por meio da expressão usada em nossa comprehension, apenas os elementos de tipo int e float foram submetidos a operação aritmética, mantendo as strings ordenadas sem alteração.



EXPRESSÕES TERNÁRIAS

Expressões ternárias na prática se assemelham muito com as comprehensions usadas anteriormente tanto em listas quanto em dicionários, porém sua aplicação real se dá quando necessitamos realizar certos tipos de validação usando condicionais.

Embora não exista especificamente comprehensions para estruturas condicionais, implicitamente podemos perfeitamente usar de expressões para para obter retornos em formato booleano.

```
1  num = 2
2
3  'Positivo' if num >= 0 else 'Negativo'
4
```

Repare que aqui, apenas para fins de exemplo, é criada uma variável de nome num que recebe atribuída para si o número int 2.

Em seguida é criada uma expressão ternária, que retornará ‘Positivo’ se o valor de num for igual ou maior que 0, caso contrário, retornará ‘Negativo’.

Apenas por curiosidade, preste atenção a sintaxe, aqui a expressão é um simples validador, não está atribuída a nenhuma variável e possui em sua composição dois possíveis retornos com base em uma estrutura condicional, tudo reduzido a uma “comprehension”.

```
'Positivo'
```

Como esperado, o retorno é ‘Positivo’, uma vez que a primeira condição é verdadeira pois o valor de num é maior que 0.

GERADORES E ITERADORES

Antes de entender de fato como funcionam os métodos geradores em Python, vamos dar uma breve recapitulada no conceito de objeto iterável.

Se você chegou a este ponto, certamente sabe que quando estamos falando de iterações estamos falando, por mais redundante que isso seja, de interações.

Logo, um objeto iterável é aquele tipo de objeto ao qual podemos interagir com seus elementos, cada elemento de sua composição.

```
1  minha_lista = [1,2,3,4]
2
3  print(hasattr(minha_lista, '__iter__'))
4
5  for i in minha_lista:
6      print(i)
7
```

Aqui, por exemplo, temos uma variável de nome minha_lista que, como esperado, recebe como atributo uma lista de elementos.

Por meio da função hasattr( ) podemos verificar se internamente minha_lista possui o método __iter__, lembrando que muita coisa em Python é implícita, pré-construída e pré-carregada automaticamente assim que criamos qualquer variável/objeto, classe, método/função.

Todo objeto iterável em Python internamente possui em sua composição o método __iter__ como um sinalizador para que o interpretador permita o instanciamento e uso de todo e qualquer dado deste objeto.

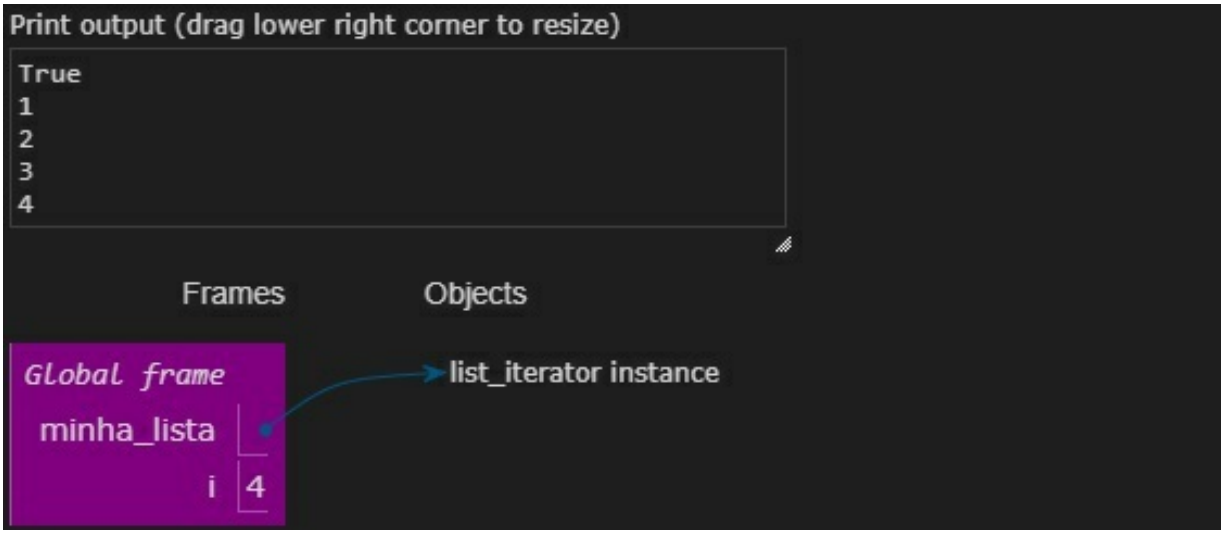
Sendo assim, aproveitando esse exemplo, como minha_lista é um objeto iterável, podemos usar do laço for para que realizemos a leitura de cada um dos elementos que compõem minha_lista.

```
True
1
2
3
4
```

Como era de se esperar, como retorno para hasattr( ) temos True, seguido do retorno do laço for, mostrando a cada laço um elemento de minha_lista.

```
1  minha_lista = [1,2,3,4]
2  minha_lista = iter(minha_lista)
3
```

Lembrando que um iterável por si só é um tipo de dado, em determinados casos, podemos definir manualmente que um certo objeto é um iterador por meio de iter( ).



Geradores e Memória

Como já mencionado algumas vezes ao longo de outros capítulos, devemos sempre prezar pela legibilidade de nosso código e pela performance do mesmo.

Um ponto importante a salientar aqui é que todo e qualquer objeto em Python ocupará um espaço alocado em memória com tamanho dinamicamente proporcional ao seu conteúdo.

Quando declaramos um determinado tipo de dado manualmente em Python, o mesmo tende a ter um tamanho diferente de quando declarado automaticamente, isto se dá porque ao especificar manualmente um determinado tipo de dado, implicitamente serão criadas, de acordo com o tipo de dado, alguns métodos para iteração do mesmo.

Em outras palavras, atribuir para uma variável qualquer, por exemplo, ‘Fernando’ e str(‘Fernando’) terão tamanhos diferentes, uma vez que str( ) como iterável carrega em sua estrutura interna alguns métodos ocultos do desenvolvedor, além é claro, dos dados/valores do objeto em si.

O uso de geradores, quando usados para volumes de dados consideráveis, serve não somente para poupar tempo de codificação mas alocação de memória, pois seus dados serão gerados e alocados conforme necessidade do código.

```
1  import sys
2
3  lista_manual = list([0,1,2,3,4,5,6,7,8,9,10,11,12,13,14])
4  lista_gerada = list(range(14))
5
6  listamilhao = list(range(1000001))
7
8  print('Tamanho em bytes:', sys.getsizeof(lista_manual))
9  print('Tamanho em bytes:', sys.getsizeof(lista_gerada))
10
11 print('Tamanho em bytes:', sys.getsizeof(listamilhao))
12
```

Para que fique mais claro, vamos ao exemplo. Repare que foram criadas

duas variáveis lista_manual e lista_gerada, ambas listas com 15 elementos numéricos em sua composição.

```
Tamanho em bytes: 240
Tamanho em bytes: 232
```

Por meio da função print( ) parametrizada com sys.getsizeof( ) por sua vez parametrizada com cada uma das listas, podemos reparar a diferença de tamanho em bytes alocados em memória.

Na primeira linha do retorno temos o valor referente a lista_manual, na segunda linha, o valor em bytes de lista_gerada.

Aqui temos um exemplo bastante básico, porém já é possível notar que a mesma lista, quando criada via gerador, tem tamanho menor. Raciocine em grandes volumes de dados o quanto esta proporção de tamanho faria a diferença no impacto de performance de seu programa.

```
Tamanho em bytes: 240
Tamanho em bytes: 232
Tamanho em bytes: 9000120
```

Apenas como exemplo, também, note o tamanho em bytes de listamilhão, criada aqui fora de contexto apenas para demonstrar o tamanho de uma expressão de um valor de milhão.

Novamente, procure raciocinar na proporção do espaço de memória alocado de acordo com o tamanho do conteúdo da respectiva variável.

```
1  import sys
2
3  def gera_numero():
4      num = 0
5      while True:
6          yield num
7          num += 1
8
9  gerador = gera_numero()
10
```

Partindo para uma aplicação real, supondo que depende do contexto de nosso programa, precisássemos de um gerador de números sequenciais.

Para isso criamos a função gera_numero(), sem parâmetros mesmo. Dentro de seu corpo criamos a variável num inicialmente com valor zerado, na sequência criamos de fato o gerador para num por meio de yield.

Em outros momentos, como de costume, usávamos de return para retornar um determinado dado/valor para uma variável ao final da execução da função, aqui, no contexto de um gerador, return é substituído por yield, por fim num é atualizada com seu valor acrescido de 1.

Fora da função é criada a variável gerador que chama a função gera_numero( ).

```
1  import sys
2
3  def gera_numero():
4      num = 0
5      while True:
6          yield num
7          num += 1
8
9  gerador = gera_numero()
10
11 print(next(gerador))
12 print(next(gerador))
13 print(next(gerador))
14 print(next(gerador))
15 print(next(gerador))
16
```

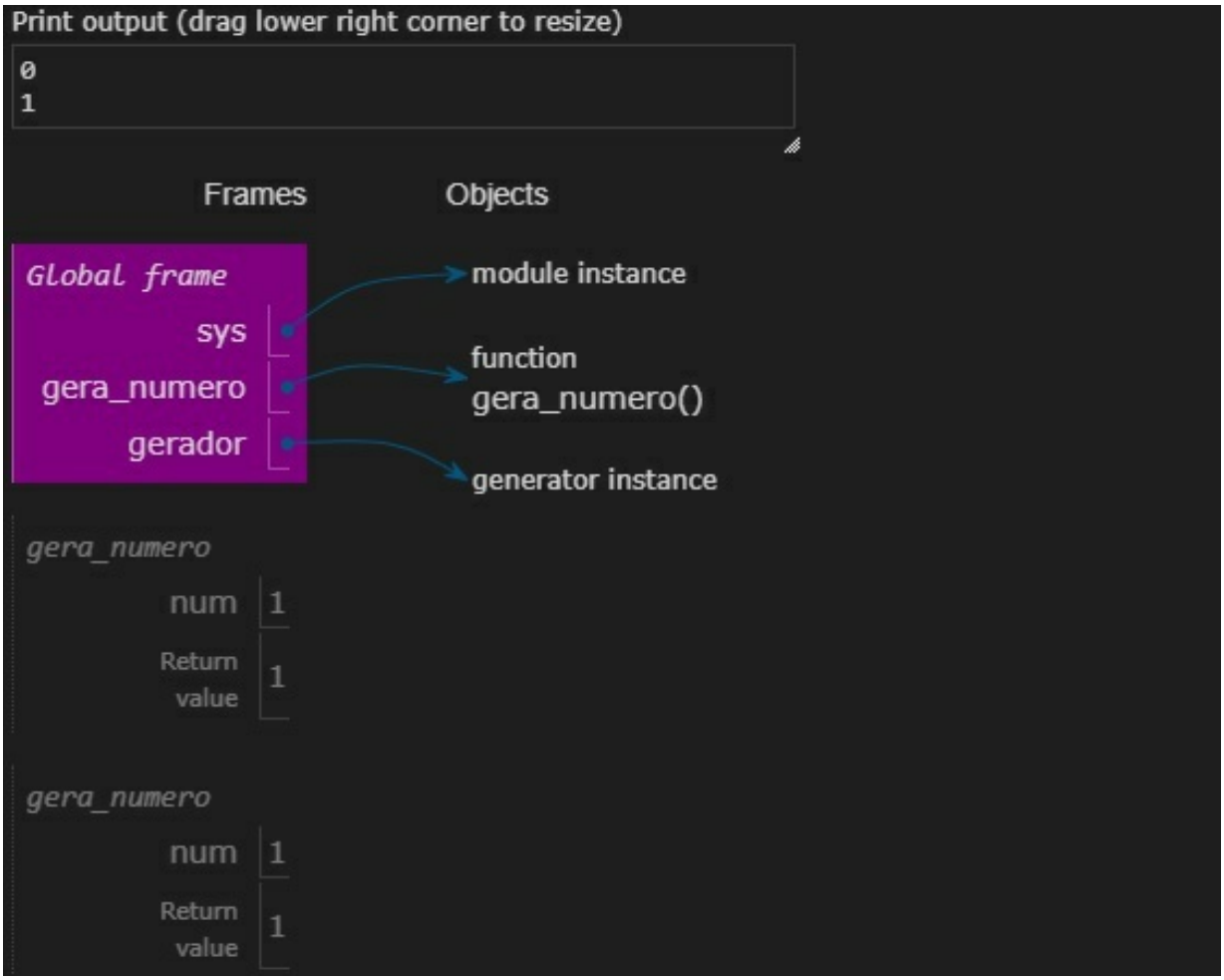
A partir desse momento, temos nosso gerador instanciado e em execução.

Por meio de nossa função print( ) parametrizada com next( ) por sua vez parametrizada com nosso gerador, a cada execução será gerado um número sequencial, conforme a demanda.

Dessa forma, usaremos apenas o necessário de números e apenas no tempo certo em que o restante do código exigir tal funcionalidade, poupando espaço e processamento.

```
0
1
2
3
4
```

O retorno, como esperado, foram 5 números sequenciais referentes aos 5 comandos print(next()) usados no exemplo.



ZIP

Muitas das vezes que estamos trabalhando com dados em formato de lista nos deparamos com situações onde precisamos unir listas de modo que possamos iterar sobre seus elementos como um todo.

Dentre as funcionalidades do Python que temos para essa finalidade, talvez a mais interessante delas seja a manipulação de listas por meio do método zip( ).

Por meio dessa função podemos realizar a união não somente de listas de mesmo tamanho, onde os elementos de uma serão equivalentes aos da outra, mas podemos perfeitamente realizar a união de listas de diferentes tamanhos sem que isso gere alguma exceção ou comportamento adverso em nossas estruturas de dado.

```
1  cidades = ['Porto Alegre', 'Curitiba', 'Salvador', 'Belo Horizonte']
2  estados = ['RS', 'PR', 'BH', 'MG']
3
4  cidades_estados = zip(cidades, estados)
5
```

Para esse exemplo criamos duas listas equivalentes, cada uma com 4 elementos em formato de string, atribuídas para as variáveis cidades e estados respectivamente.

Imediatamente fazendo o uso de zip( ), note que criamos uma terceira variável de nome cidades_estados, atribuída para si temos a função zip( ) por sua vez parametrizada com os dados de cidades e estados.

Nesse caso, onde cada lista possui um número específico de elementos e ambas possuem o mesmo número de elementos, o que a função zip( ) fará é simplesmente unir cada elemento de uma lista com outra, com base no valor de índice de cara elemento.

Em outras palavras, por meio da função zip( ) o retorno que teremos é a junção dos elementos de índice 0 em ambas listas, seguidos dos elementos de índice 1, seguidos dos elementos de índice 2, sucessivamente por quantos elementos existirem.

```
('Porto Alegre', 'RS')
('Curitiba', 'PR')
('Salvador', 'BH')
('Belo Horizonte', 'MG')
```

Preste atenção no retorno, pois há uma particularidade aqui, em função da equivalência de elementos, os mesmos são colocados unidos entre parênteses.

```
6  print(type(cidades_estados))
7
```

Veja que interessante, uma vez criada nossa variável cidades_estados, podemos iterar sobre os dados da mesma, mas antes disso, vamos realizar uma simples verificação quanto ao tipo de dado de cidades_estados.

```
<class 'zip'>
('Porto Alegre', 'RS')
('Curitiba', 'PR')
('Salvador', 'BH')
('Belo Horizonte', 'MG')
```

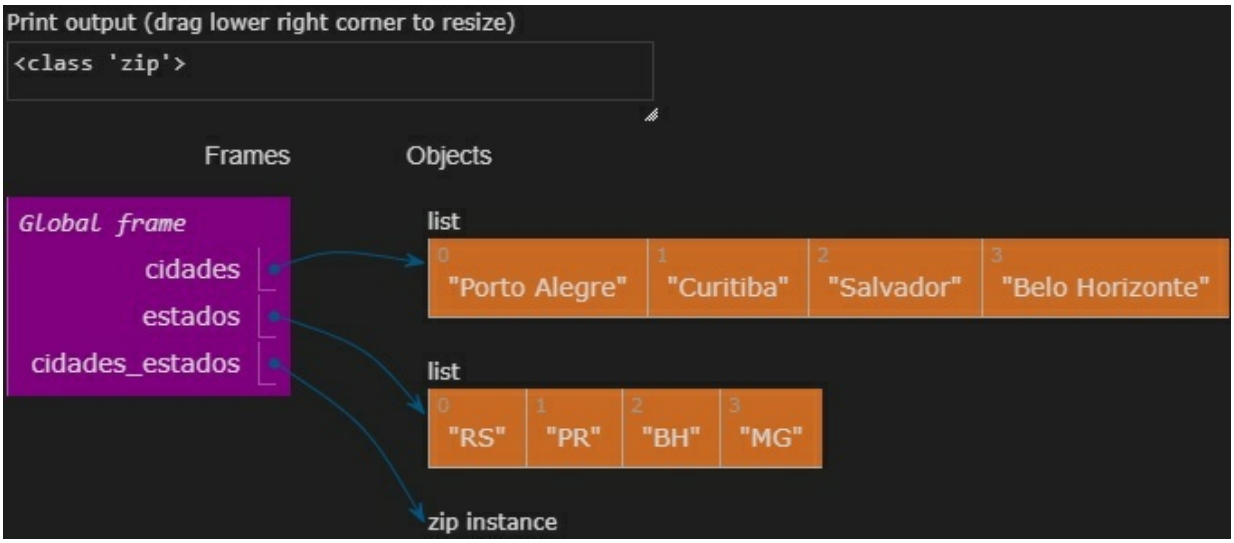
Nesse caso, o que nos é retornado é que o tipo de dado em questão, para nossa variável cidades_estados, é do tipo zip. Em outras palavras, por meio da função zip( ) temos o tipo de dado zip.

```
8  for i in cidades_estados:
9      print(i)
0
```

Realizando uma simples varredura sobre os elementos de cidades_estados, podemos verificar que a estrutura geral dessa variável é um objeto do tipo zip, porém internamente, como é de se esperar, cada elemento é uma tupla independente.

```
<class 'zip'>
('Porto Alegre', 'RS')
<class 'tuple'>
('Curitiba', 'PR')
<class 'tuple'>
('Salvador', 'BH')
<class 'tuple'>
('Belo Horizonte', 'MG')
<class 'tuple'>
```

Sendo assim, devemos sempre levar em consideração a real necessidade da união de listas, pois nesse processo ocorre não somente a união mas a conversão dos tipos de dados, limitando um pouco as possibilidades de manipulação dos mesmos.



Zip longest

Se você entendeu a lógica do método zip( ), deve ter reparado que tal método trabalha limitado a união de listas com equivalência de elementos, ou seja, necessariamente precisávamos de listas com o mesmo número de elementos para que os mesmos pudessem ser convertidos.

Porém, trazendo tal conceito para algo mais próximo da realidade, muitas das vezes nos deparamos justamente com o contrário, com situações onde as listas que precisamos parear terão número e elementos diferentes.

Nesses casos em particular, podemos fazer o uso do método zip_longest( ) da biblioteca itertools.

```
1  from itertools import zip_longest
2
3  cidades = ['Porto Alegre', 'Curitiba', 'Salvador',
4             'Belo Horizonte', 'Rio de Janeiro', 'Goiânia']
5  estados = ['RS', 'PR', 'BH', 'MG']
6
7  cidades_estados = zip_longest(cidades, estados)
8  cidades_estados2 = zip_longest(cidades, estados,
9                                 fillvalue='Desconhecido')
10
```

Como zip_longest( ) em questão não é uma função nativa do Python, inicialmente precisamos realizar a importação da mesma para nosso código. Isso é feito pelo comando from itertools import zip_longest, uma vez que precisamos apenas dessa função e não de toda a biblioteca itertools.

Na sequência criamos uma variável de nome cidades que recebe uma lista com alguns nomes de cidade em forma de string. Da mesma forma criamos uma variável de nome estados que recebe por sua vez uma lista com algumas siglas para estados brasileiros.

Note que aqui não existe uma equiparação de elementos, haja visto que a lista de cidades possui mais elementos que a lista de estados.

Para esses casos, onde queremos realizar a união de listas de diferentes tamanhos, a função adequada é a zip_longest( ).

Sendo assim, inicialmente criamos uma variável de nome cidades_estados que recebe como atributo a função zip_longest( ) parametrizada com os dados de cidades e de estados.

```
('Porto Alegre', 'RS')
('Curitiba', 'PR')
('Salvador', 'BH')
('Belo Horizonte', 'MG')
('Rio de Janeiro', None)
('Goiânia', None)
```

Visualizando tais dados é possível perceber que a união das listas foi feita, sem gerar exceções, porém, onde havia discrepância de elementos, o “espaço vazio” foi preenchido automaticamente com None.

None por sua vez é uma palavra reservada ao sistema para sinalizar que justamente naquela posição do código não existe nenhum dado/valor.

Por meio da função zip_longest( ) podemos contornar isso facilmente através do parâmetro fillvalue, atribuindo para o mesmo algum dado ou valor a ser inserido onde for um espaço vazio.

Realizando este pequeno tratamento, replicamos o mesmo código, porém agora atribuído para cidades_estados2, fazendo o uso de fillvalue = ‘Desconhecido’.

```
('Porto Alegre', 'RS')
('Curitiba', 'PR')
('Salvador', 'BH')
('Belo Horizonte', 'MG')
('Rio de Janeiro', 'Desconhecido')
('Goiânia', 'Desconhecido')
```

O retorno, como esperado, é um objeto zip com suas respectivas tuplas de elementos, onde os elementos anteriormente faltantes agora possuem a string ‘Desconhecido’ em sua composição.


Frames
Objects
Global frame
zip_longest
cidades
estados
cidades_estados
cidades_estados2
zip_longest class
show attributes
list
0 "Porto Alegre"
1 "Curitiba"
2 "Salvador"
3 "Belo Horizonte"
4 "Rio de Janeiro"
5 "Goiânia"
list
0 "RS"
1 "PR"
2 "BH"
3 "MG"
zip_longest instance
zip_longest instance

COUNT

Quando se estuda os fundamentos de uma linguagem de programação, um dos conceitos presentes é o de contadores ou incremento de valores dependendo do contexto.

Avançando um pouco dentro desses conceitos, visando apenas otimizar a performance de execução de nosso código, podemos fazer o uso da ferramenta count( ).

```
1  from itertools import count
2  contador = count()
3
```

Na prática, como tal módulo não é nativo do Python, mas compõe a biblioteca itertools, precisamos realizar a importação do mesmo. Como já feito outras vezes, a importação é realizada de forma simples por meio do comando from itertools import count.

Lembrando que como iremos usar apenas dessa função da biblioteca itertools, uma vez importado o módulo/pacote, podemos usar o mesmo diretamente, sem precisar instanciar a biblioteca em si.

Note que é criada a variável contador que simplesmente chama a função count( ), sem parâmetros mesmo, atribuída para si,

```
4  for numero in contador:
5      print(numero)
6      if numero >= 10:
7          break
8
```

Uma vez que temos uma variável chamando a função, podemos a partir disso realizar qualquer tipo de interação sobre a mesma, internamente todo e qualquer código que iterar com essa variável estará iterando com count( ).

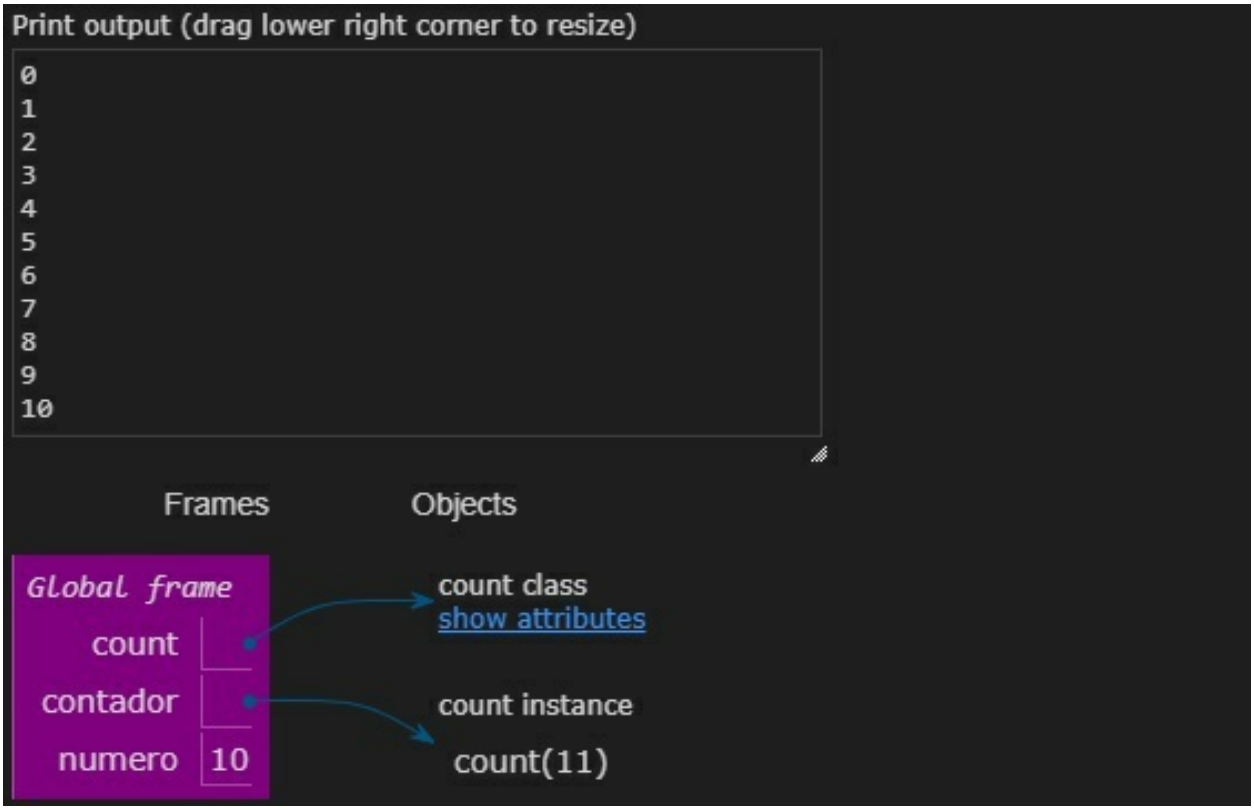
Sendo assim, podemos criar um laço de repetição bastante básico, onde iremos interagir com a variável contador.

Repare que no corpo desse laço for simplesmente é exibido o

número em questão que está sendo gerado e processado pelo contador, também é criada uma simples estrutura condicional dizendo que se o número gerado for igual ou maior que 10, é para que o processamento seja interrompido.



O retorno é, como esperado, 11 números sequenciais, uma vez que o contador sempre é por padrão inicializado em 0.



Por meio do pré-visualizador podemos notar algo interessante, diferentemente do método tradicional incrementando uma variável, onde a

cada laço é gerada uma nova instância da variável alocando espaço em memória, aqui temos um espaço alocado dinamicamente atualizado a cada laço.

Como dito anteriormente, a justificativa de se usar tal método é simplesmente a otimização de performance de nosso código. Em geral, uma boa prática de programação é dedicar uma fase de polimento a nossos programas, de modo a, se for o caso, até mesmo reescrever certos blocos de código quando for possível um ganho de performance.

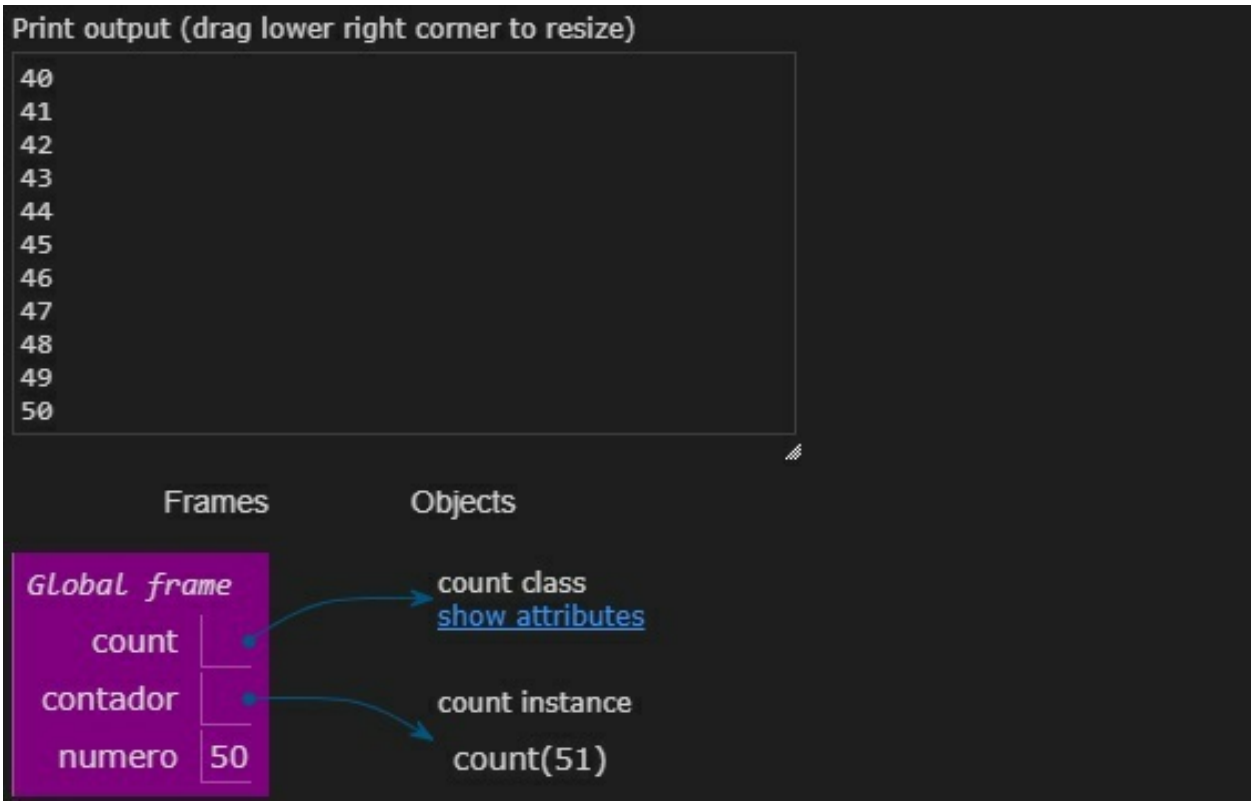
```
1  from itertools import count
2  contador = count(start = 40)
3
4  for numero in contador:
5    print(numero)
6    if numero >= 50:
7      break
8
```

Como mencionado anteriormente, por padrão todo e qualquer contador tem sua inicialização contado números a partir do zero, porém, é perfeitamente possível via função count( ) definir um marco inicial manualmente, e isso é feito por meio do parâmetro start seguido do valor a ser usado como referência.

```
40
41
42
43
44
45
46
47
48
49
50
```

Adaptado o exemplo anterior para agora iniciar em 40 e contar até que o valor seja igual ou maior que 50, o resultado é uma sequência de números

dentro desse intervalo.

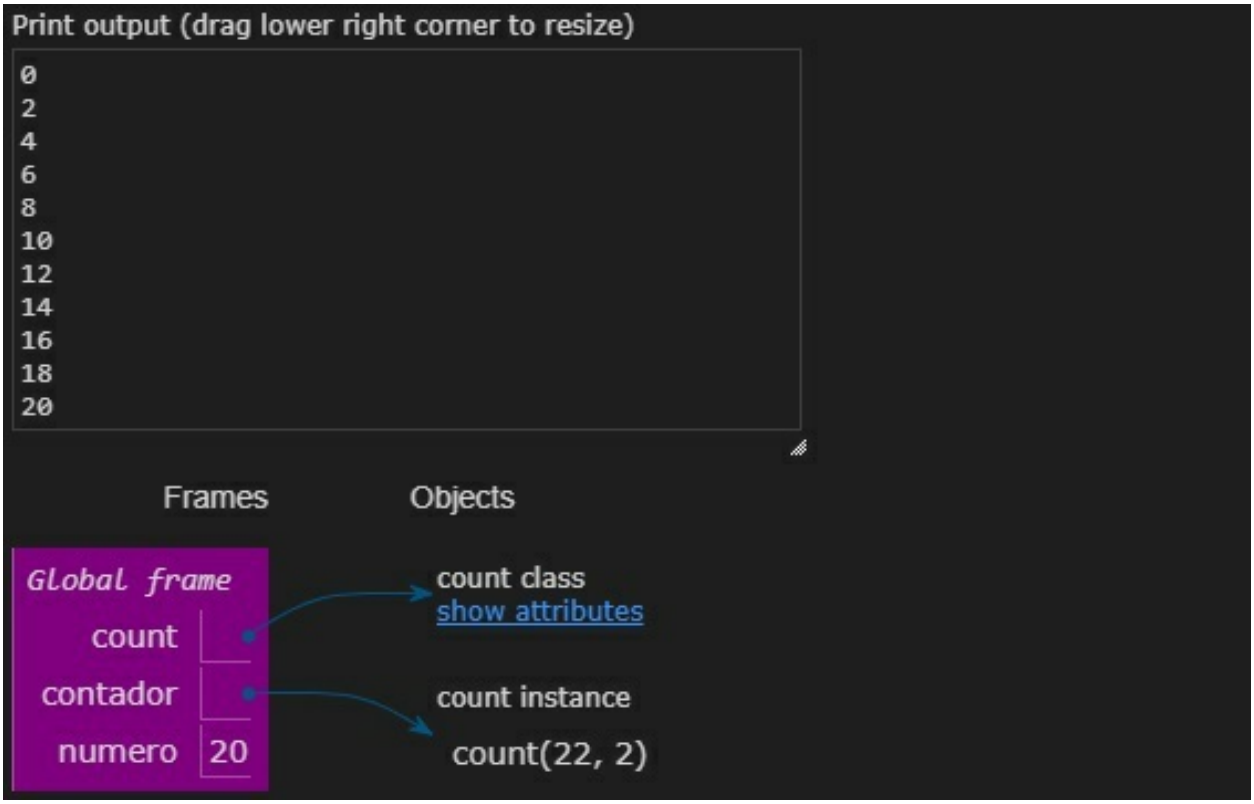



```
1  from itertools import count
2  contador = count(start = 0, step = 2)
3
4  for numero in contador:
5      print(numero)
6      if numero >= 20:
7          break
8
```

Explorando uma outra funcionalidade muito utilizada rotineiramente, ao fazer o uso de nosso contador via count( ) também podemos definir de quantos em quantos números será feita a devida contagem.

```
0
2
4
6
8
10
12
14
16
18
20
```

Como esperado, o retorno é uma sequência de números de 0 a 20, contados de 2 em 2 números.



MAP

Uma das funcionalidades mais interessantes nativas do Python, desconhecida por boa parte dos usuários, principalmente os mais iniciantes, é a função map( ).

Por meio da função map( ) podemos aplicar uma determinada função a cada elemento ou linha por linha de uma lista/tupla/dicionário.

Isso é útil quando queremos extrair o máximo de performance de um código, reduzindo e otimizando o mesmo até se comparado com uma list ou dict comprehension.

Map iterando sobre uma lista

```
1  lista = [1,2,3,4,5,6,7,8,9,10]
2
3  nova_lista = map(lambda x: x * 2, lista)
4
5  # Mesmo que:
6  nova_lista2 = [x * 2 for x in lista]
7
```

Vamos ao exemplo, inicialmente é criada uma variável de nome lista que recebe, como esperado, uma lista com uma série de elementos atribuída para si.

A partir deste ponto já podemos iterar sobre os elementos dessa lista, e já faremos isso por meio de map( ).

Para isso é criada uma variável de nome nova_lista que chama a função map( ), parametrizando a mesma com uma expressão lambda que pega cada elemento dessa lista e simplesmente multiplica seu valor por 2.

Apenas para fins de comparação, também é criada a variável nova_lista2 que realiza o mesmo procedimento porém usando de uma estrutura de list comprehension.

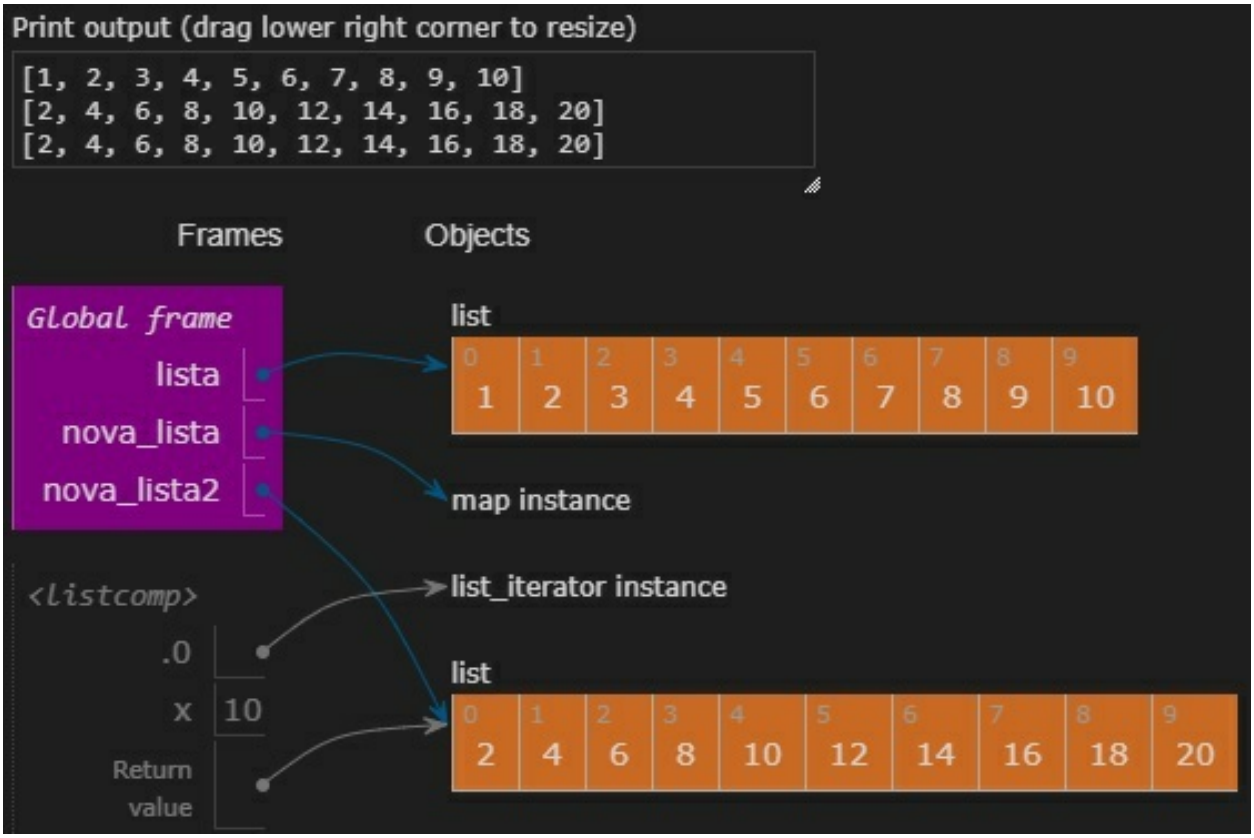
```
 8  print(lista)
 9  print(list(nova_lista))
10
11  print(nova_lista2)
12
```

Como feito até então, haja visto que não estamos retornando nada, mas simplesmente exibindo resultados de exemplos, por meio da função print( ) podemos verificar os resultados.

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

Como esperado, os devidos resultados de nossas funções print( ) onde, na primeira linha temos o resultado para lista, na segunda linha para nova_lista e na terceira linha para nova_lista2.

Novamente estamos trabalhando com uma questão de otimização, raciocine que a iteração realizada via função map( ) possui uma performance superior quando comparado aos meios convencionais e até mesmo, como colocado no exemplo acima, o uso de list comprehension, haja visto que estes métodos usam de um laço for para percorrer cada elemento de nossa lista inicial, enquanto a função map( ) realiza esse procedimento por meio de uma função interna melhor otimizada.



Map iterando sobre um dicionário

```
produtos = [
            {'nome': 'item 1', 'preco': 32},
            {'nome': 'item 2', 'preco': 23},
            {'nome': 'item 3', 'preco': 12},
            {'nome': 'item 4', 'preco': 10},
            {'nome': 'item 5', 'preco': 55},
]

precos = map(lambda p: p['preco'], produtos)

for preco in precos:
  print(preco)

```

Seguindo com nosso raciocínio, como dito anteriormente, uma das usabilizades de nossa função map( ) é poder iterar sobre um elemento em específico de uma lista ou dicionário. Aqui para fins de exemplo vamos misturar as duas coisas.

Inicialmente é criada uma variável de nome produtos, que por sua vez recebe uma lista onde cada elemento da mesma é um par de dados em formato de dicionário.

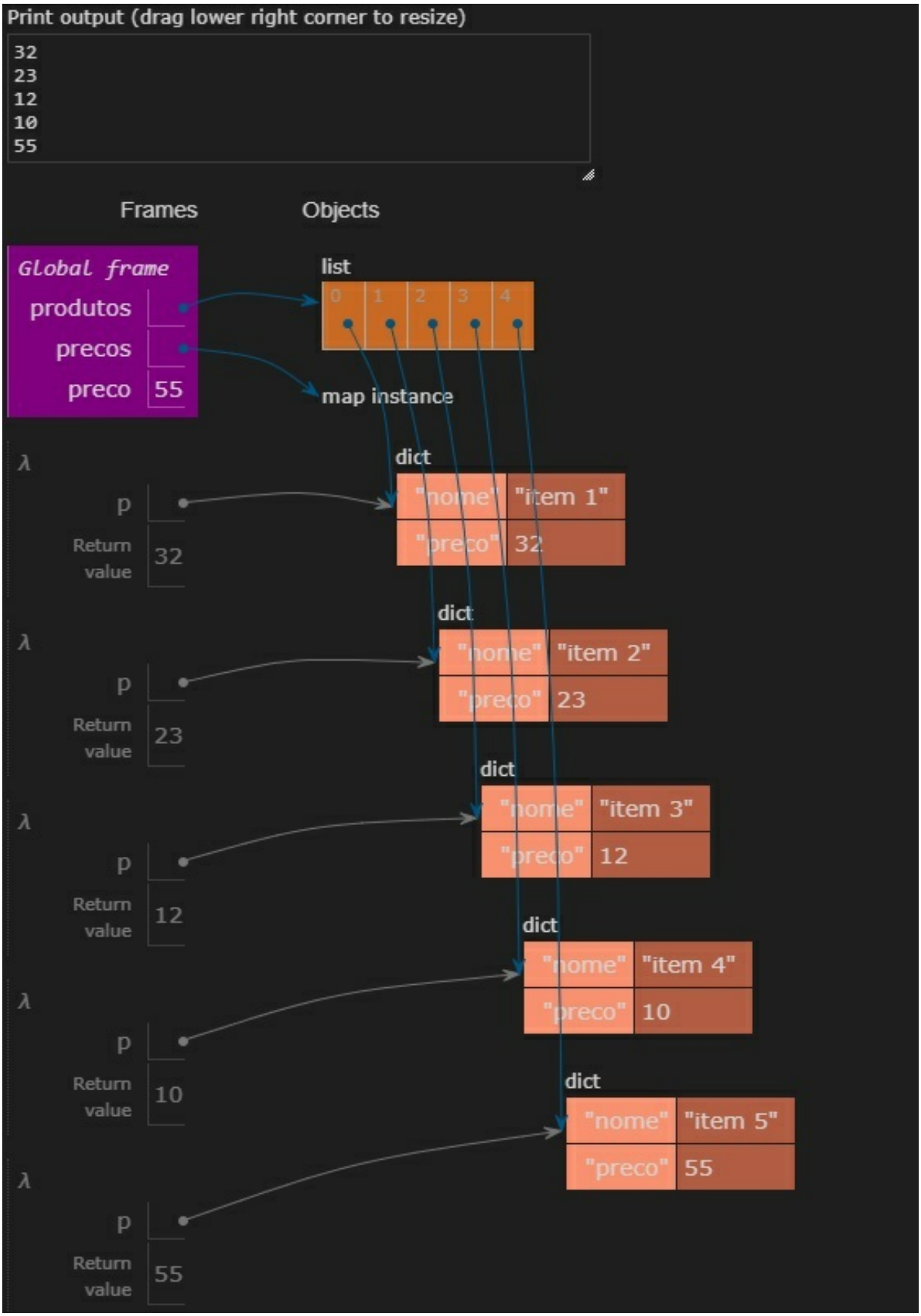
Através dos meios convencionais teríamos que iterar sobre os dados fazendo referência de seus índices e subdivisões, o que na prática é um tanto quanto confuso, e muito suscetível a erros.

Aqui simplesmente criamos uma variável de nome precos que chama a função map( ) parametrizando a mesma com uma expressão lambda, lendo apenas os dados/valores atribuídos para ‘preco’ dos dados de produtos.

Por fim é criado um laço for, agora somente para que possamos exibir tais valores.



E o retorno são os respectivos valores encontrados em ‘precos’ de cada elemento que compõe produtos.


Print output (drag lower right corner to resize)
32
23
12
10
55
Frames
Objects
Global frame
produtos
precos
preco
55
list
0
1
2
3
4
map instance
λ
p
Return value
32
dict
"nome"
"item 1"
"preco"
32
λ
p
Return value
23
dict
"nome"
"item 2"
"preco"
23
λ
p
Return value
12
dict
"nome"
"item 3"
"preco"
12
λ
p
Return value
10
dict
"nome"
"item 4"
"preco"
10
λ
p
Return value
55
dict
"nome"
"item 5"
"preco"
55

```
1  def aumenta_precos(p):
2      p['preco'] = p['preco'] * 1.05
3      return p
4
5  precos2 = map(aumenta_precos, produtos)
6
7  for produto in precos2:
8      print(produto)
9
```

Apenas concluindo nossa linha de raciocínio, dentre a gama de possibilidades que temos quando fazendo o uso da função map( ), uma das práticas mais comuns, também visando performance, é a de parametrizar a mesma com uma função independente.

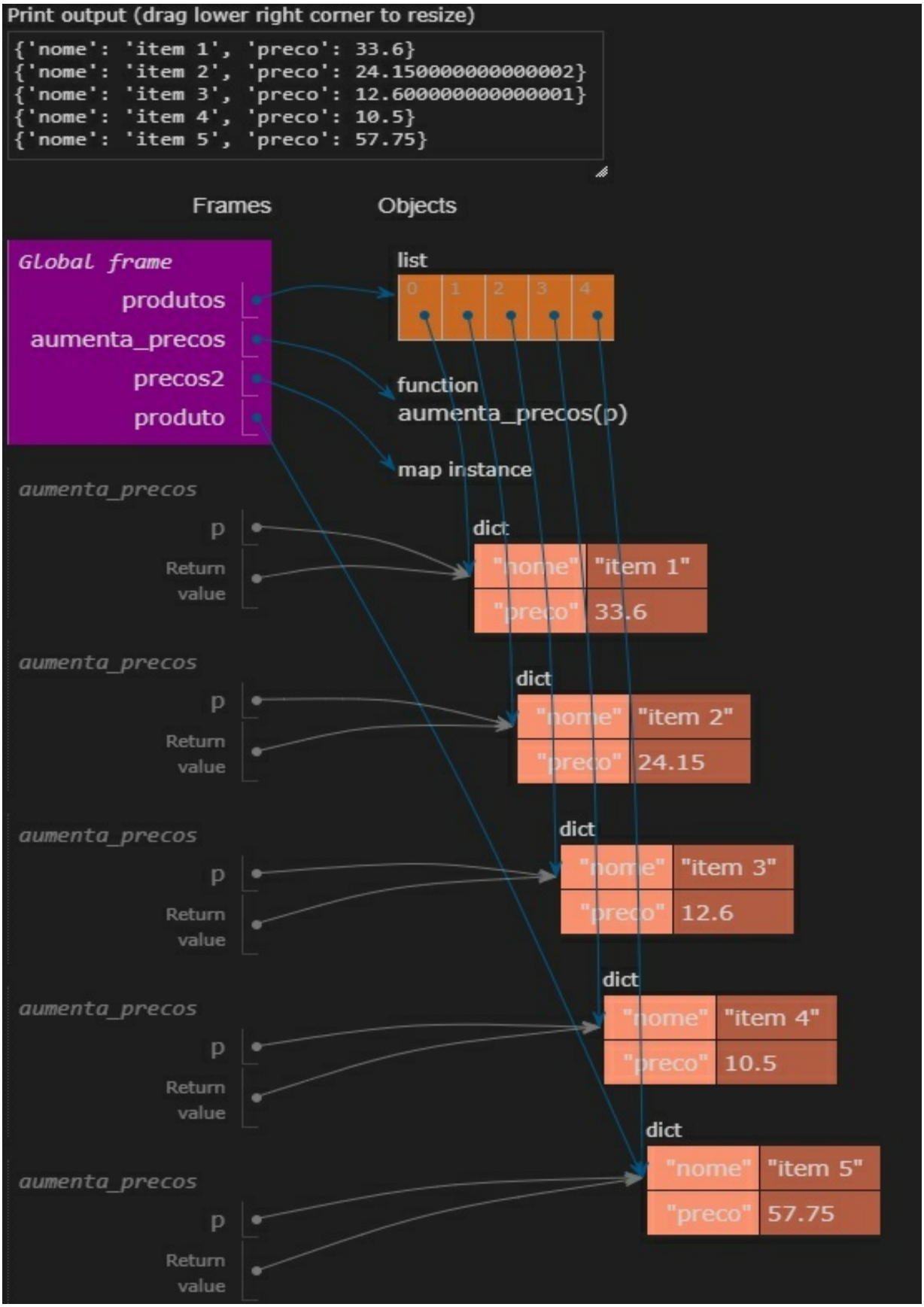
Por exemplo, inicialmente criamos uma função de nome aumenta_precos( ) que recebe um preço para p. Dentro do corpo dessa função existe uma simples expressão para pegar o valor existente em ‘preco’ e o multiplicar por 1.05, retornando esse valor.

Na sequência é criada a variável precos2, que por sua vez chama a função map( ) parametrizando a mesma com a função aumenta_precos e com a variável de onde os dados serão lidos, neste caso, produtos.

Por fim criamos aqui um laço for somente para exibir tais valores.

```
{'nome': 'item 1', 'preco': 33.6}
{'nome': 'item 2', 'preco': 24.150000000000002}
{'nome': 'item 3', 'preco': 12.600000000000001}
{'nome': 'item 4', 'preco': 10.5}
{'nome': 'item 5', 'preco': 57.75}
```

Como esperado, iterando apenas sobre os dados/valores em ‘precos’ simulamos a atualização dos preços de alguns determinados itens.


Print output (drag lower right corner to resize)
{'nome': 'item 1', 'preco': 33.6}
{'nome': 'item 2', 'preco': 24.150000000000002}
{'nome': 'item 3', 'preco': 12.600000000000001}
{'nome': 'item 4', 'preco': 10.5}
{'nome': 'item 5', 'preco': 57.75}
Frames
Objects
Global frame
produtos
aumenta_precos
precos2
produto
list
0
1
2
3
4
function
aumenta_precos(p)
map instance
aumenta_precos
p
Return value
dict
"nome"
"item 1"
"preco"
33.6
aumenta_precos
p
Return value
dict
"nome"
"item 2"
"preco"
24.15
aumenta_precos
p
Return value
dict
"nome"
"item 3"
"preco"
12.6
aumenta_precos
p
Return value
dict
"nome"
"item 4"
"preco"
10.5
aumenta_precos
p
Return value
dict
"nome"
"item 5"
"preco"
57.75

FILTER

Entendida a lógica da função map( ) podemos partir para a função filter( ), pois ambas são de propósitos parecidos mas de aplicações diferentes.

Vimos anteriormente que por meio da função map( ) podemos de forma reduzida e eficiente iterar sobre algum elemento específico de uma lista ou dicionário.

Pois bem, por meio da função filter( ) também realizaremos iterações sobre um elemento de uma lista ou dicionário, porém usando de lógica condicional.

Em outras palavras, via map( ) podemos realizar toda e qualquer iteração sobre um determinado elemento, enquanto via filter( ) literalmente filtraremos um determinado dado/valor de um elemento quando o mesmo respeitar uma certa condição imposta.

Internamente os dados os quais serão reaproveitados pela função serão todos que, de acordo com a condição, retornem True.

```
1  pessoas = [
2              {'nome': 'Ana', 'idade': 22},
3              {'nome': 'Maria', 'idade': 72},
4              {'nome': 'Paulo', 'idade': 55},
5              {'nome': 'Pedro', 'idade': 68},
6              {'nome': 'Rafael', 'idade': 99},
7              {'nome': 'Tania', 'idade': 18},
8  ]
9
10 idosos = filter(lambda x: x['idade'] > 70, pessoas)
11
12 print(list(idosos))
13
```

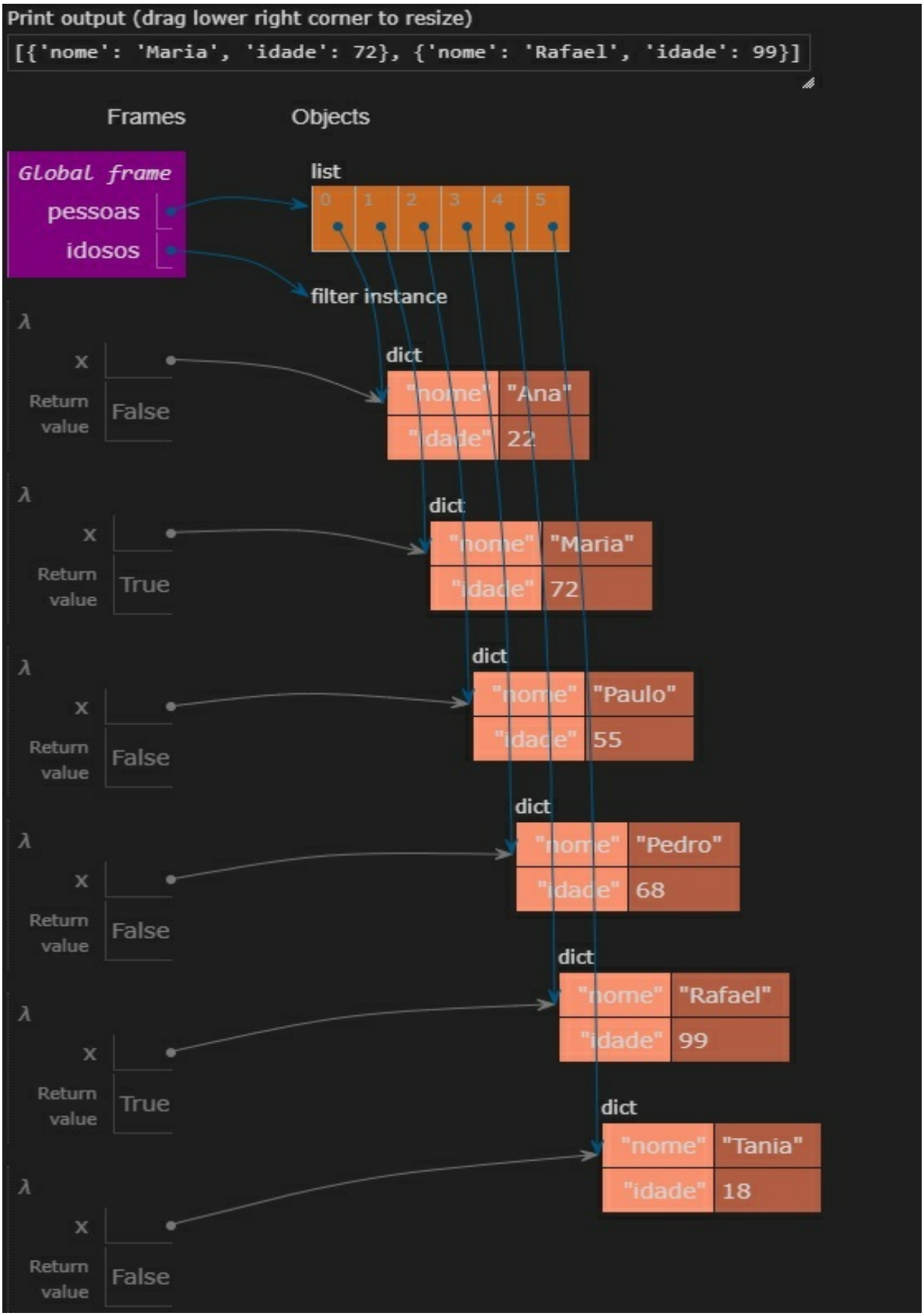
Partindo para prática, inicialmente criamos uma variável de nome pessoas, atribuída para a mesma está uma lista onde cada elemento é um par de dados em formato de dicionário, muito parecido com os exemplos usados anteriormente.

Em seguida criamos uma variável de nome idosos, que chama a função filter( ) parametrizando a mesma com uma expressão lambda que basicamente irá ler os dados/valores de ‘idade’, mas apenas fazendo o uso dos quais tiverem este valor superior a 70.

Em seguida é repassado como segundo parâmetro a variável de onde serão extraídos tais dados, nesse caso, pessoas.

```
[{'nome': 'Maria', 'idade': 72}, {'nome': 'Rafael', 'idade': 99}]
```

Analisando o retorno, note que foram retornados apenas os elementos os quais em seu campo ‘idade’ o valor era maior que 70.


Print output (drag lower right corner to resize)
[{'nome': 'Maria', 'idade': 72}, {'nome': 'Rafael', 'idade': 99}]
Frames
Objects
Global frame
pessoas
idosos
list
0 1 2 3 4 5
filter instance
λ
x
Return value
False
dict
"nome" "Ana"
"idade" 22
λ
x
Return value
True
dict
"nome" "Maria"
"idade" 72
λ
x
Return value
False
dict
"nome" "Paulo"
"idade" 55
λ
x
Return value
False
dict
"nome" "Pedro"
"idade" 68
λ
x
Return value
True
dict
"nome" "Rafael"
"idade" 99
λ
x
Return value
False
dict
"nome" "Tania"
"idade" 18

```
produtos2 = [
                {'nome': 'item 1', 'preco': 32},
                {'nome': 'item 2', 'preco': 23},
                {'nome': 'item 3', 'preco': 62},
                {'nome': 'item 4', 'preco': 10},
                {'nome': 'item 5', 'preco': 55},
]

def filtra(p):
  if p['preco'] > 50:
    return True

nova_lista3 = filter(filtra, produtos2)
for produtox in nova_lista3:
  print(produtox)

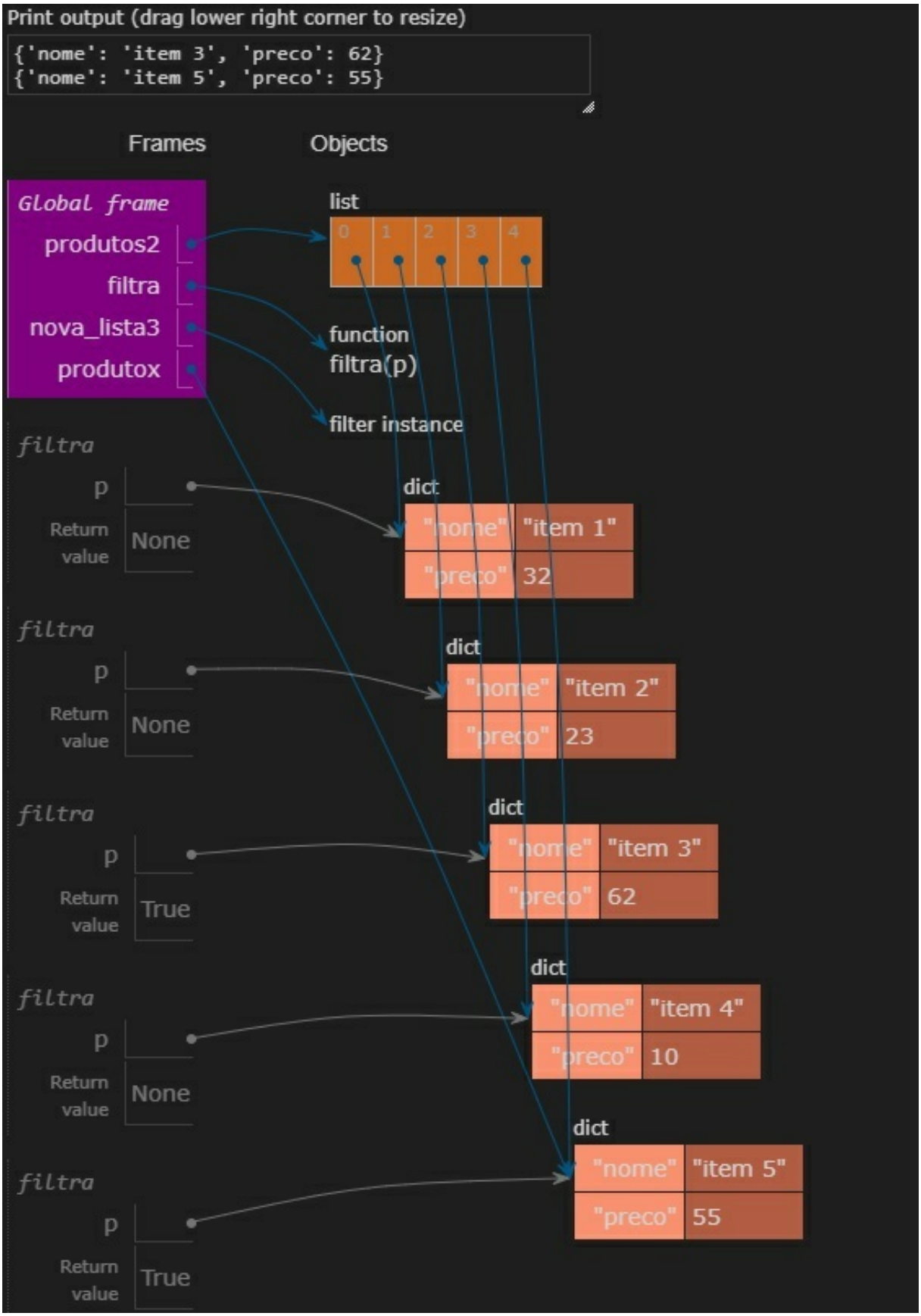
```

Assim como demonstrado em map( ), a função filter( ) também suporta que para a mesma seja repassado como parâmetro uma função independente.

Neste outro exemplo, realizando a filtragem de valores superiores a 50, note que para a variável nova_lista3 temos a função filter( ) por sua vez parametrizada com a função filtra, lendo dados de produtos2.

```
{'nome': 'item 3', 'preco': 62}
{'nome': 'item 5', 'preco': 55}
```

Nesse caso, o retorno nos remete aos dois elementos os quais para o campo ‘preco’ o valor atribuído era maior que 50.


Print output (drag lower right corner to resize)
{'nome': 'item 3', 'preco': 62}
{'nome': 'item 5', 'preco': 55}
Frames
Objects
Global frame
produtos2
filtra
nova_lista3
produtox
list
0
1
2
3
4
function
filtra(p)
filter instance
filtra
p
Return value
None
dict
"nome"
"item 1"
"preco"
32
filtra
p
Return value
None
dict
"nome"
"item 2"
"preco"
23
filtra
p
Return value
True
dict
"nome"
"item 3"
"preco"
62
filtra
p
Return value
None
dict
"nome"
"item 4"
"preco"
10
filtra
p
Return value
True
dict
"nome"
"item 5"
"preco"
55

REDUCE

Finalizando essa parte onde exploramos algumas possibilidades mais avançadas de tratamento de dados a partir de listas, vamos entender uma função que é bastante utilizada, a chamada função reduce( ).

```
1  from functools import reduce
2
```

A função reduce( ) por si só não é nativa das bibliotecas pré-carregadas em Python, sendo assim, é necessário realizar a importação desse módulo/pacote da biblioteca functools, isso é feito pelo simples comando from functools import reduce.

```
 1  from functools import reduce
 2
 3  def soma(num1, num2):
 4      return num1 + num2
 5
 6  lista = [1,2,3,4,5,6,7,8,9,10]
 7
 8  resultado = reduce(soma, lista)
 9  print(resultado)
10
```

Em seguida criamos uma simples função soma, que recebe dois números e retorna a soma dos mesmos, até aqui nada de novo.

Na sequência vamos reutilizar a variável lista que usamos em map( ), mas dessa vez, criamos uma variável de nome resultado que chama a função reduce( ) repassando como parâmetros para a mesma nossa função soma seguido de lista.

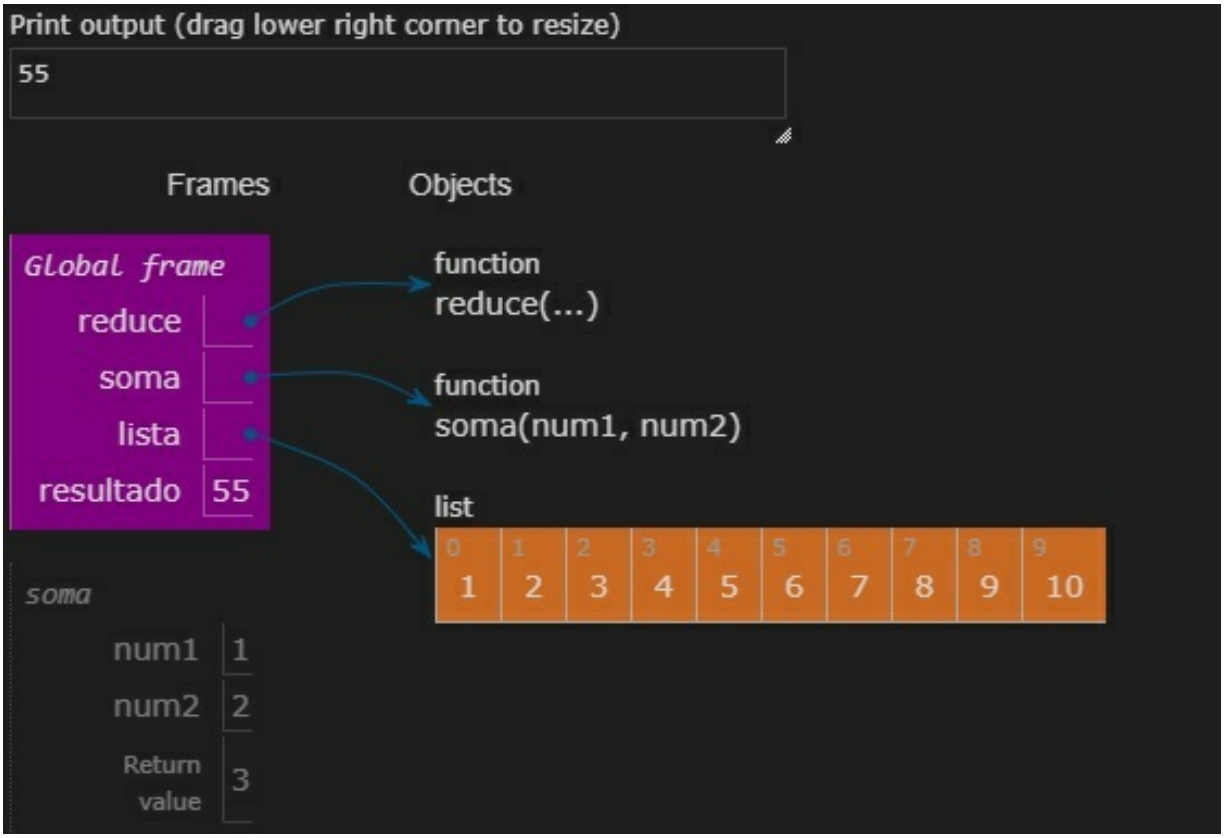
Vamos procurar entender o que de fato acontece aqui. A função reduce( ) em si normalmente é utilizada para realizar o agrupamento dos dados de uma lista, em nosso exemplo, podemos realizar a soma de todos elementos da mesma obtendo um novo dado.

A função reduce( ) pode sua vez recebe sempre dois parâmetros, que função será utilizada e sobre quais dados será aplicada a função, respectivamente.

Nesse caso, a ideia é que seja realizada a soma do primeiro elemento de nossa lista com o segundo elemento, este resultado será somado com o terceiro elemento, este resultado será somado com o quarto elemento, repetindo esse processo até o último elemento da lista, se forma sequencial e sempre respeitando a função que declaramos, somando sempre dois valores.

```
55
```

O resultado da soma de 1+2=3, 3+3=6, 6+4=10, 10+5=15, 15+6=21, 21+7=28, 28+8=36, 36+9=45, 45+10=55, retorna o valor 55.



COMBINANDO MAP, FILTER E REDUCE

Como você bem sabe, Python é uma linguagem de programação dinâmica, onde a mesma ação pode ser codificada de diferentes formas obtendo o mesmo resultado, ficando a critério do desenvolvedor optar por usar do método ao qual se sente mais confortável e que não afete negativamente a performance do código.

Nessa lógica, uma vez entendido o funcionamento de funções como map, filter e reduce, podemos realizar a combinação dos mesmos, o que é uma prática comum em aplicações reais.

Apenas uma ressalva aqui, lembre-se que nem sempre o método mais avançado/complexo é o mais eficiente, dentre os fatores que o desenvolvedor deve levar em consideração para criar seu programa, os mais importantes deles são, sem dúvida, a eficiência tanto em performance quanto em legibilidade do código.

Como visto anteriormente, embora parecidas, as funções em map, filter e reduce possuem algumas particularidades e aplicações específicas.

Sendo assim, temos de analisar muito bem o contexto onde as mesmas devem ser aplicadas, assim como as possíveis combinações entre as mesmas de modo a otimizar o código.

Vamos procurar entender essas diferenças de uma vez por todas com base no exemplo a seguir.

```
1  estoque = [
2      {'Item01': 'Camisa Nike', 'Preco':39.90},
3      {'Item02': 'Camisa Adidas', 'Preco':37.90},
4      {'Item03': 'Moletom 00', 'Preco':79.90},
5      {'Item04': 'Calca Jeans', 'Preco': 69.90},
6      {'Item05': 'Tenis AllStar', 'Preco': 59.90}
7  ]
8
```

Inicialmente criamos uma variável de nome estoque, que recebe atribuída para si uma lista, dentro da lista estão cadastrados alguns itens em formato de dicionário, a partir desse momento podemos começar as devidas iterações.

```
 9  # Método convencional para extrair dados
10  precos01 = []
11  for preco in estoque:
12      precos01.append(preco['Preco'])
13  print(f'Preços de estoque (normal) {precos01}')
14
```

Se você está familiarizado a trabalhar com listas deve se lembrar que um dos métodos mais básicos de extrair algum dado/valor de algum elemento da lista é criando um laço for que percorre cada elemento da lista retornando seu respectivo dado/valor.

Aqui criamos uma variável de nome precos01, que inicialmente recebe como atributo uma lista vazia, em seguida criamos um laço for que percorrerá cada elemento de estoque, depois ele pega especificamente os valores das chaves ‘Preco’, retornando os mesmos para preco01.

Por fim via função print( ) é criada uma mensagem que faz referência aos valores encontrados a precos01.

Extraindo dados via Map + Função Lambda

A partir deste momento teríamos tais dados separados prontos para qualquer uso, porém nosso foco nesse momento será ver outras formas de realizar essa mesma ação.

```
16  # Extraindo dados via Map + Função Lambda
17  precos02 = list(map(lambda p: p['Preco'], estoque))
18  for preco in precos02:
19      print(preco)
20  print(f'Preços de estoque (Map + Lambda) {precos02}')
21
```

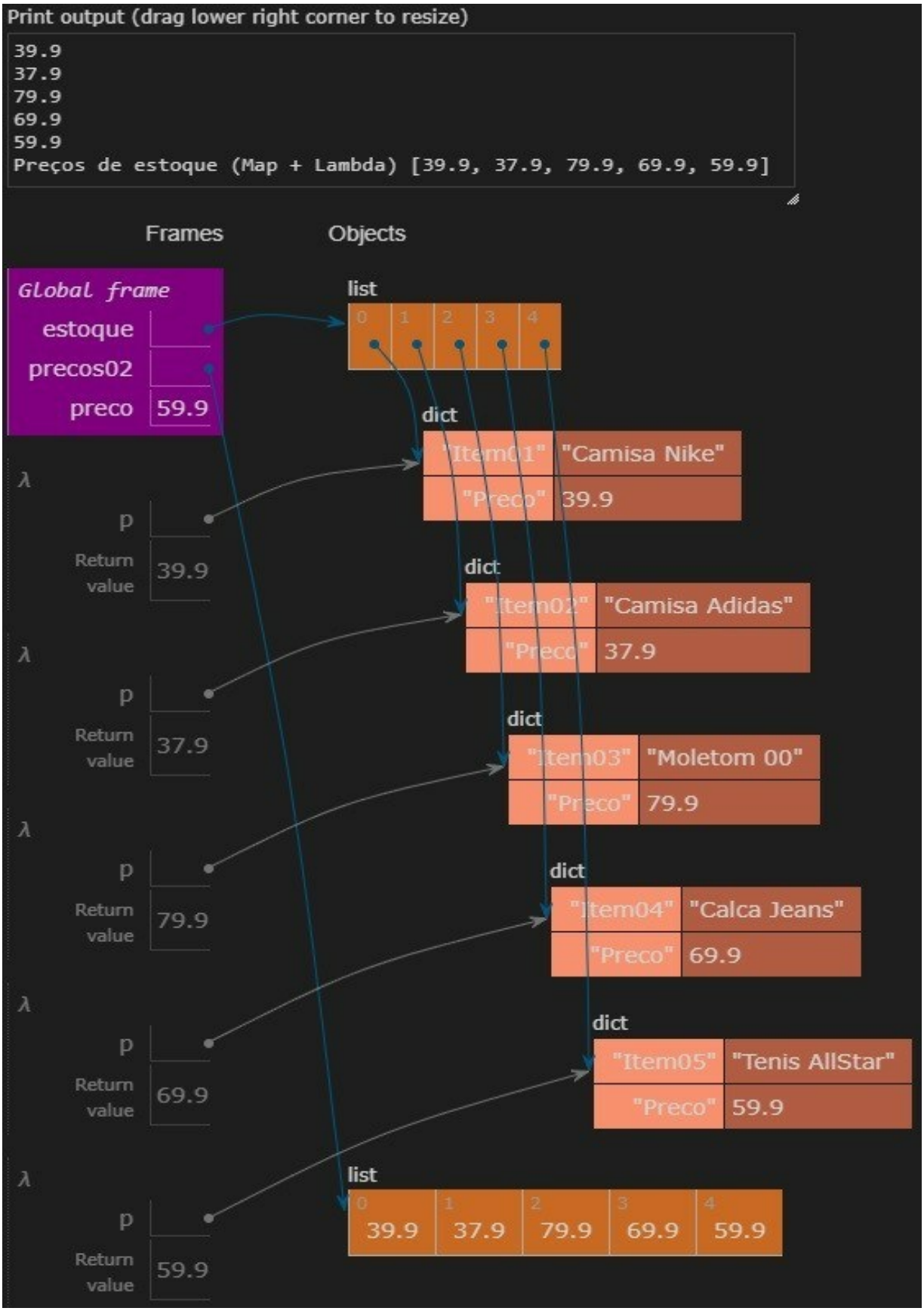
Usando de outra sistemática, um pouco mais complexa, podemos via Map realizar o mesmo procedimento do exemplo anterior.

Para esse exemplo criamos uma variável de nome precos02 que recebe atribuído para si uma lista que faz o uso de map para, com o auxílio de uma expressão lambda, extrai os valores de ‘Preco’ de nossa base de dados inicial estoque.

De forma acessória, aqui também é usado um laço for para exibir apenas os dados que queremos, neste caso, apenas os preços.

Outro ponto a se salientar aqui é que, em função das particularidades de leitura da função map, aquela mensagem que colocamos em nossa função print( ) irá retornar apenas o primeiro valor, e a cada execução, o respectivo valor seguinte, o que não é o que buscamos para esse tipo de exemplo.

Isso justifica o uso de laço for nesse exemplo para que possamos ter um dado com todos valores que precisamos.


Print output (drag lower right corner to resize)
39.9
37.9
79.9
69.9
59.9
Preços de estoque (Map + Lambda) [39.9, 37.9, 79.9, 69.9, 59.9]
Frames
Objects
Global frame
estoque
precos02
preco
59.9
list
0
1
2
3
4
dict
"Item01"
"Camisa Nike"
"Preco"
39.9
λ
p
Return value
39.9
dict
"Item02"
"Camisa Adidas"
"Preco"
37.9
λ
p
Return value
37.9
dict
"Item03"
"Moletom 00"
"Preco"
79.9
λ
p
Return value
79.9
dict
"Item04"
"Calca Jeans"
"Preco"
69.9
λ
p
Return value
69.9
dict
"Item05"
"Tenis AllStar"
"Preco"
59.9
λ
p
Return value
59.9
list
0
39.9
1
37.9
2
79.9
3
69.9
4
59.9

Extraindo dados via List Comprehension

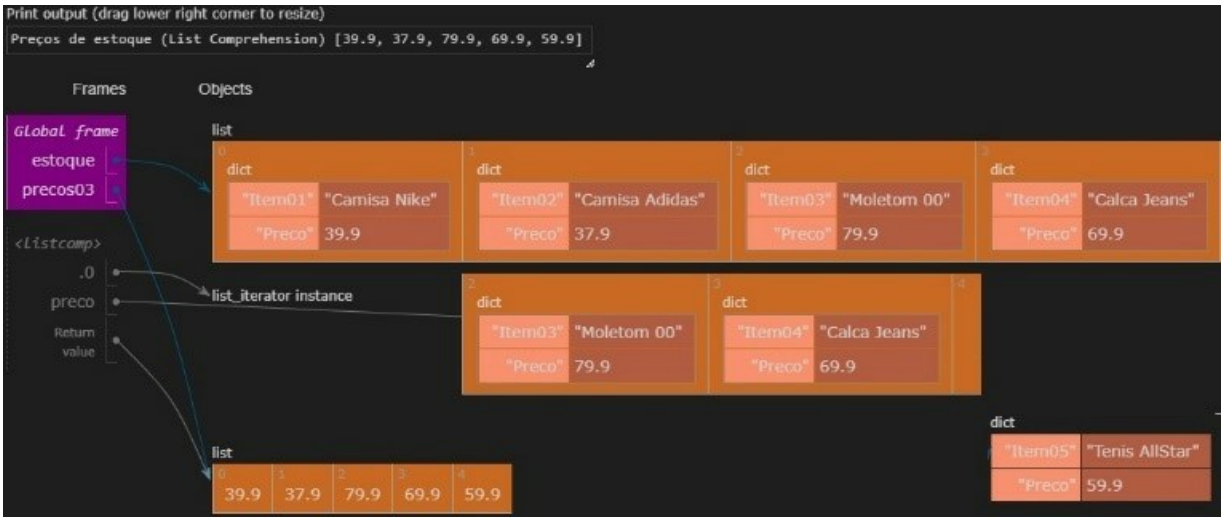
```
23  # Extraindo dados via List Comprehension
24  precos03 = [preco['Preco'] for preco in estoque]
25  print(f'Preços de estoque (List Comprehension) {precos03}')
26
```

Outra forma de realizar a mesma extração, é por meio de List Comprehension, o que aumenta um pouco a complexidade caso você não domine a lógica deste tipo de tratamento de dados.

Para esse exemplo criamos uma variável de nome precos03 que por sua vez recebe uma lista, dentro da lista temos toda a estrutura que irá extrair os dados de estoque.

Note que aqui criado nos moldes de list comprehension simplesmente temos um laço que percorre estoque, retornando apenas os dados/valores encontrados em ‘Preco’.

Como nos exemplos anteriores, também criamos uma mensagem que exibirá o devido resultado.



Extraindo dados via Filter

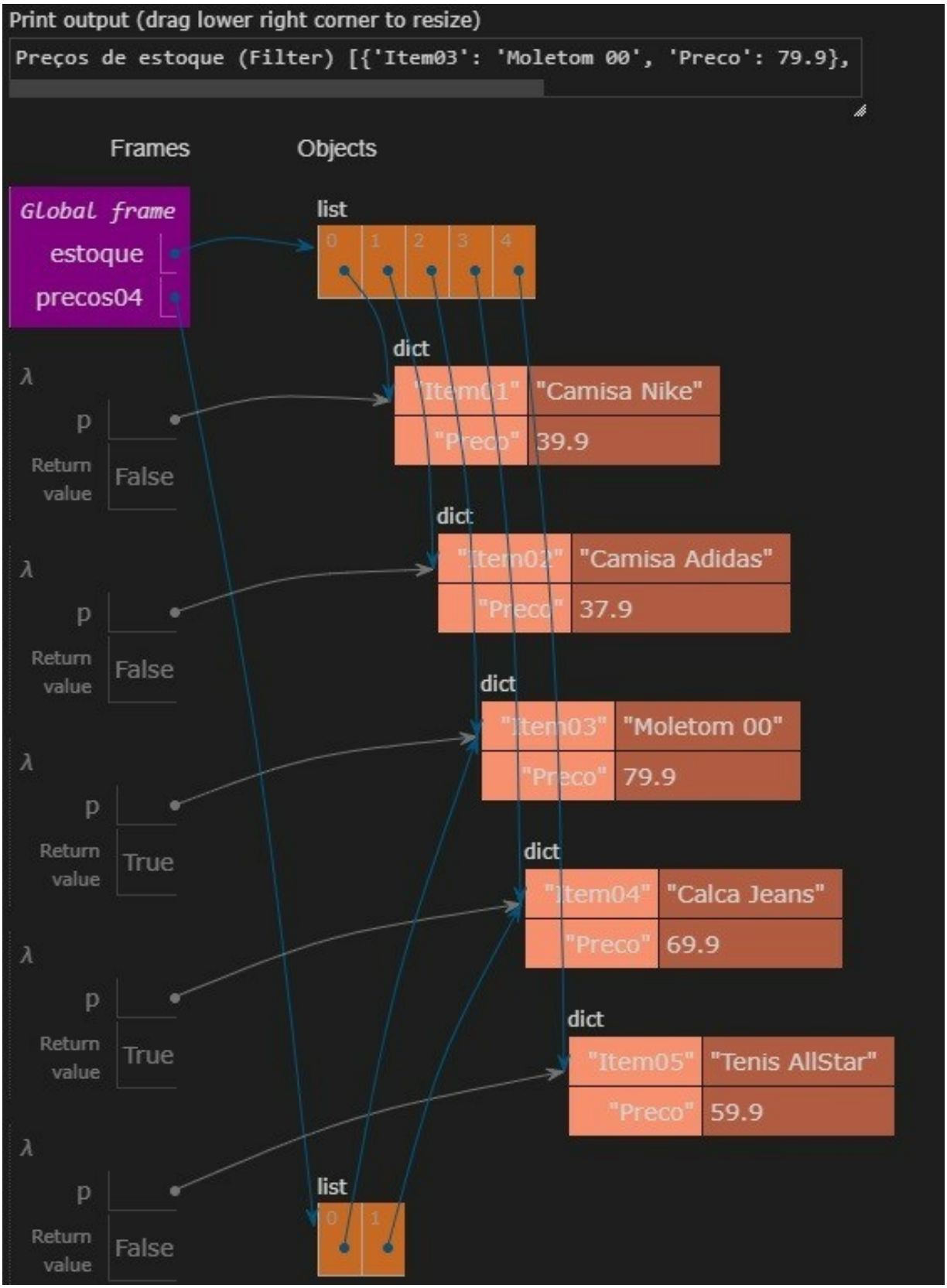
```
28  # Extraindo dados via Filter
29  precos04 = list(filter(lambda p: p['Preco'] >= 60, estoque))
30  print(f'Preços de estoque (Filter) {precos04}')
31
```

Mais uma forma que temos de extrair dados de nossa base estoque é por meio de Filter. Lembrando que filter por sua vez possui uma estrutura otimizada para que literalmente se aplique um filtro sobre algum tipo de dado retornando um dado/valor.

Em situações onde é necessário retornar múltiplos dados/valores via Filter a mesma se mostra menos eficiente que o convencional.

Na mesma linha de raciocínio, criamos uma variável precos04 que recebe como atributo uma lista, que por meio da função filter( ) extrai de ‘Preco’ os valores iguais ou maiores que 60, neste caso apenas como exemplo.

Por fim também é criada via print( ) uma mensagem para exibir tais dados.


Print output (drag lower right corner to resize)
Preços de estoque (Filter) [{'Item03': 'Moletom 00', 'Preco': 79.9},
Frames
Objects
Global frame
estoque
precos04
list
0
1
2
3
4
dict
"Item01"
"Camisa Nike"
"Preco"
39.9
λ
p
Return value
False
dict
"Item02"
"Camisa Adidas"
"Preco"
37.9
λ
p
Return value
False
dict
"Item03"
"Moletom 00"
"Preco"
79.9
λ
p
Return value
True
dict
"Item04"
"Calca Jeans"
"Preco"
69.9
λ
p
Return value
True
dict
"Item05"
"Tenis AllStar"
"Preco"
59.9
λ
p
Return value
False
list
0
1

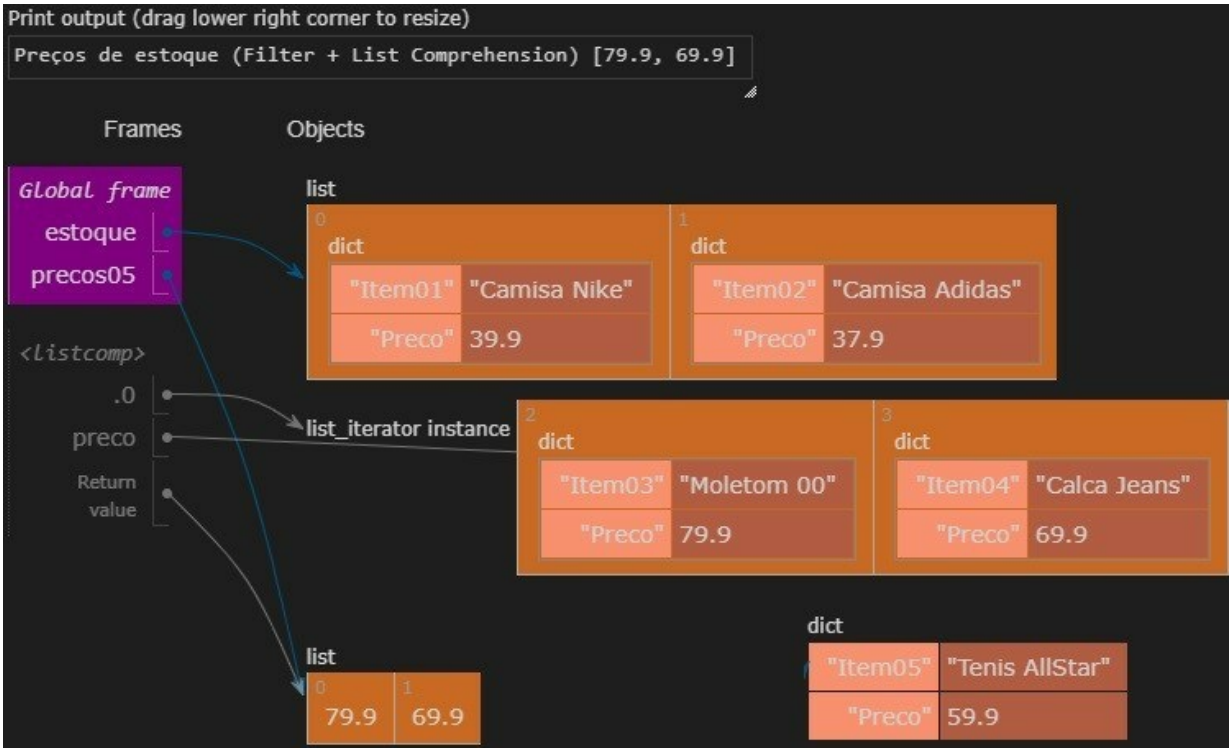
Combinando Filter e List Comprehension

```
33  # Combinando Filter e List Comprehension
34  precos05 = [preco['Preco'] for preco in estoque if preco['Preco'] > 60]
35  print(f'Preços de estoque (Filter + List Comprehension) {precos05}')
36
```

Apenas para continuar na mesma sequência lógica, trabalhando com Filter também é perfeitamente possível fazer o uso de list comprehension.

Repare que para esse exemplo criamos a variável precos05, que por sua vez recebe uma lista onde internamente está sendo criado um laço for que percorre os dados de estoque, mais especificamente ‘Preco’, na mesma expressão é colocada uma estrutura condicional para retornar de ‘Preco’ apenas os valores maiores que 60, em função do propósito original de Filter.

Também é criada via print( ) uma mensagem exibindo tais dados.



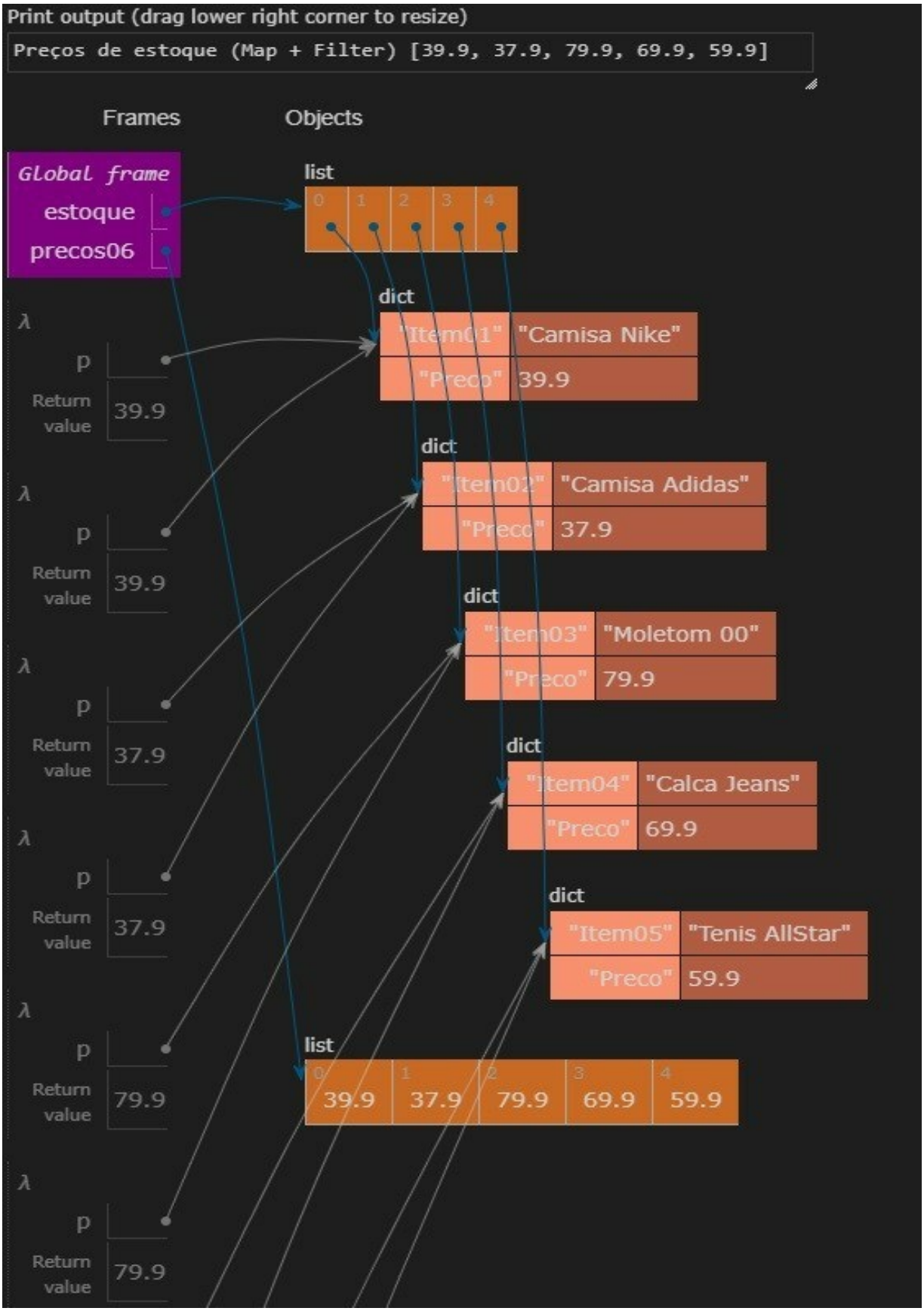
Combinando Map e Filter

```
38  # Combinando Map e Filter para extrair um dado
39  precos06 = list(map(lambda p: p['Preco'],
40                      filter(lambda p: p['Preco'], estoque)))
41  print(f'Preços de estoque (Map + Filter) {precos06}')
42
```

Outra possibilidade que temos é combinar Map e Filter dentro de uma mesma expressão. Embora possa parecer confuso, é uma prática um tanto quanto avançada, mas bastante usual.

Note que para esse exemplo criamos a variável precos06 que recebe uma lista que faz o uso de map( ), onde o primeiro elemento de mapeamento é uma expressão lambda que percorre e retorna os valores de ‘Preco’, como segundo elemento de mapeamento temos filter( ) que realiza o mesmo procedimento extraindo dados de estoque.

Por fim, como de costume, criamos uma mensagem que exibe tais dados.


Print output (drag lower right corner to resize)
Preços de estoque (Map + Filter) [39.9, 37.9, 79.9, 69.9, 59.9]
Frames
Objects
Global frame
estoque
precos06
list
0 1 2 3 4
dict
"Item01" "Camisa Nike"
"Preco" 39.9
dict
"Item02" "Camisa Adidas"
"Preco" 37.9
dict
"Item03" "Moletom 00"
"Preco" 79.9
dict
"Item04" "Calca Jeans"
"Preco" 69.9
dict
"Item05" "Tenis AllStar"
"Preco" 59.9
λ
p
Return value
39.9
λ
p
Return value
39.9
λ
p
Return value
37.9
λ
p
Return value
37.9
λ
p
Return value
79.9
λ
p
Return value
79.9
list
0 1 2 3 4
39.9 37.9 79.9 69.9 59.9

Extraindo dados via Reduce

```
44  # Extraindo dados via Reduce
45  from functools import reduce
46  def func_reduce(soma, valores):
47      return soma + valores['Preco']
48  precos_total = reduce(func_reduce, estoque, 0)
49  print(f'Soma dos Preços (Reduce) {precos_total}')
50
```

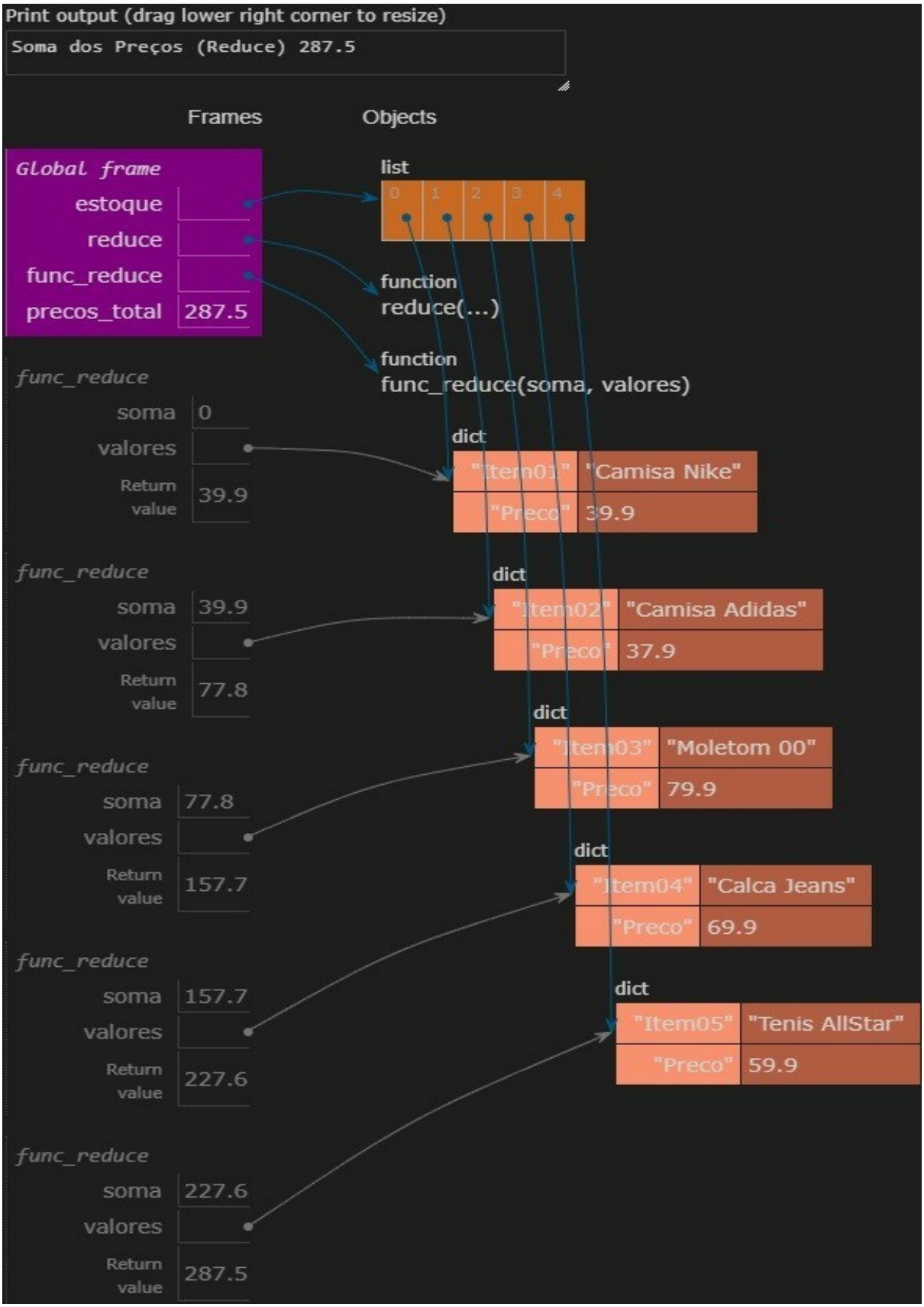
Outra maneira, um pouco mais avançada se comparada com as demais, é realizar a mesma extração de dados via Reduce.

Para isso inicialmente precisamos importar Reduce uma vez que reduce por si só é um módulo da biblioteca functools, que por padrão não é pré-carregada junto das outras bibliotecas nativas.

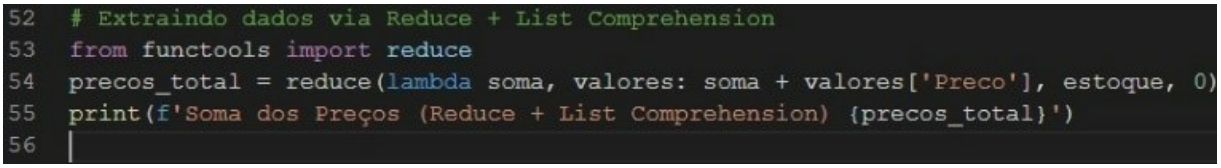
Em seguida é criada a função func_reduce( ) que recebe um objeto soma e outro valores, internamente ela realiza a simples soma dos dados que forem atribuídos ao objeto soma e de ‘Preco’.

Na sequência é criada uma variável preços_total que chama a função reduce( ) parametrizando a mesma com a função func_reduce( ), os dados de estoque e aquele terceiro elemento 0 nada mais é do que um parâmetro para que a soma desses valores comece em zero.

Mais uma vez, criamos uma mensagem para exibir tais dados via função print( ).


Print output (drag lower right corner to resize)
Soma dos Preços (Reduce) 287.5
Frames
Objects
Global frame
estoque
reduce
func_reduce
precos_total
287.5
list
0
1
2
3
4
function
reduce(...)
function
func_reduce(soma, valores)
func_reduce
soma
0
valores
Return value
39.9
dict
"Item01"
"Camisa Nike"
"Preco"
39.9
func_reduce
soma
39.9
valores
Return value
77.8
dict
"Item02"
"Camisa Adidas"
"Preco"
37.9
func_reduce
soma
77.8
valores
Return value
157.7
dict
"Item03"
"Moletom 00"
"Preco"
79.9
func_reduce
soma
157.7
valores
Return value
227.6
dict
"Item04"
"Calca Jeans"
"Preco"
69.9
func_reduce
soma
227.6
valores
Return value
287.5
dict
"Item05"
"Tenis AllStar"
"Preco"
59.9

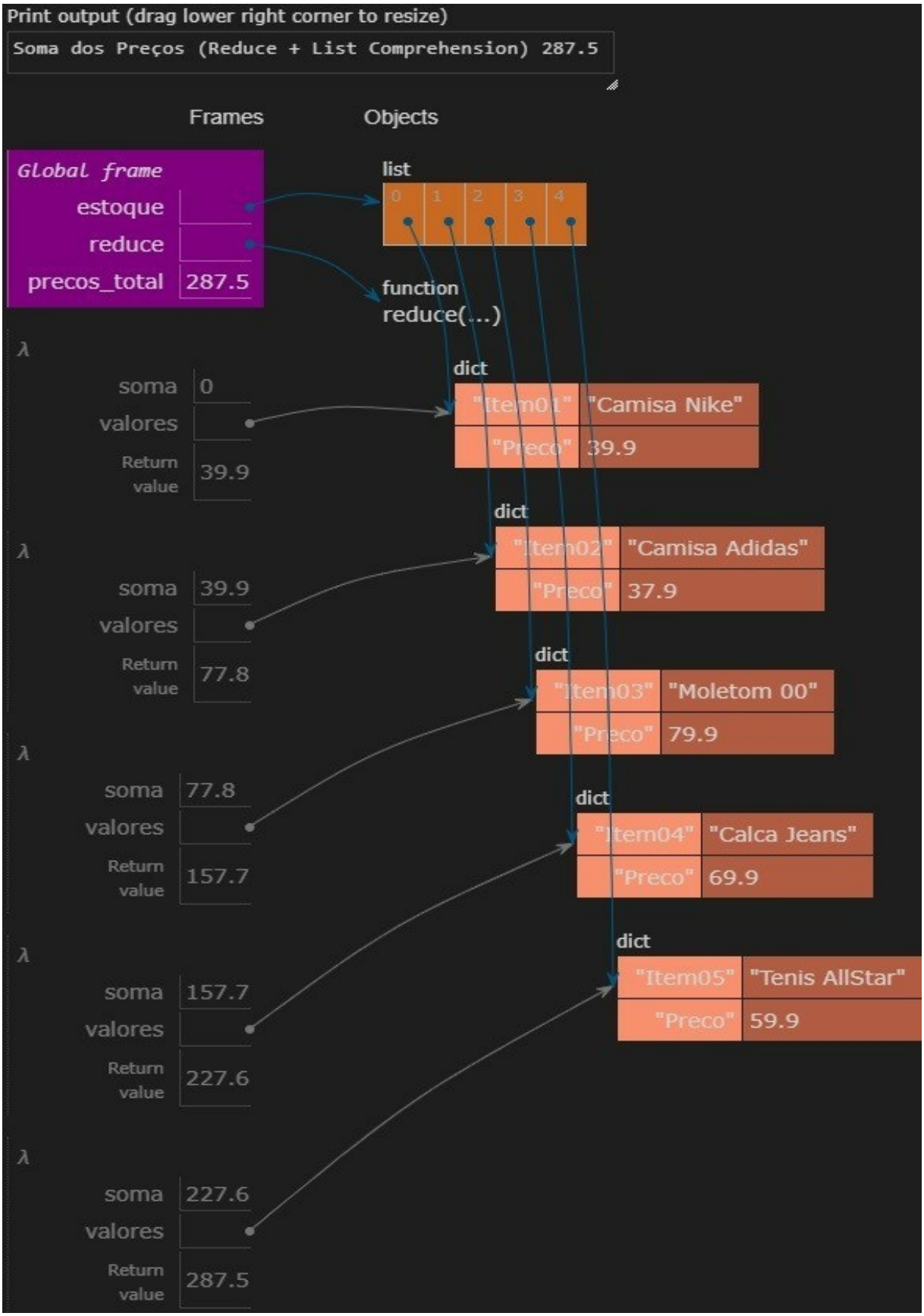
Reduce + List Comprehension



Não muito diferente dos exemplos anteriores, trabalhando com Reduce podemos perfeitamente usar de list comprehension dentro de nossas expressões.

Inicialmente é criada uma variável preços_total que recebe reduce( ) parametrizando a mesma com uma expressão lambda que recebe soma, valores e realiza a soma dos mesmos, buscando dados de estoque e iniciando a soma em 0.

Sim, é exatamente a mesma função criada no exemplo anterior, agora em forma de list comprehension.


Print output (drag lower right corner to resize)
Soma dos Preços (Reduce + List Comprehension) 287.5
Frames
Objects
Global frame
estoque
reduce
precos_total
287.5
list
0
1
2
3
4
function
reduce(...)
λ
soma
0
valores
Return value
39.9
dict
"Item01"
"Camisa Nike"
"Preco"
39.9
λ
soma
39.9
valores
Return value
77.8
dict
"Item02"
"Camisa Adidas"
"Preco"
37.9
λ
soma
77.8
valores
Return value
157.7
dict
"Item03"
"Moletom 00"
"Preco"
79.9
λ
soma
157.7
valores
Return value
227.6
dict
"Item04"
"Calca Jeans"
"Preco"
69.9
λ
soma
227.6
valores
Return value
287.5
dict
"Item05"
"Tenis AllStar"
"Preco"
59.9

Reduce (Método Otimizado)

```
58  # Extraindo dados via Reduce (Método Otimizado)
59  from functools import reduce
60  precos_total = sum([preco['Preco'] for preco in estoque])
61  print(f'Soma dos Preços (Reduce Método Otimizado) {precos_total}')
62
```

Finalizando nossa abordagem de diferentes maneiras de extrair os mesmos dados, podemos ainda realizar uma última otimização do exemplo anterior, usando dessa vez da função sum( ) nativa do Python.

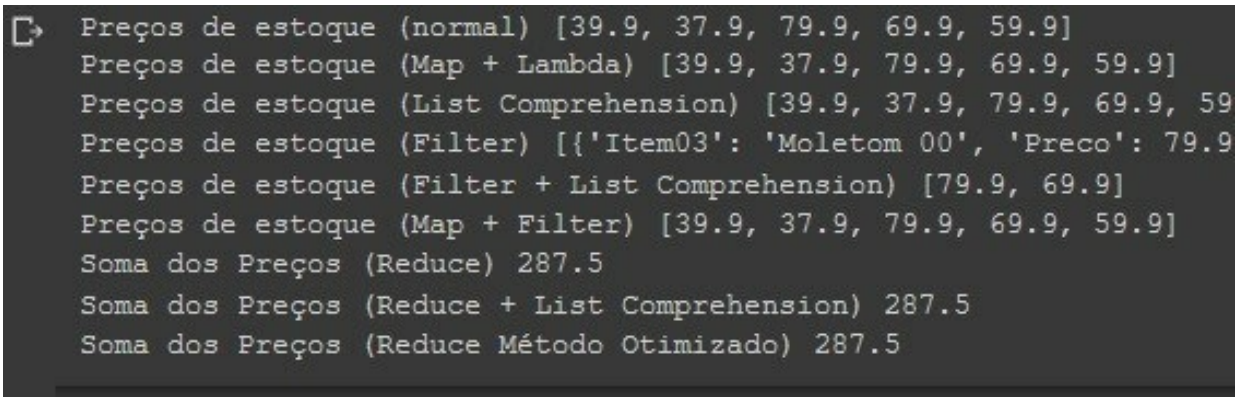
Repare que criamos a mesma variável preços_total, que agora recebe como parâmetro sum( ) parametrizado com os dados de ‘Preco’ encontrados via laço for em nosso objeto estoque.

Por fim é criada uma mensagem de retorno via função print( ).



Encerrando nosso raciocínio sobre este capítulo, em Python sempre haverão várias formas de se realizar a mesma ação, cabendo ao usuário

escolher a que julgar mais eficiente.



Eis que temos os retornos das funções print( ) criadas em cada um dos exemplos anteriores.

Essas foram apenas algumas das funções mais utilizadas nos chamados Built-ins do Python, diversas outras com suas respectivas funcionalidades podem ser encontradas na própria documentação do Python.

Python Built-in Functions

- Python abs() function
- Python bin() function
- Python id() function
- Python map() function
- Python zip() function
- Python filter() function
- Python reduce() function
- Python sorted() function
- Python enumerate() function
- Python reversed() function
- Python range() function
- Python sum() function
- Python max() function
- Python min() function
- Python eval() function
- Python len() function
- Python ord() function
- Python chr() function
- Python any() function
- Python all() function
- Python globals() function
- Python locals() function

TRY, EXCEPT

Uma das boas práticas de programação é, no desenvolvimento de uma determinada aplicação, tentar na medida do possível prever todas as situações onde o usuário pode querer fazer o uso inadequado de uma funcionalidade, gerando um erro.

Você certamente já utilizou de softwares que ao tentar realizar uma determinada ação simplesmente deixavam de funcionar, e isso ocorre porque aquela situação não foi prevista pelo desenvolvedor.

Retomando nossa linha de raciocínio, independente da complexidade de nosso programa, devemos nos colocar no lugar do usuário e tenra por a prova todas as possíveis situações adversas que podem gerar erros de execução de nosso programa.

Internamente, no âmbito de nosso código, as chamadas exceções devem ser tratadas de modo que de nenhuma forma seja interrompida a execução de um código, e isto é feito via Try e Except.

Certamente se você chegou a este ponto, deve dominar ao menos a lógica de estruturas condicionais. Lembrando que por meio de estruturas de código como if, elifs e elses, criamos condições que somente ao serem alcançadas um determinado bloco de código era executado.

Não muito diferente disso teremos as estruturas de Try e Except, pois aqui criaremos estruturas de validação onde caso uma exceção ocorra, haverá um meio alternativo de tentar executar a respectiva funcionalidade sem que pare a execução de nosso programa.

```
1  try:
2    print(a)
3  except:
4    print('Não foi possível exibir o conteúdo de "a"')
5
```

Começando do início, a sintaxe básica deste tipo de estrutura de dados validadora se dá como apresentado acima e pode ser incorporado em

qualquer outra estrutura de código.

A estrutura mais básica deste tipo de validador será sempre algo que o interpretador tentará executar indentado a try, onde caso o mesmo não consiga executar tal bloco de código, deverá executar o que está indentado para except.

Como dito anteriormente, esta estrutura se assemelha muito com as já conhecidas estruturas condicionais, inclusive é perfeitamente possível criar estruturas de validação por meio de estruturas condicionais.

A grande diferença se dá que aqui nesse contexto, uma exceção será gerada pela tentativa não bem sucedida de executar um determinado bloco de código, o que faria nosso programa parar de funcionar.

Enquanto nas estruturas condicionais simplesmente se tal condição não fosse alcançada seria ignorada pelo interpretador.

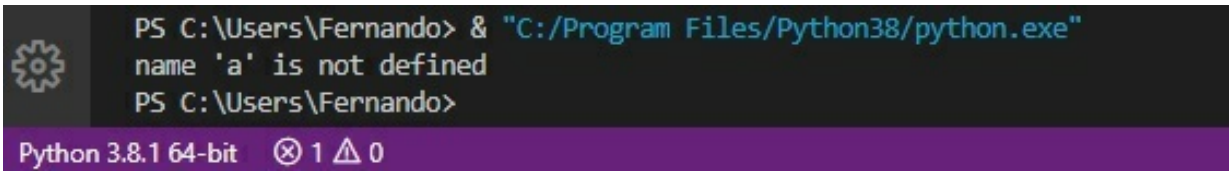
Voltando ao exemplo, repare que em try estamos tentando exibir o conteúdo de uma variável de nome a, essa variável por si só não existe aqui nesse exemplo, logo, se essa linha de código estivesse no corpo de nosso programa, ao tentar executar o mesmo teríamos um erro.

Aqui, nesse contexto, por não ser possível executar o comando indentado para try, o interpretador simplesmente fará a execução do que estiver indentado em except, nesse caso, uma mensagem de erro já definida pelo desenvolvedor.

```
1  except Exception as erro:
2      print('Ocorreu um erro inesperado')
3
```

Uma vez que contornamos a questão de nosso programa parar de funcionar em decorrência de um erro, algo bastante comum é definirmos uma espécie de erro genérico, onde para o usuário será exibida uma mensagem de erro, e para o desenvolvedor será exibido via console qual foi o erro gerado para que possa ser trabalhado para correção do mesmo.

Por meio da palavra reservada ao sistema Exception, podemos realizar tal tipo de inspeção em nosso erro.



Enquanto o usuário recebe a mensagem “Ocorreu um erro inesperado”, o desenvolvedor em seu terminal tem uma resposta mais objetiva. Nesse caso, o log nos mostra que, como esperado, o erro se dá em função de que a variável a não existe, não existe o nome “a” atribuído a nada no código.

```
except NameError as erro:
    print('Ocorreu um erro:',erro)

```

O mesmo pode ser feito com outros tipos de identificadores de erro, na imagem acima, por meio de NameError teríamos, independente do tipo de erro, o nome do erro em si.

Apenas concluindo essa linha de raciocínio, você pode via Exception gerar um erro genérico, sem problema, assim como você pode usar de alguns identificadores para obter um log de erro mais específico.

Por exemplo usando de MemoryError nos é retornado qual objeto alocado em memória está gerando a exceção, podendo assim encontrar o mesmo no corpo do código e tratar o erro.

Outros identificadores como ValueError, KeyboardInterrupt, SystemExit, etc… podem ser encontrados na documentação do Python.

```
def conversor_num(num):
    try:
        num = int(num)
        return num
    except ValueError:
        try:
            num = float(num)
            return num
        except ValueError:
            pass

```

Simulando uma aplicação real, vamos criar uma estrutura de validação

via try e except. Supondo que estamos a criar uma função que simplesmente tenta converter um número a ser repassado pelo usuário.

Sendo assim criamos a função conversor_num( ) que receberá um número como parâmetro, dentro do corpo desse bloco de código criamos a primeira fase de validação onde via try, o interpretador tentará converter o número repassado para int, se isso for possível, retornará o próprio número.

Na sequência, caso esta primeira “condição” não seja alcançada, o interpretador tentará agora converter este número para float.

Caso nenhuma das tentativas seja bem sucedida, o interpretador simplesmente irá ignorar a execução dessa função. Poderíamos aqui, neste último except, perfeitamente criar uma mensagem de erro ao usuário informando o mesmo que não foi possível converter o dado repassado pelo mesmo.

Note também que a estrutura de validação em si está toda construída dentro da função.

```
12  num1 = conversor_num(input('Digite um número: '))
13
14  if num1 is not None:
15      print(num1 + 100)
16  else:
17      print('Operação inválida.')
18
```

Seguindo o código, é criada uma variável de nome num1 que chama a função conversor_num( ) pedindo para que o usuário digite um número.

Em seguida é criada uma estrutura condicional onde se o valor atribuído a num1 não for None, é exibida a soma daquele número já validado pela função por 100. Caso contrário, é exibida a mensagem de erro.

Raciocine que, por exemplo o usuário digitasse qualquer caractere diferente de um número, o validador construído em nossa função conversor_num( ) faria suas tentativas de conversão, não conseguindo, não iria retornar nada. Dessa forma, na estrutura condicional criada fora da função, o dado/valor atribuído para num1 seria None.

Finally

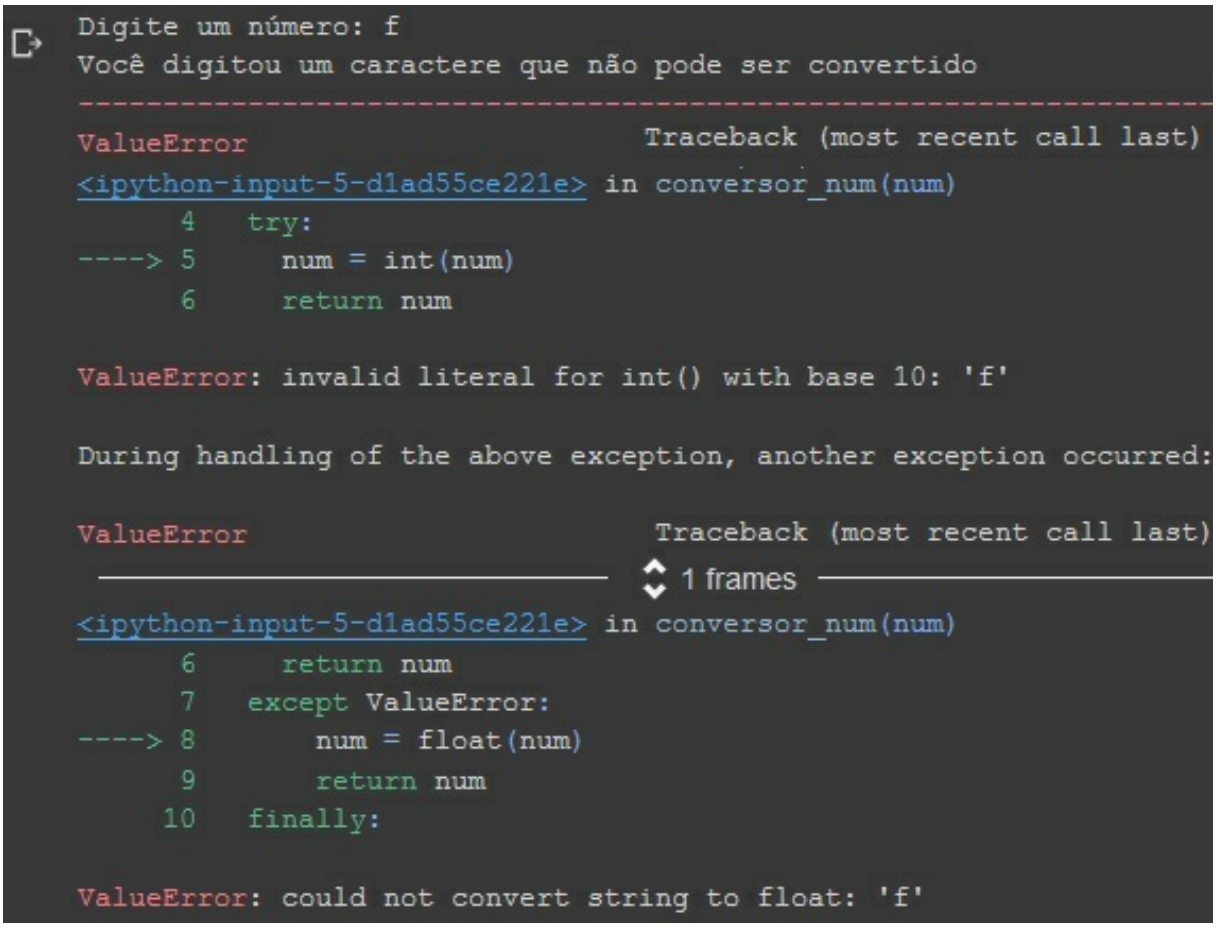
Complementar ao exemplo anterior, porém com usabilidade diferente, temos o operador finally. Como visto no exemplo anterior, criamos um simples sistema de validação onde caso nada das tentativas desse certo, o interpretador simplesmente iria ignorar a execução daquele bloco para que nosso programa não parasse de funcionar.

Diferentemente disso, podemos via finally criar uma expressão onde, caso nenhuma das tentativas de validação dê certo, que o programa seja interrompido de propósito naquele ponto, salientando o erro por parte do usuário.

```
1  def conversor_num(num):
2      try:
3          num = int(num)
4          return num
5      except ValueError:
6              num = float(num)
7              return num
8      finally:
9          print('Você digitou um caractere que não pode ser convertido')
10
11 num1 = conversor_num(input('Digite um número: '))
12
```

Usando de um exemplo parecido com o anterior, note que agora temos indentado na sequência de try e except a palavra reservada do sistema finally.

Agora dispensamos aquela estrutura condicional que estava fora da função e simplesmente realizamos a implementação de uma mensagem a ser exibida ao usuário.



Simulando um erro, ao digitar “f” é exibido ao usuário a mensagem de erro contida em finally, e para o desenvolvedor é apresentado o erro em si, que neste caso, não é nenhum erro por parte do código mas por parte do usuário.

Raise

Uma última possibilidade bastante interessante é a de “levantar” um erro de forma reduzida por meio de raise. Raise por sua vez é uma palavra reservada ao sistema para casos onde queremos especificamente retornar um erro.

```
1  idade = int(input('Digite sua idade: '))
2
3  if idade <= 0:
4    raise Exception('Idade inválida!!!')
5
6  print(f'Você tem {idade} anos!!!')
7
```

Basicamente raise pode ser usado sempre que quisermos levantar um erro ao qual já foi previsto ou que é comumente possível de ocorrer. Nesse contexto, geramos uma mensagem específica para o erro.

```
Digite sua idade: 0
---------------------------------------------------------------------------
Exception                                 Traceback (most recent c
<ipython-input-57-eb524191e42d> in <module>()
      2
      3 if idade <= 0:
----> 4   raise Exception('Idade inválida!!!')
      5
      6 print(f'Você tem {idade} anos!!!')

Exception: Idade inválida!!!
```

Simulando de propósito um erro, digitando 0 como idade quando solicitado, é exibido o traceback onde é possível ver que na última linha a descrição do erro é a mensagem que havíamos declarado anteriormente.

BIBLIOTECAS, MÓDULOS E PACOTES

Ao longo dos últimos capítulos fomos explorando funcionalidades do Python nos mais diversos contextos, onde em alguns casos foi necessário realizar a importação de bibliotecas, módulos e/ou pacotes que por uma questão de performance não eram carregados por padrão.

Nesses moldes, vimos algumas dessas arquiteturas de código independente que de acordo com a aplicação adicionavam as ferramentas necessárias para os mais específicos fins.

Como comentado algumas vezes anteriormente, com o básico de Python é possível fazer muita coisa, e enquanto estudantes de programação desconhecemos toda a gama de possibilidades que a linguagem de programação em si oferece.

Nessa linha de raciocínio, é importante saber onde consultar a documentação da linguagem assim como buscar entender suas ferramentas.

```
1  dir()
2
```

Independente da IDE que você esteja usando, é totalmente possível executar a qualquer momento a função dir( ), esta por sua vez reservada ao sistema, irá oferecer como retorno tudo o que está pré-carregado no Python, assim como pode ser usada para explorar tudo o que há dentro de uma biblioteca, módulo ou pacote.

```
['In',
 'Out',
 '_',
 '__',
 '___',
 '__builtin__',
 '__builtins__',
 '__doc__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 '_dh',
 '_i',
 '_i1',
 '_ih',
 '_ii',
 '_iii',
 '_oh',
 '_sh',
 'exit',
 'get_ipython',
 'quit']
```

Executando a função dir( ) em qualquer parte do corpo do código, sem parâmetros, é possível ver tudo o que está pré-carregado e disponível para uso.

```
1  import builtins
2
3  dir(builtins)
4
```

Todo estudante de Python em algum momento ouviu falar o jargão “Python já vem com pilhas inclusas”, fazendo referência a que justamente a linguagem em seu estado “puro” já possui muitas ferramentas pré-carregadas.

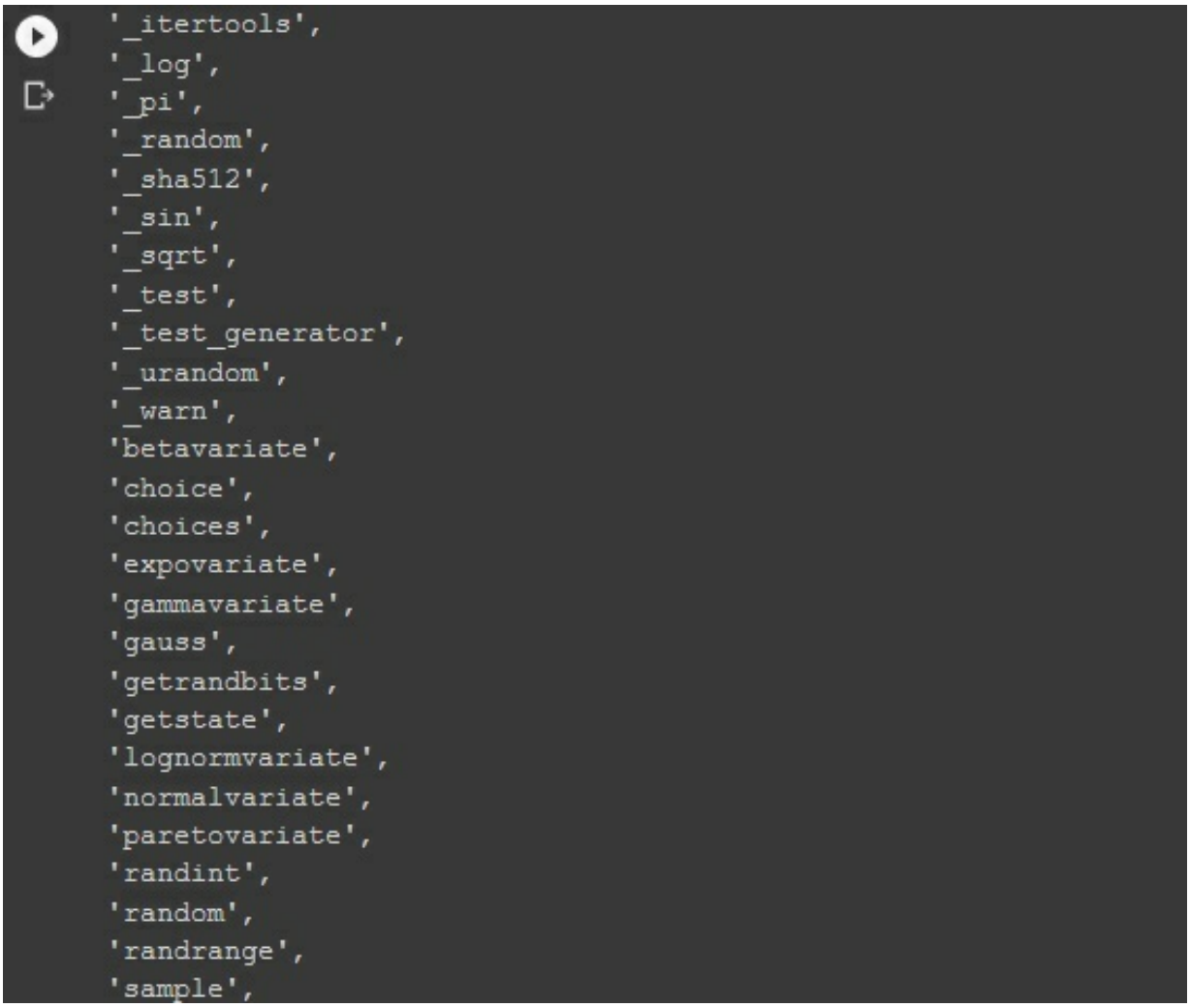
Porém, de acordo com a necessidade é perfeitamente normal fazer a importação de alguma biblioteca externa.

```
['ArithmeticError',        'ImportError',          'StopAsyncIteration',
 'AssertionError',         'ImportWarning',        'StopIteration',
 'AttributeError',         'IndentationError',     'SyntaxError',
 'BaseException',          'IndexError',           'SyntaxWarning',
 'BlockingIOError',        'InterruptedError',     'SystemError',
 'BrokenPipeError',        'IsADirectoryError',    'SystemExit',
 'BufferError',            'KeyError',             'TabError',
 'BytesWarning',           'KeyboardInterrupt',    'TimeoutError',
 'ChildProcessError',      'LookupError',          'True',
 'ConnectionAbortedError' 'MemoryError',          'TypeError',
 'ConnectionError',        'ModuleNotFoundError', 'UnboundLocalError',
 'ConnectionRefusedError' 'NameError',           'UnicodeDecodeError',
 'ConnectionResetError',   'None',                 'UnicodeEncodeError',
 'DeprecationWarning',     'NotADirectoryError',  'UnicodeError',
 'EOFError',               'NotImplemented',       'UnicodeTranslateError',
 'Ellipsis',               'NotImplementedError', 'UnicodeWarning',
 'EnvironmentError',       'OSError',              'UserWarning',
 'Exception',              'OverflowError',        'ValueError',
 'False',                  'PendingDeprecationWar 'Warning',
 'FileExistsError',        'PermissionError',      'ZeroDivisionError',
 'FileNotFoundError',      'ProcessLookupError',  '__IPYTHON__',
 'FloatingPointError',     'RecursionError',       '__build_class__',
 'FutureWarning',          'ReferenceError',       '__debug__',
 'GeneratorExit',          'ResourceWarning',      '__doc__',
 'IOError',                'RuntimeError',         '__import__',
 'ImportError',            'RuntimeWarning',       '__loader__',
```

Apenas como exemplo, importando builtins é carregado em nossa IDE uma grande quantidade de métodos/funções para os mais diversos fins.

```
1  import random
2
3  dir(random)
4
```

Fazendo o mesmo com uma biblioteca externa, nesse caso a biblioteca random, muito utilizada para realizar operações aritméticas em cima de números gerados aleatoriamente.



Como retorno é possível ver tudo o que é carregado da referente biblioteca. Bastando apenas recorrer a sua documentação para entender a fundo as funcionalidades as quais serão implementadas em nosso código.

Referenciando uma biblioteca/módulo

```
1  import random as ra
2
3  for i in range(10):
4      print(ra.randint(1,10))
5
```

Uma questão que devemos dar atenção sempre é quanto a legibilidade de nosso código, seja criando estruturas de dados com nomes que façam sentido a nós mesmos, seja referenciando estruturas de código externas para que facilite nossa vida.

Apenas como exemplo, ao importar qualquer biblioteca ou módulo, podemos dar uma espécie de apelido para a mesma, para que seja mais simples sua implementação.

Em nosso exemplo atual estamos importando a biblioteca random, referenciando a mesma como ra. A partir disto, podemos chamar a biblioteca, seguido de alguma função sua, por meio de ra.nome_da_função( ).

```
1  from random import randint
2
3  for i in range(10):
4      print(randint(0,10))
5
```

Como já explicado em outros momentos, ao importar uma biblioteca estamos importando tudo o que existe compondo a mesma.

A medida que finalizamos nosso programa, realizando o polimento do mesmo, uma boa prática de programação é, quando feito o uso de algum método/função de uma biblioteca, realizar a importação específica daquela método/função.

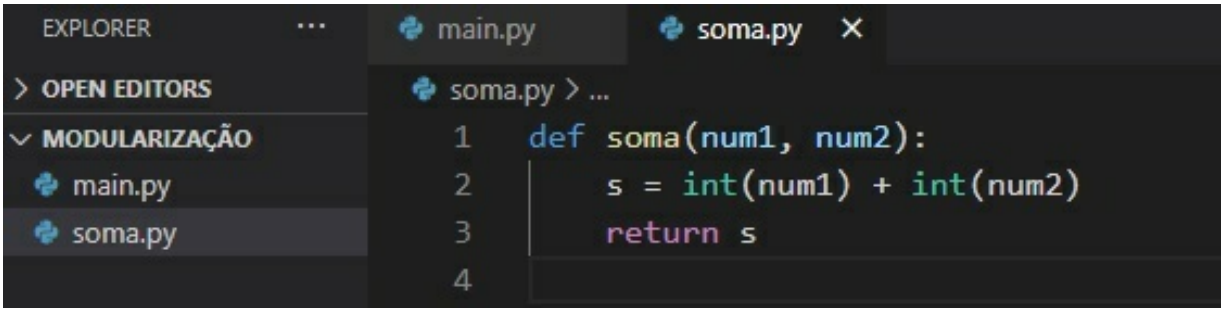
Em outras palavras, se de uma biblioteca como a random necessitamos apenas da função randint para uma aplicação, podemos importar somente esta estrutura de código, mantendo assim um número de estruturas carregadas menor e mais eficiente.

Apenas concluindo essa linha de raciocínio, repare que ao importar especificamente um método/função de um módulo ou de uma biblioteca, a partir da importação não precisamos mais referenciar a biblioteca, mas usar diretamente do método/função como parte do builtin do código.

Criando módulos/pacotes

Se você realizar uma breve pesquisa em repositórios de bibliotecas, módulos e pacotes desenvolvidos para Python, verá que boa parte, senão a maioria delas, é criada pela própria comunidade e distribuída em licença livre e código aberto.

Sendo assim, você já pode presumir que é perfeitamente possível criar suas próprias bibliotecas, módulos e pacotes, e isso é feito de forma muito simples em Python, bastando simplesmente separar as estruturas de código a serem modularizadas em arquivos independentes de extensão .py.

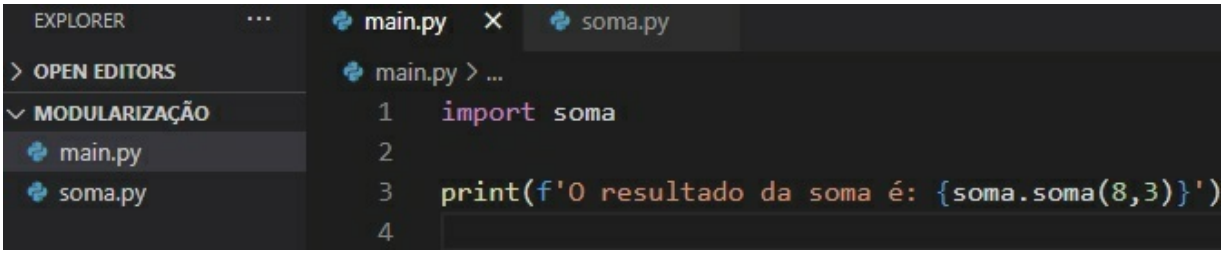


Supondo que estamos a criar um módulo inteiramente único, exclusivo para nosso programa.

Nesse caso, usando de um exemplo bastante básico, supondo que estamos criando um módulo de nome soma, que internamente possui uma simples função que soma dois números e retorna o valor desta operação.

Note que no diretório de nosso projeto essa estrutura de código é salva em um arquivo independente de nome soma.py.

A partir do momento que está criado e pronto este arquivo, esse módulo pode ser importado para qualquer parte de nosso programa.



No mesmo diretório desse nosso projeto temos um arquivo de nome main.py, supondo que este é o arquivo principal de nosso programa, o núcleo do mesmo.

Para usar das funcionalidades de nosso módulo soma, basta realizar o comando import soma, sem a extensão do arquivo. Feito isto, tudo o que estiver dentro de nosso arquivo soma.py será carregado pela IDE e estará pronto para uso.

Nesse caso, apenas como exemplo, repare que criamos uma função print( ) que por meio de f’strings exibe uma mensagem ao usuário e na mesma composição da mensagem, por meio de máscaras de substituição, é chamada a função soma, do módulo soma, parametrizando a mesma com os números 8 e 3.

```
PS C:\Users\Fernando\Desktop\Modularização> & "C:/Program Files/Python38/python.
O resultado da soma é: 11
PS C:\Users\Fernando\Desktop\Modularização>
```

Como esperado, foi de fato realizada a operação e retornado o valor 11.

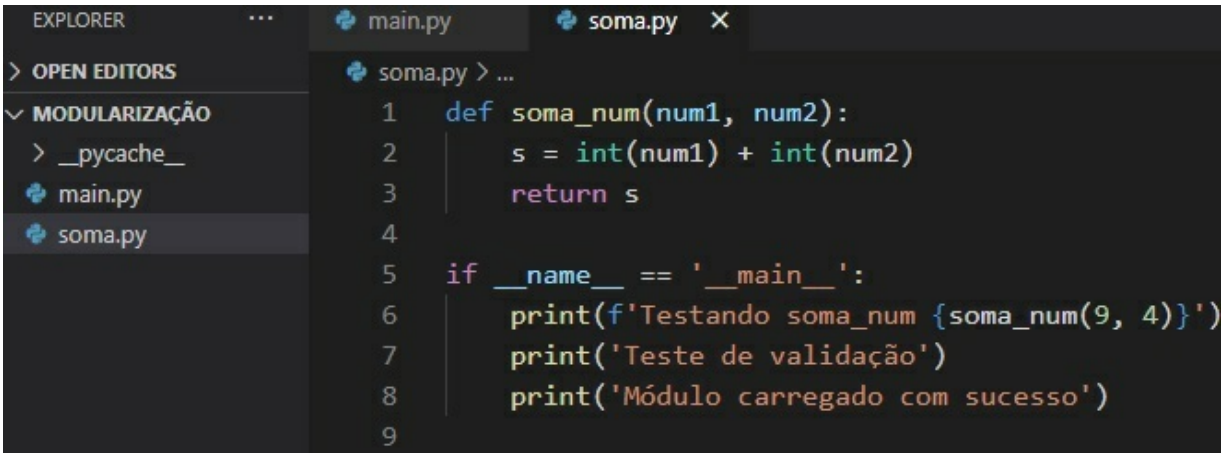
Testando e validando um módulo

Até o momento você viu que em certas situações, fazemos uso de algumas funcionalidades internas do Python, normalmente por meio de estruturas de código associadas a palavras reservadas ao sistema.

Da mesma forma, em outros momentos comentamos que algumas estruturas em Python são implícitas, geradas automaticamente no esqueleto do código cada vez que um novo arquivo é criado.

Por fim, você viu que é possível importar módulos de terceiros assim como criar os seus próprios módulos, bastando realizar a interconexão entre os arquivos e os devidos instanciamentos dentro do código para que os mesmos sejam executados na ordem correta.

Sendo assim, realizando algumas ações diretamente sobre o esqueleto de nosso código podemos realizar algumas validações para ver se de fato nosso código interage com os devidos módulos da maneira correta.



Usando do mesmo exemplo anterior, repare que agora em nosso módulo foi adicionada uma estrutura condicional, porém uma estrutura condicional diferente do que você está habituado a trabalhar.

Analisando esta linha de código, repare que o que está sendo verificado aqui é se __name__ é igual a __main__.

Vamos entender de fato o que é isto: Todo e qualquer arquivo criado em Python, ou seja, com extensão .py, ao inicializado o mesmo não somente é reconhecido pelo interpretador como implicitamente é criado um

esqueleto de código implícito com algumas configurações básicas pré-definidas.

Uma dessas configurações é que ao se criar um novo arquivo em Python se presume que aquele arquivo será o núcleo de nosso programa, o arquivo principal do mesmo, e dessa forma ele possui um escopo implicitamente criado definido como __main__.

Quando estamos usando este arquivo como módulo de um programa, o arquivo principal do programa também tem sua pré-configuração atribuída a __main__, para que não haja conflito entre esses arquivos, e os mesmos tenham seus devidos comportamentos de acordo com sua hierarquia, uma prática comum é realizar uma simples validação, verificando se o módulo em si, quando importado para um programa, não sobrepõe o mesmo.

```
PS C:\Users\Fernando\Desktop\Modularização> & "C:/Program Files/Python38/python.
Testando soma_num 13
Teste de validação
Módulo carregado com sucesso
PS C:\Users\Fernando\Desktop\Modularização>
```

Executando nosso arquivo soma.py, o retorno é a própria mensagem de validação, confirmando que as funcionalidades deste módulo (nesse caso apenas uma soma de dois números) ocorre como esperado.

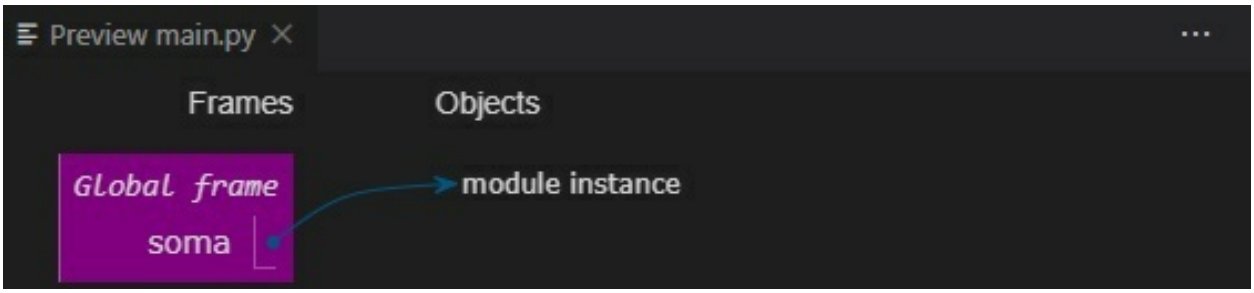
```
Digite um número: 9
Digite outro número: 7
16
__main__
PS C:\Users\Fernando\Desktop\Modularização>
```

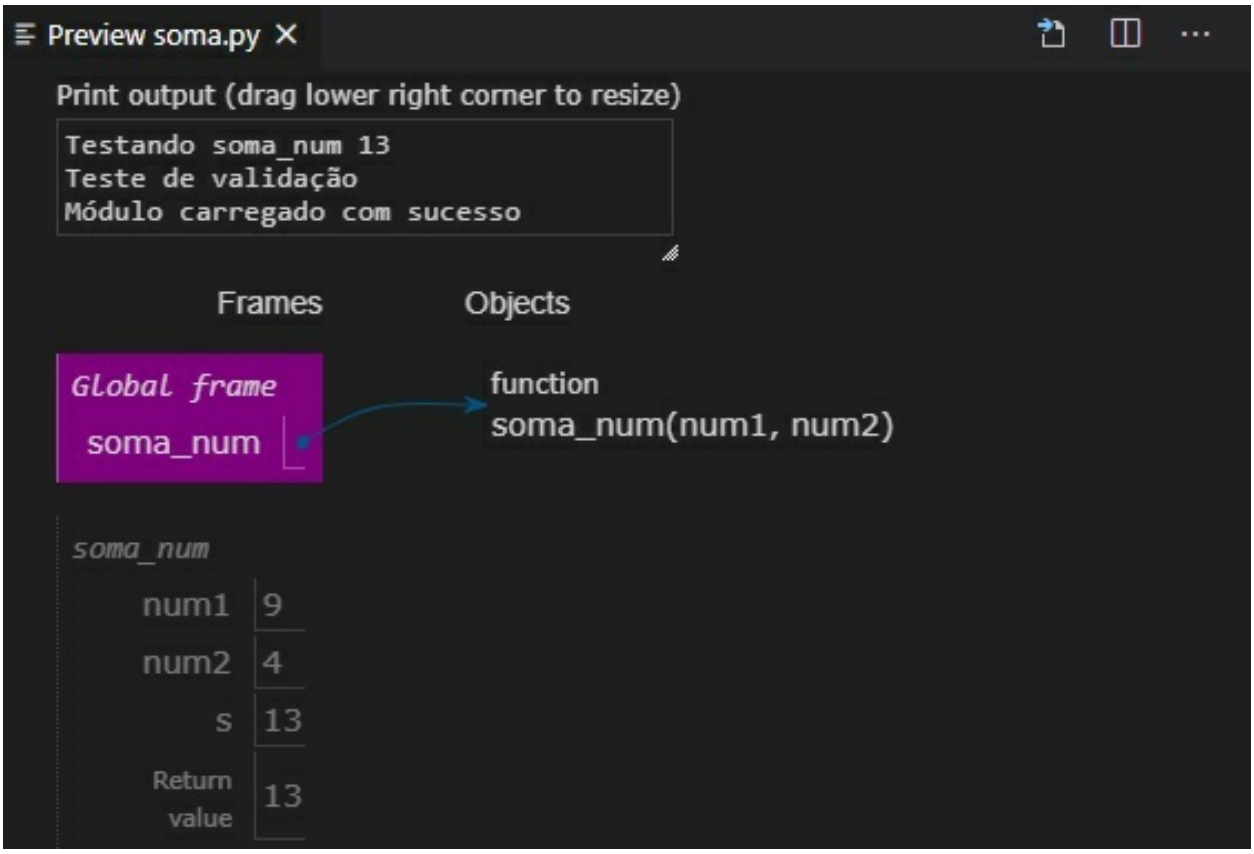
Executando nosso arquivo main.py, como esperado, é chamada a função soma_num( ), pedido que o usuário digite dois números e é retornado a soma dos mesmos. Note que aqui não é executada a estrutura de validação de soma.py.

Em outras palavras, a partir do momento que importamos nosso módulo soma a partir do arquivo soma.py, este módulo, agora subordinado a main.py, não terá mais eu nome igual a “__main__”, mas terá seu nome reconhecido como “soma”.

Dessa forma a hierarquia está correta, sendo as estruturas de código de soma lidas e executadas conforme a demanda do arquivo principal

main.py.

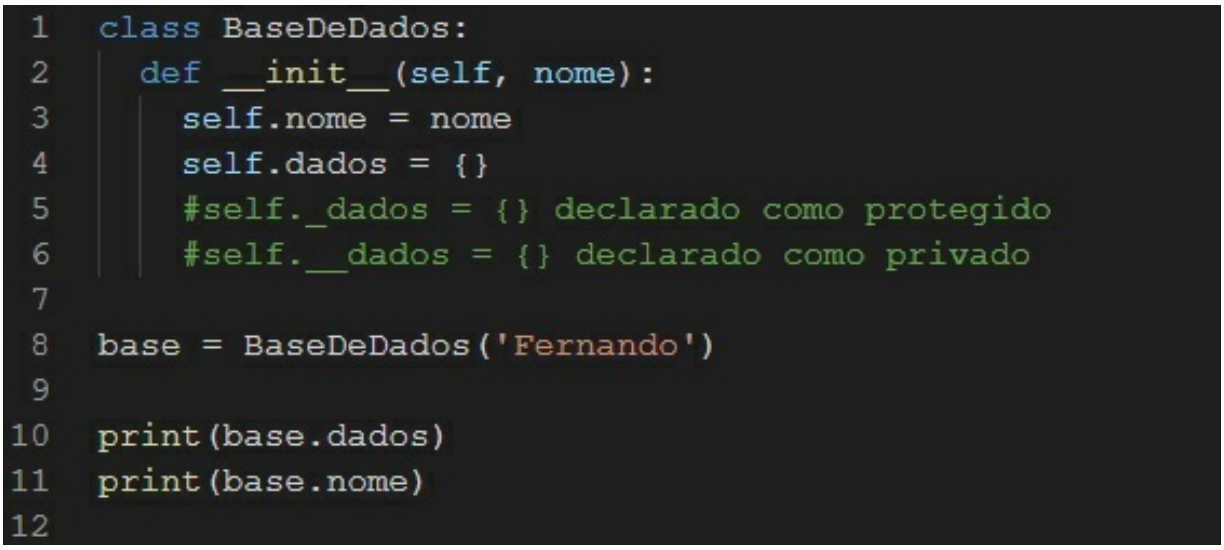

Preview main.py
Frames
Objects
Global frame
soma
module instance


Preview soma.py
Print output (drag lower right corner to resize)
Testando soma_num 13
Teste de validação
Módulo carregado com sucesso
Frames
Objects
Global frame
soma_num
function
soma_num(num1, num2)
soma_num
num1
9
num2
4
s
13
Return value
13

Encapsulamento

Complementando o capítulo anterior, e como já mencionado algumas vezes, em Python temos algumas nomenclaturas que podem parecer confusas porque às vezes nomes diferentes se referem a mesma coisa, apenas tendo nomenclatura diferente quando no âmbito da programação estruturada ou quando se trata de programação orientada a objetos.

O chamado encapsulamento nada mais é do que a forma como iremos declarar a visibilidade de nossos objetos por meio de sua sintaxe.



```
1  class BaseDeDados:
2      def __init__(self, nome):
3          self.nome = nome
4          self.dados = {}
5          #self._dados = {} declarado como protegido
6          #self.__dados = {} declarado como privado
7
8  base = BaseDeDados('Fernando')
9
10 print(base.dados)
11 print(base.nome)
12
```

Usando de um exemplo bastante simples, temos nossa classe BaseDeDados que por sua vez tem seu método construtor que recebe um nome, dentro deste método construtor é criado o objeto para nome e um segundo objeto dados.

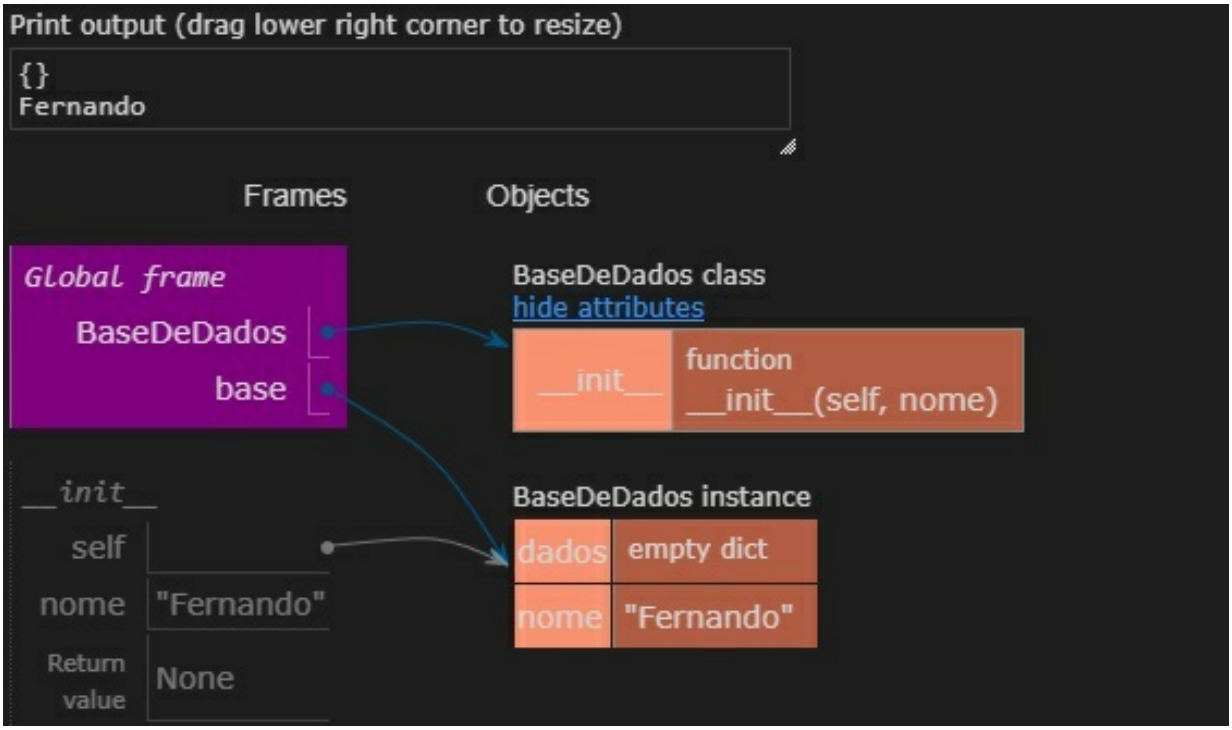
Entendendo de forma prática, simplesmente perceba que existem três formas de declarar um determinado objeto, nesse caso, dados.

A primeira delas “dados” simplesmente faz referência a um objeto do escopo global dessa classe, podendo ser instanciado por qualquer método de classe ou até mesmo por variáveis e funções de fora da classe.

O segundo método, “_dados”, faz com que este objeto seja declarado como protegido, em outras palavras esse objeto está simplesmente sinalizado

como algo importante e que não deve ser alterado por qualquer motivo. Esse objeto por sua vez ainda é acessível dentro e fora da classe.

Por fim o terceiro método, “__dados”, declarado desta maneira significando um objeto privado, inacessível de fora da classe e praticamente imutável mesmo dentro dela (salvo algumas exceções).



Essa sintaxe funciona perfeitamente para qualquer objeto o qual você queira proteger ou privar o acesso.

ITERTOOLS

combinations( )

A seguir veremos alguns possíveis tratamentos de dados em formato de listas. Novamente, a justificativa deste capítulo se dá por trazer aplicações bastante rotineiras de certas funções sobre nossos dados.

Como já se presume pelo título deste tópico, por meio de algumas ferramentas da biblioteca itertools podemos realizar, por exemplo, a combinação de dados a partir de listas.

Esta função em si realiza um tipo de combinação bem específica, onde se faz necessário ler todos elementos de uma lista e ver as possíveis combinações entre tais elementos, sem que hajam combinações repetidas.

```
1  from itertools import combinations
2
3  lista10 = ['Ana', 'Bianca', 'Carla', 'Daniela', 'Franciele', 'Maria']
4
```

Indo para o exemplo, como de costume, todo processo se inicia com as importações das bibliotecas, módulos e pacotes não nativos do Python.

Sendo assim, por meio de import poderíamos importar toda a biblioteca itertools, porém, como já dito em outros capítulos, uma boa prática de programação é realizarmos a importação apenas do que é necessário, por uma questão de performance.

Então, da biblioteca itertools importamos combinations, função dedicada ao nosso propósito atual.

Em seguida é criada uma variável de nome lista10 que atribuído para si tem uma simples lista com 6 elementos. A partir deste momento podemos realizar qualquer tipo de interação com tais dados.

```
5  for combinacoes in combinations(lista10, 2):
6      print(combinacoes)
7
```

Uma das formas mais básicas de realizar a leitura dos elementos de uma lista é por meio de um laço de repetição que percorre elemento por elemento da mesma.

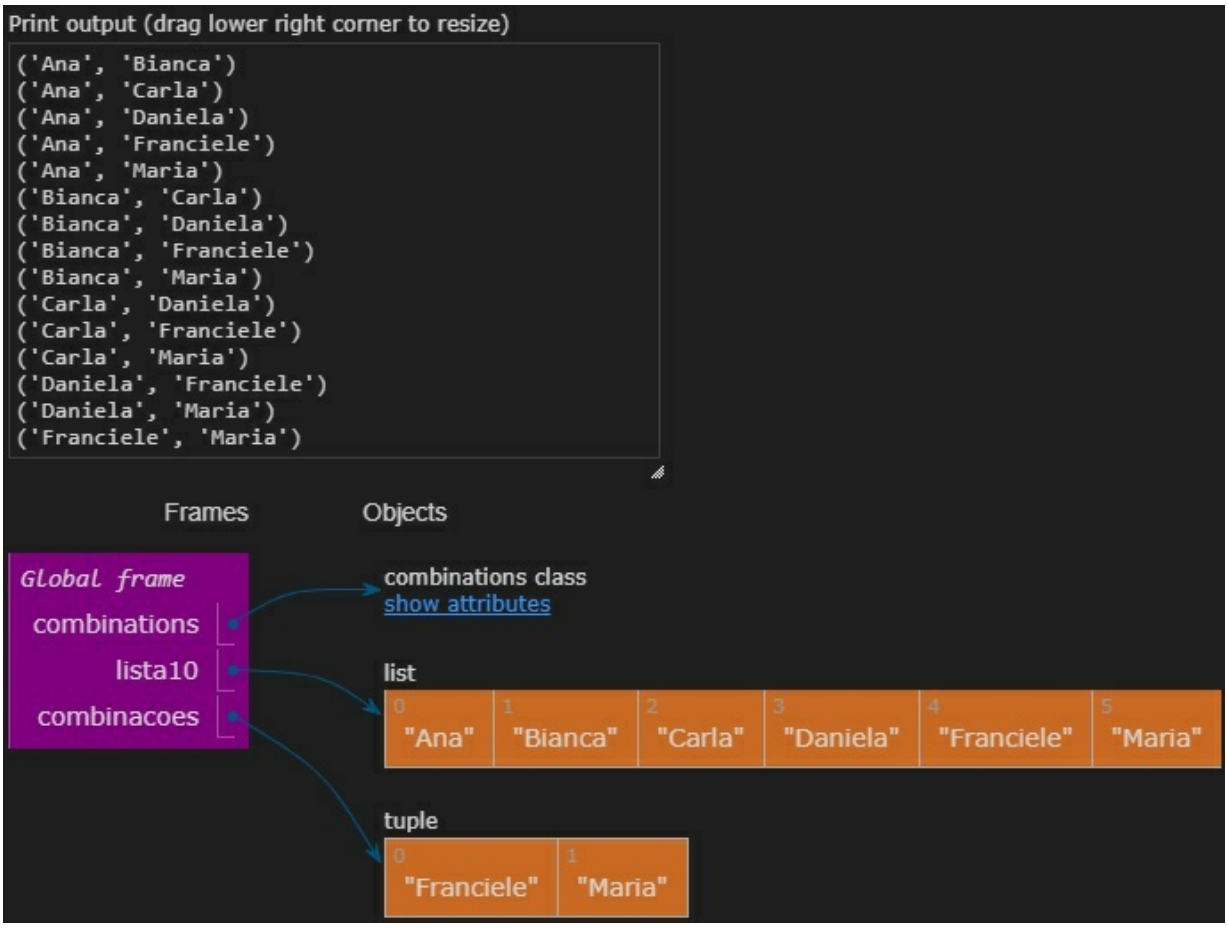
Aqui, incorporamos em nosso laço for a função combinations( ), que por sua vez deve ser parametrizado com o nome da variável de onde serão lidos os dados, seguido do número de possíveis combinações para cada elemento.

```
('Ana', 'Bianca')
('Ana', 'Carla')
('Ana', 'Daniela')
('Ana', 'Franciele')
('Ana', 'Maria')
('Bianca', 'Carla')
('Bianca', 'Daniela')
('Bianca', 'Franciele')
('Bianca', 'Maria')
('Carla', 'Daniela')
('Carla', 'Franciele')
('Carla', 'Maria')
('Daniela', 'Franciele')
('Daniela', 'Maria')
('Franciele', 'Maria')
```

Nesse caso nos é retornado as possíveis combinações dos elementos, mas repare que sem haver repetições, independentemente da ordem dos elementos.

Em outras palavras, para o primeiro elemento ‘Ana’ foi possível realizar combinações com todos os demais, já para ‘Bianca’ as combinações foram feitas com todos elementos exceto ‘Ana’, onde já existia tal combinação, e assim por diante.

Apenas salientando um ponto importante, todos os elementos onde houverem mais de uma combinação com os mesmos serão retornados ainda como lista, o último par de combinações, nesse exemplo ‘Franciele’ e ‘Maria’ serão retornados em formato de tupla.


Print output (drag lower right corner to resize)
('Ana', 'Bianca')
('Ana', 'Carla')
('Ana', 'Daniela')
('Ana', 'Franciele')
('Ana', 'Maria')
('Bianca', 'Carla')
('Bianca', 'Daniela')
('Bianca', 'Franciele')
('Bianca', 'Maria')
('Carla', 'Daniela')
('Carla', 'Franciele')
('Carla', 'Maria')
('Daniela', 'Franciele')
('Daniela', 'Maria')
('Franciele', 'Maria')
Frames
Objects
Global frame
combinations
lista10
combinacoes
combinations class
show attributes
list
0 "Ana"
1 "Bianca"
2 "Carla"
3 "Daniela"
4 "Franciele"
5 "Maria"
tuple
0 "Franciele"
1 "Maria"

permutations( )

Como visto no tópico anterior, realizar combinações pode meio de combinations( ) é eficiente para alguns contextos, porém possui a grande limitação de não retornar as repetições de combinações, o que pode nos ser útil em certos casos.

Nesses casos, o mais adequado é realizar o mesmo processo, mas por meio da função permutations( ).

```
1  from itertools import permutations
2
3  lista10 = ['Ana', 'Bianca', 'Carla', 'Daniela', 'Franciele', 'Maria']
4
5  for combinacoes in permutations(lista10, 2):
6      print(combinacoes)
7
```

Reaproveitando praticamente todo exemplo anterior, a primeira mudança é a troca de função, de combinations( ) para permutations( ) no momento da importação e uso. O resto continua exatamente igual, inclusive a lógica de parametrização para tais funções.

```
('Ana', 'Bianca')
('Ana', 'Carla')
('Ana', 'Daniela')
('Ana', 'Franciele')
('Ana', 'Maria')
('Bianca', 'Ana')
('Bianca', 'Carla')
('Bianca', 'Daniela')
('Bianca', 'Franciele')
('Bianca', 'Maria')
('Carla', 'Ana')
('Carla', 'Bianca')
('Carla', 'Daniela')
('Carla', 'Franciele')
('Carla', 'Maria')
('Daniela', 'Ana')
('Daniela', 'Bianca')
('Daniela', 'Carla')
('Daniela', 'Franciele')
('Daniela', 'Maria')
('Franciele', 'Ana')
('Franciele', 'Bianca')
('Franciele', 'Carla')
('Franciele', 'Daniela')
('Franciele', 'Maria')
('Maria', 'Ana')
('Maria', 'Bianca')
('Maria', 'Carla')
('Maria', 'Daniela')
('Maria', 'Franciele')
```

Executando esse bloco, agora temos um retorno mais interessante, mostrando todas as possíveis combinações entre os elementos, inclusive as combinações que se repetem quando apenas invertemos a ordem dos mesmos.

Print output (drag lower right corner to resize)

```
('Ana', 'Bianca')
('Ana', 'Carla')
('Ana', 'Daniela')
('Ana', 'Franciele')
('Ana', 'Maria')
('Bianca', 'Ana')
('Bianca', 'Carla')
('Bianca', 'Daniela')
('Bianca', 'Franciele')
('Bianca', 'Maria')
('Carla', 'Ana')
('Carla', 'Bianca')
('Carla', 'Daniela')
('Carla', 'Franciele')
('Carla', 'Maria')
('Daniela', 'Ana')
('Daniela', 'Bianca')
('Daniela', 'Carla')
('Daniela', 'Franciele')
('Daniela', 'Maria')
('Franciele', 'Ana')
('Franciele', 'Bianca')
('Franciele', 'Carla')
('Franciele', 'Daniela')
('Franciele', 'Maria')
('Maria', 'Ana')
('Maria', 'Bianca')
('Maria', 'Carla')
('Maria', 'Daniela')
('Maria', 'Franciele')
```



product( )

Entendidos os tópicos anteriores, onde realizamos as combinações entre os elementos de uma lista, primeiro realizando a puta combinação dos mesmos, sem repetições, posteriormente permitindo repetições por meio das funções combinations( ) e permutations( ) respectivamente.

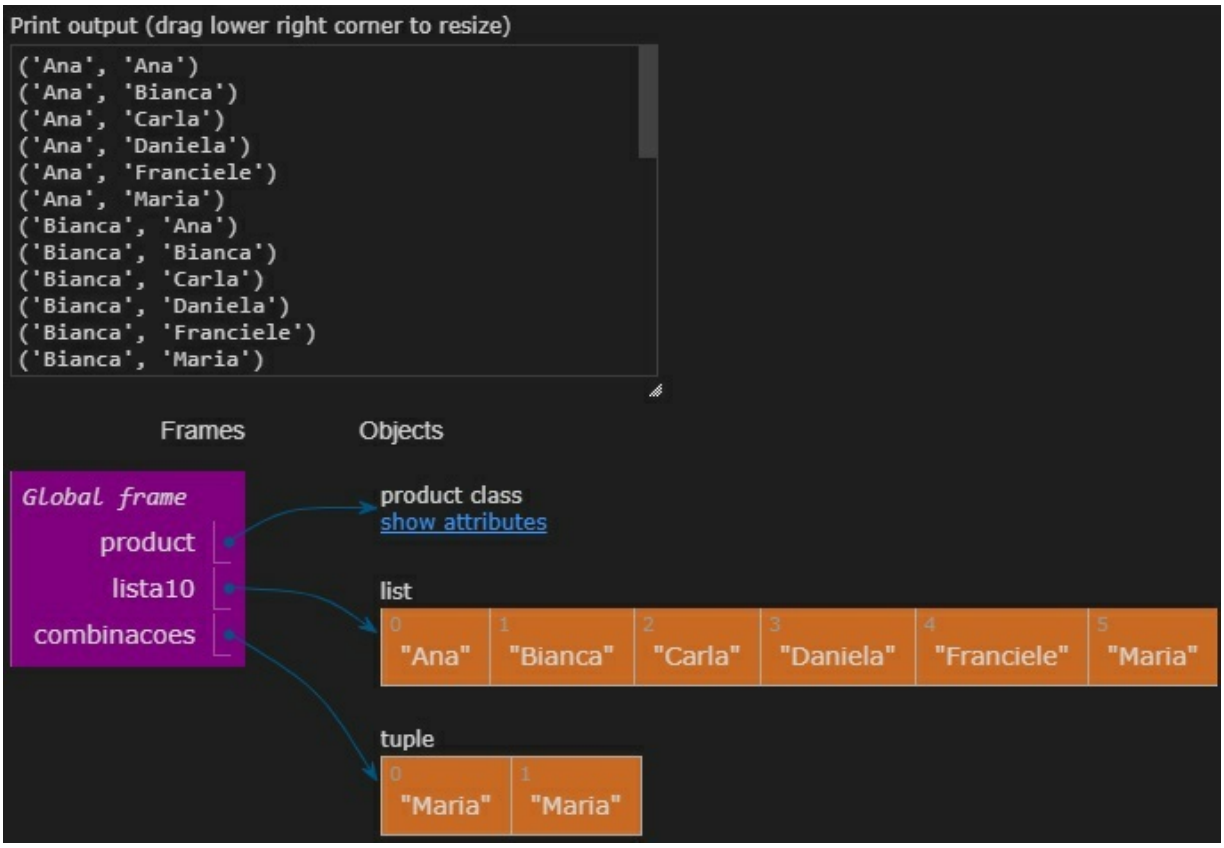
Porém aqui nos deparamos com uma situação onde nenhuma das funções permitia a combinação de um elemento com ele mesmo, e nesses casos isso é possível por meio da função product( ).

```
1  from itertools import product
2
3  lista10 = ['Ana', 'Bianca', 'Carla', 'Daniela', 'Franciele', 'Maria']
4
5  for combinacoes in product(lista10, repeat = 2):
6      print(combinacoes)
7
```

Seguindo com o mesmo exemplo anterior, feita a importação e a substituição de permutations( ) por product( ), nosso resultado deverá ser ainda mais robusto.

```
('Ana', 'Ana')
('Ana', 'Bianca')
('Ana', 'Carla')
('Ana', 'Daniela')
('Ana', 'Franciele')
('Ana', 'Maria')
('Bianca', 'Ana')
('Bianca', 'Bianca')
('Bianca', 'Carla')
('Bianca', 'Daniela')
('Bianca', 'Franciele')
('Bianca', 'Maria')
('Carla', 'Ana')
('Carla', 'Bianca')
('Carla', 'Carla')
('Carla', 'Daniela')
('Carla', 'Franciele')
('Carla', 'Maria')
('Daniela', 'Ana')
('Daniela', 'Bianca')
('Daniela', 'Carla')
('Daniela', 'Daniela')
('Daniela', 'Franciele')
('Daniela', 'Maria')
('Franciele', 'Ana')
('Franciele', 'Bianca')
('Franciele', 'Carla')
('Franciele', 'Daniela')
('Franciele', 'Franciele')
('Franciele', 'Maria')
('Maria', 'Ana')
('Maria', 'Bianca')
('Maria', 'Carla')
('Maria', 'Daniela')
('Maria', 'Franciele')
('Maria', 'Maria')
```

Como esperado, agora temos literalmente todas as possíveis combinações entre os elementos. Desde os elementos uns com os outros, aceitando quando tais combinações existem tanto na inversão da ordem de combinação quanto quando um certo elemento pode ser combinado com ele próprio.


Print output (drag lower right corner to resize)
('Ana', 'Ana')
('Ana', 'Bianca')
('Ana', 'Carla')
('Ana', 'Daniela')
('Ana', 'Franciele')
('Ana', 'Maria')
('Bianca', 'Ana')
('Bianca', 'Bianca')
('Bianca', 'Carla')
('Bianca', 'Daniela')
('Bianca', 'Franciele')
('Bianca', 'Maria')
Frames
Objects
Global frame
product
lista10
combinacoes
product class
show attributes
list
0 "Ana"
1 "Bianca"
2 "Carla"
3 "Daniela"
4 "Franciele"
5 "Maria"
tuple
0 "Maria"
1 "Maria"

TÓPICOS AVANÇADOS EM POO

Métodos de classe

O paradigma da orientação a objetos, independentemente da linguagem de programação ao qual estamos trabalhando, costuma ser uma grande barreira a ser superada pois sua estrutura lógica em parte é mais complexa se comparada com o viés estruturado de programação.

Na parte de orientação a objetos, criadas nossas primeira estruturas de classe, temos funções que podem ser criadas dentro de suas estruturas de dados, de forma a interagir com estruturas tanto de dentro quanto fora da classe, uma vez que toda e qualquer classe, além de um molde, deve ter funcionalidade que justifique sua incorporação no código.

No âmbito das estruturas de código internas a uma classe, temos funções, que por convenção chamamos de métodos de classe, que por sua vez podem ser declaradas de forma “automática”, onde algumas estruturas são internas e padrão para compatibilidade de interação com o método em si, mas também podemos manualmente especificar algumas regras de funcionamento de um método de classe.

Por meio do decorador @classmethod podemos, por exemplo, definir manualmente que um determinado método de classe interage livremente com outros métodos de classe que possam existir, mas obrigatoriamente retornando dados para o escopo global da classe.

```
1  class Pessoa:
2    ano_atual = 2020
3
4    def __init__(self, nome, idade):
5      self.nome = nome
6      self.idade = idade
7
8    @classmethod
9    def ano_nasc(cls, nome, ano_nascimento):
10     idade = cls.ano_atual - ano_nascimento
11     return cls(nome, idade)
12
13 pessoa2 = Pessoa.ano_nasc('Fernando', 1987)
14
15 print(pessoa2.idade)
16
```

Para que tais conceitos fiquem mais claros, vamos partir para o código. Inicialmente é criada uma classe de nome Pessoa, dentro da mesma, em seu escopo global é criado um objeto de classe de nome ano_atual que recebe como atributo um número inteiro. Este objeto, por estar situado no escopo global dessa classe, pode ser usado por qualquer método de classe.

Na sequência é criado um método construtor __init__ que recebe um nome e uma idade, atribuindo os respectivos dados aos seus devidos objetos.

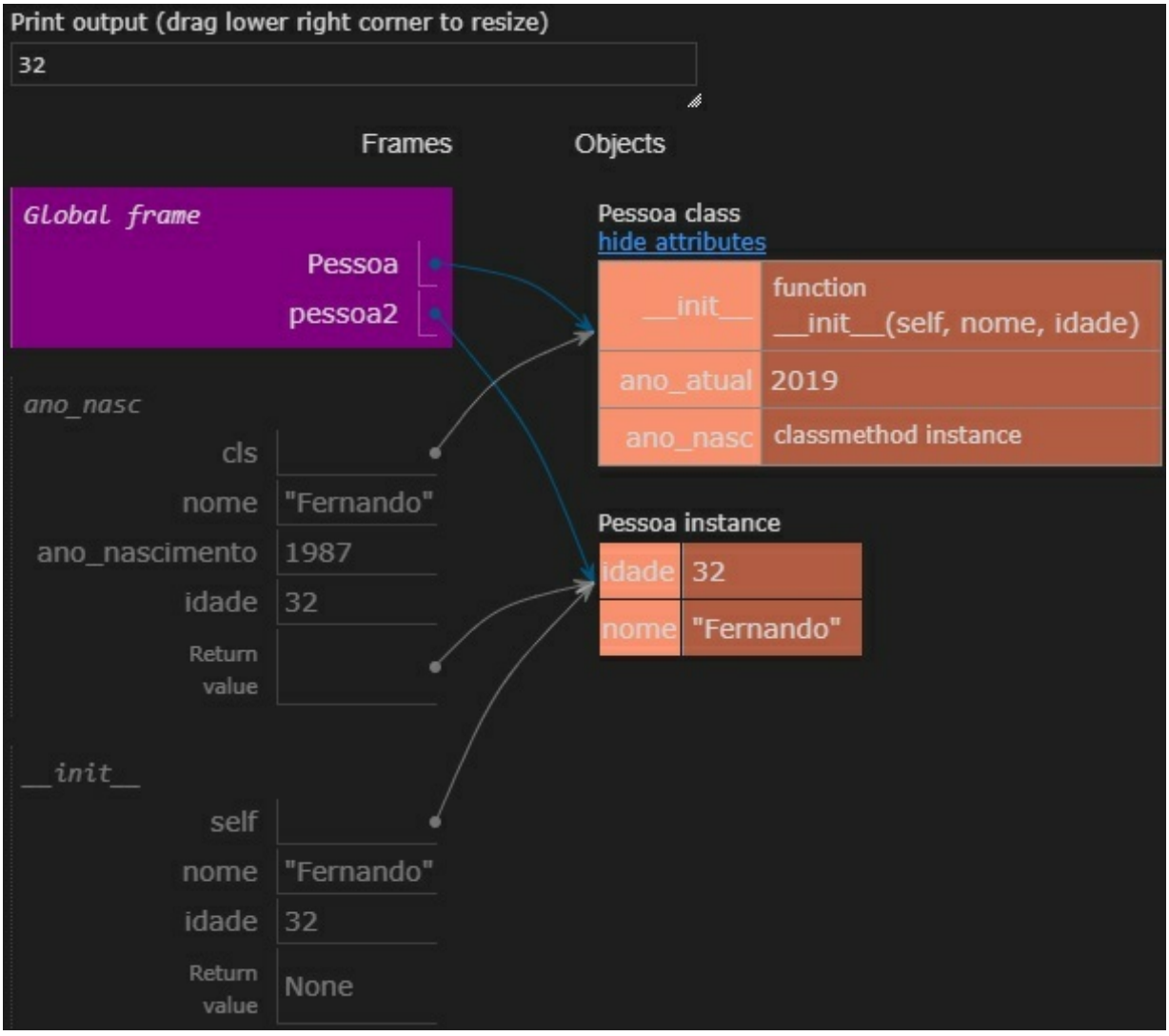
Em seguida é criado um método de classe de nome ano_nasc( ), que por sua vez recebe três atributos de classe, cls, nome e ano_nascimento. Dentro de sua estrutura é criado um objeto de nome idade que recebe como atributo cls que por sua vez fará a subtração dos valores de ano)atual e ano_nascimento, retornando tais dados tanto para nome quanto para idade, que ambos compõe o método construtor da classe.

Repare que antes da declaração do método de classe ano_nasc( ) é colocado um decorador @classmethod, pela leitura léxica de nosso interpretador, o método de classe que estiver associado a este decorador obrigatoriamente deverá retornar um dado/valor para o escopo global da classe.

Fora da classe é criada uma variável de nome pessoa2, que instancia a

classe Pessoa, chamando a função ano_nasc( ) parametrizando a mesma com ‘Fernando’ e 1987. Esses dados serão atribuídos para nome e ano_nascimento, onde feitas as devidas interações, a diferença entre o valor de ano_atual e de ano_nascimento será atribuída para idade.

Repare no nível de interações que ocorre dentro da classe, entre os dados repassados pelo usuário e a função ano_nasc( ) que por sua vez interage tanto com o escopo global da classe quanto com a variável que está fora dela. Uma das formas de garantir que tais interações ocorram corretamente é fazendo o uso de um decorador, neste caso, @classmethod.



Métodos estáticos

Logo que iniciamos nossos estudos em orientação a objetos nos é passado, além é claro da sintaxe, algumas estruturas bastante engessadas para que possamos criar nossos objetos de forma que os mesmos sempre estejam atrelados a métodos e atributos de classe.

Porém, a partir do momento que começamos a ver nossas classes como uma estrutura “molde” para outras estruturas de objetos, começamos a entender que praticamente tudo o que se é permitido fazer na programação estruturada é permitido fazer na orientada a objetos.

Falando especificamente de nossos métodos de classe, por padrão temos o costume de já criá-los atrelados a um objeto ou de forma a ser instanciado tanto dentro quanto fora de nossa classe.

Mas, nesse cenário acabamos por esquecer que um método de classe, assim como uma função comum, pode ser totalmente independente, sendo chamada arbitrariamente apenas quando necessário.

Em outras palavras, um método estático funciona como uma função normal dentro de uma classe, independente, podendo ser utilizada (inclusive recursivamente) sem que a mesma seja instanciada para a classe ou por algum objeto da mesma.

Nessa linha de raciocínio, diferentemente de um método de classe convencional que normalmente opera em self, um método estático quando declarado, recebe ou não parâmetros como em uma função comum.

Por parte da notação em Python, podemos especificar que um determinado método de classe é estático por meio do decorador @staticmethod.

```python
from random import randint

class Usuario:
    @staticmethod
    def gerador_id():
        gerador = randint(100, 999)
        return gerador

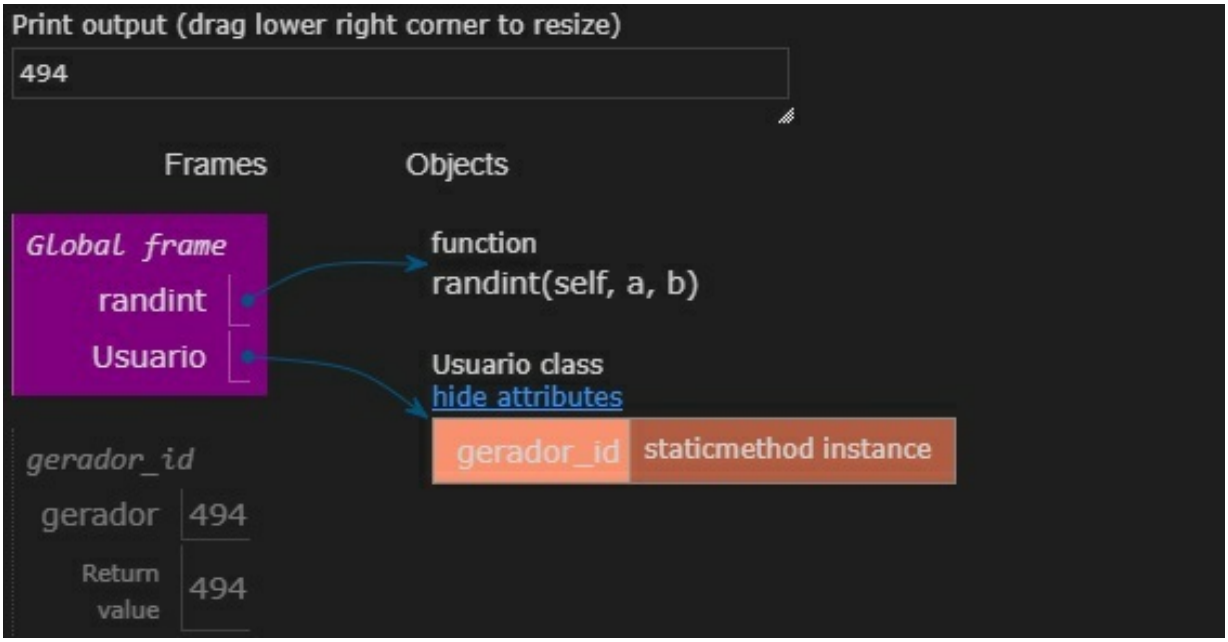
print(Usuario.gerador_id())

```

Para esse exemplo, vamos simplesmente supor que temos um simples gerador de números de usuário. Para isso importamos o módulo randint da biblioteca random.

Em seguida criamos uma classe de nome Usuário, é colocado o decorador @staticmethod identificando que o método de classe gerador_id( ) é estático, ou seja, tem o comportamento de uma função independente.

Observando sua sintaxe, note que a mesma não possui parâmetros, não retorna nada para nenhum objeto da classe ou de fora dela, é um simples gerador de números aleatórios “encapsulado” dentro de uma classe.



@property, @getters e @setters

Certamente em seus estudos de Python básico, você deve ter reparado que uma das particularidades mais interessantes dessa linguagem é o fato da mesma ter muita coisa automatizada poupando nosso tempo na construção de certas estruturas de código.

Diferentemente de outras linguagens onde é necessário declarar o escopo, a visibilidade, o tipo de variável, entre outras coisas manualmente, Python já nos oferece de a partir da primeira linha de código nos focar no código em si.

Sem sombra de dúvidas este é um dos maiores atrativos desta linguagem de programação tanto para iniciantes quanto para programadores que estão migrando de outras linguagens.

Automatizar certos processos não necessariamente significa que não temos o controle de tais parâmetros de nosso código, muito pelo contrário, é perfeitamente possível trabalhar em cima dessas estruturas e de forma simples, também pelo fato de que certas aplicações até terão como requisito alguma configuração interna definida de forma manual.

Em outros momentos vimos que podemos, por exemplo, tanto na programação estruturada quanto orientada à objetos, definir a visibilidade de um certo objeto ou função por meio de “_” ou “__”.

Para certas situações, podemos fazer algo parecido em nossos métodos de classe por meio de decoradores “@” como nossos conhecidos @staticmethod e @classmethod, assim como os que veremos a seguir.

Apenas relembrando, em Python temos uma série de palavras reservadas ao sistema assim como símbolos que indicam para nosso interpretador um determinado comportamento de um bloco de código.

Existe uma boa variedade de decoradores que podem ser consultados na documentação Python, aqui vamos nos ater aos mais comuns que são @property, @getter e @setter.

Outra coisa a ser mencionada já neste momento, tenha em mente

que basicamente a aplicação mais comum destes decoradores são em estruturas de código onde se faz necessário o uso de alguma estrutura de validação. Realizaremos algo equivalente a um sistema validador porém aplicado diretamente como método de uma classe.

Dentro dessa lógica, basicamente ao colocar o decorador @property nosso interpretador já sabe que para aquele bloco de código existem estruturas com configurações e validadores definidos manualmente. @getter por sua vez funcionará como a função que irá obter um certo dado/valor a ser validado, e @setter será a função que irá realizar as devidas conversões de tipos de dados e os validar.

Um último adendo, o uso de decoradores internamente tem o poder de modificar a ordem da leitura dos métodos de classe.

Em outras palavras, apesar de nosso interpretador possuir regras quanto sua leitura léxica, quando criado algum decorador em meio ao código ele recebe prioridade de leitura e execução.

```
1  class Produto:
2      def __init__(self, nome, preco = 0):
3          self.nome = nome
4          self.preco = preco
5
6      def aplica_desconto(self, percentual):
7          self.preco = self.preco - (self.preco*(percentual/100))
8
```

Partindo para prática, vamos supor que estamos criando um simples sistema de cadastro de produtos de uma loja, onde os funcionários tem acesso a um breve cadastro apenas especificando o nome e o preço de um determinado item e também a uma simples função que aplica um desconto ao item.

Para isso inicialmente criamos nossa classe Produto, dentro de si é criado um método construtor __init__ que simplesmente recebe um nome e um preco, aqui apenas por conversão já definido manualmente como 0.

Em seguida são criados os objetos referentes a nome e preco.

Na sequência é criado o método de classe aplica_desconto( ) que por sua vez receberá um percentual. Essa função simplesmente pega o valor

atribuído a preco e aplica uma simples fórmula de desconto.

```
 8  produto1 = Produto('Camiseta', 99)
 9  produto1.aplica_desconto(5)
10  print(produto1.preco)
11
```

Até aqui nada de novo, inclusive se os usuários usarem o programa como é esperado, não existirão erros em nenhum dos processos.

Porém, lembre-se que uma boa prática de programação é tentar prever as situações onde o usuário pode explorar alguma função de forma errada, e a partir disso gerar sistemas de validação que contornem qualquer exceção cometida pelo usuário.

Nesse caso, uma das coisas mais comuns seria um usuário repassar o valor do item em um tipo de dado diferente. Por exemplo, ao invés dos 99 (int) o mesmo cadastrar ‘99’(string). É em cima deste tipo de situação que criaremos nossas estruturas de validação, nesse caso, por meio de @getters e @setters.

```
 9    #Getter
10    @property
11    def preco(self):
12      return self._preco
13
14    #Setter
15    @preco.setter
16    def preco(self, preco_validado):
17      if isinstance(preco_validado, str):
18        preco_validado = float(preco_validado.replace('R$', ''))
19      self._preco = preco_validado
20
```

Criando os validadores, inicialmente colocamos nosso decorador @property para o método de classe preco( ), que por sua vez simplesmente retornará _preco.

Na sequência é criado nosso @setter, onde é criado o decorador @preco.setter para o método de classe preco( ), que agora recebe um preco_validado.

Repare que estamos criando virtualmente duas funcionalidades em

cima de um método de classe, hora ele se comportará como estrutura para captação dos dados/valores a serem validados, hora ele se comportará como o validador de tais dados.

Em seguida, dentro de nosso @setter, é criada uma estrutura condicional onde se o dado/valor atribuído a preco_validado for do tipo string, o mesmo será convertido para float.

Aqui apenas para construir algo mais didático também é feita a conversão sintática por meio da função replace( ), supondo que os caracteres especiais R$ devem ser eliminados.

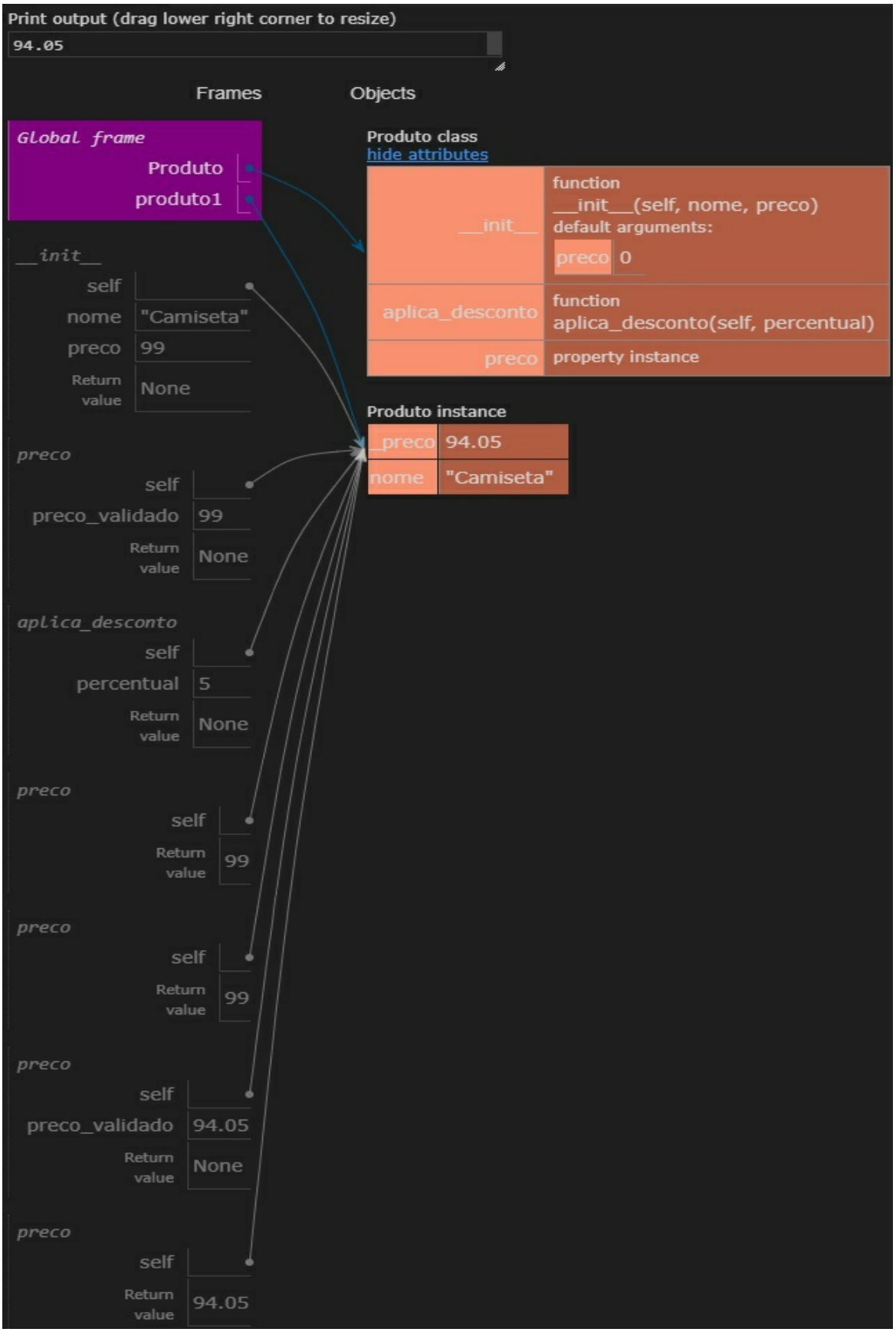
Por fim, o valor de _preco é atualizado com o valor de preco_validado.

```
21  produto1 = Produto('Camiseta', 99)
22  produto1.aplica_desconto(5)
23  print(produto1.preco)
24
25  produto2 = Produto('Calça', 'R$59')
26  produto2.aplica_desconto(15)
27  print(f'Produto2 após aplicação do desconto terá valor de {produto2.preco}')
28
```

A partir disso podemos cadastrar nossos produtos com os referentes preços, independentemente da maneira como o usuário digitar o preço, internamente antes mesmo desse valor ser atribuído para preco ele será validado como número do tipo float.

```
94.05
Produto2 após aplicação do desconto terá valor de 50.15
```

E o retorno para cada função print( ).


Print output (drag lower right corner to resize)
94.05
Frames
Objects
Global frame
Produto
produto1
Produto class
hide attributes
__init__
function
__init__(self, nome, preco)
default arguments:
preco 0
aplica_desconto
function
aplica_desconto(self, percentual)
preco
property instance
Produto instance
_preco 94.05
nome "Camiseta"
__init__
self
nome "Camiseta"
preco 99
Return value None
preco
self
preco_validado 99
Return value None
aplica_desconto
self
percentual 5
Return value None
preco
self
Return value 99
preco
self
Return value 99
preco
self
preco_validado 94.05
Return value None
preco
self
Return value 94.05

Associação de Classes

Associação de classes, ou entre classes, é uma prática bastante comum uma vez que nossos programas dependendo sua complexidade não se restringem a poucas funções básicas, dependendo do que o programa oferece ao usuário uma grande variedade de funções internas são executadas ou ao menos estão à disposição do usuário para que sejam executadas.

Internamente estas funções podem se comunicar e até mesmo trabalhar em conjunto, assim como independentes elas não dependem uma da outra.

Raciocine que esta prática é muito utilizada pois quando se está compondo o código de um determinado programa é natural criar uma função para o mesmo e, depois de testada e pronta, esta pode ser modularizada, ou ser separada em um pacote, etc... de forma que aquela estrutura de código, embora parte do programa, independente para que se faça a manutenção da mesma assim como novas implementações de recursos.

É normal, dependendo da complexidade do programa, que cada “parte” de código seja na estrutura dos arquivos individualizada de forma que o arquivo que guarda a mesma quando corrompido, apenas em último caso faça com que o programa pare de funcionar.

Se você já explorou as pastas dos arquivos que compõe qualquer programa deve ter se deparado com milhares de arquivos dependendo o programa.

A manutenção dos mesmos se dá diretamente sobre os arquivos que apresentam algum problema.

```
1  class Usuario:
2      def __init__(self, nome):
3          self.__nome = nome
4          self.__logar = None
5      @property
6      def nome(self):
7          return self.__nome
8      @property
9      def logar(self):
10         return self.__logar
11     @logar.setter
12     def logar(self, logar):
13         self.__logar = logar
14
```

Partindo para a prática, note que em primeiro lugar criamos nossa classe Usuario, a mesma possui um método construtor que recebe um nome (com variável homônima declarada como privada), em seguida precisamos realizar a criação de um getter e setter para que possamos instanciar os referentes métodos de fora dessa classe.

Basicamente, pondo em prática tudo o que já vimos até então, esta seria a forma adequada de permitir acesso aos métodos de classe de Usuario, quando as mesmas forem privadas ou protegidas.

```
15 class Identificador:
16     def __init__(self, numero):
17         self.__numero = numero
18     @property
19     def numero(self):
20         return self.__numero
21     def logar(self):
22         print('Logando no sistema...')
23
```

Na sequência criamos uma nova classe de nome Identificador que aqui, apenas como exemplo, seria o gatilho para um sistema de autenticação de usuário em um determinado sistema.

Logo, a mesma também possui um método construtor que recebe

um número para identificação do usuário, novamente, é criado um getter e um setter para tornar o núcleo desse código instanciável.

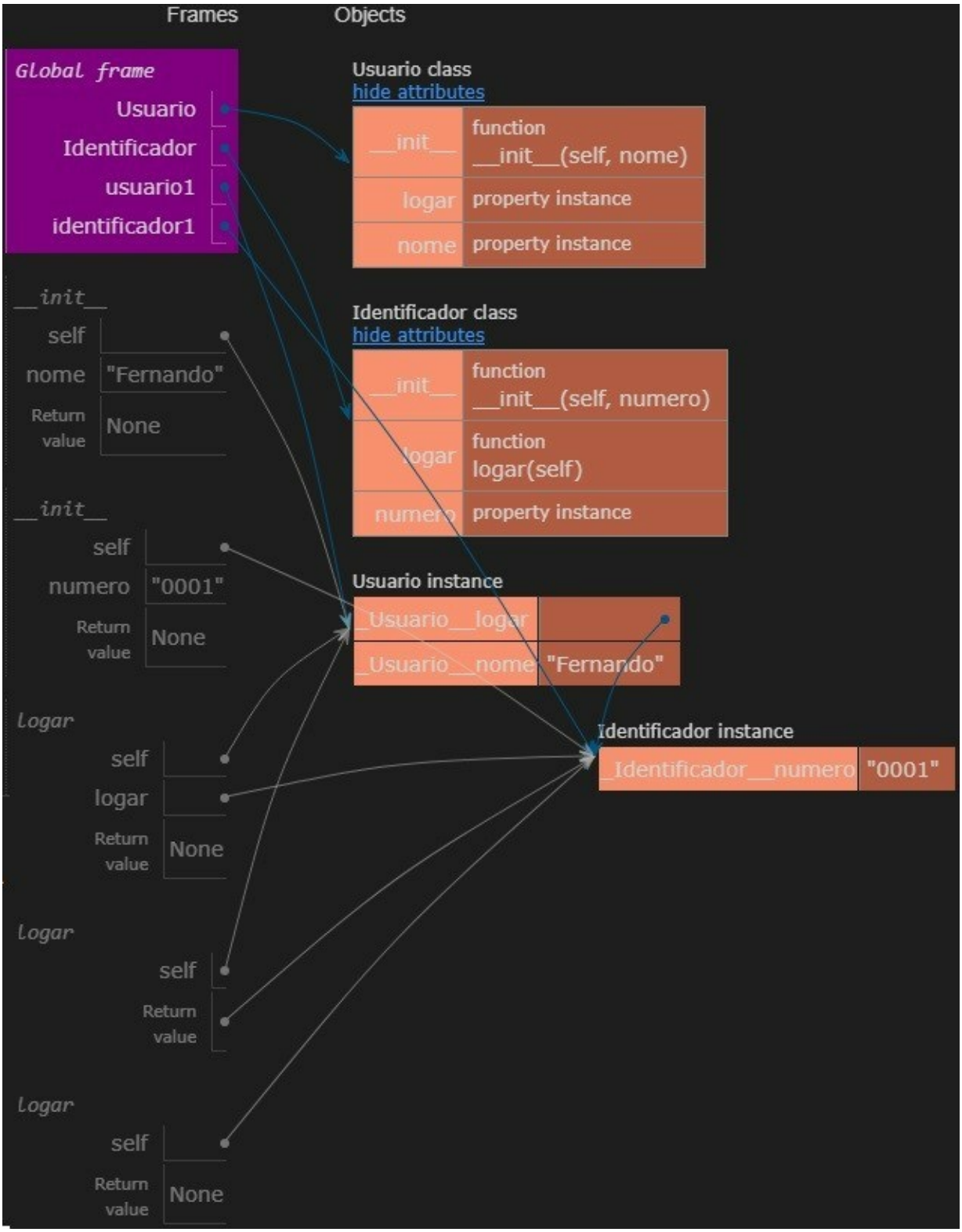
```
24  usuario1 = Usuario('Fernando')
25  identificador1 = Identificador('0001')
26
27  usuario1.logar = identificador1
28  usuario1.logar.logar()
29
```

Por fim, é criado um objeto no corpo de nosso código de nome usuario1 que instancia a classe Usuario, lhe passando como atributo ‘Fernando’.

O mesmo é feito para o objeto identificador1, que instancia a classe Identificador passando como parâmetro ‘0001’. Agora vem a parte onde estaremos de fato associando essas classes e fazendo as mesmas trabalhar em conjunto.

Para isso por meio de usuário.logar = identificador associamos 0001 a Fernando, e supondo que essa é a validação necessária para esse sistema de identificação (ou pelo menos uma das etapas).

Por meio da função usuario1.logar.logar( ) o retorno é: Logando no sistema...



Raciocine que este exemplo básico é apenas para entender a lógica de associação, obviamente um sistema de autenticação iria exigir mais dados como senha, etc... assim como confrontar esses dados com um banco de

dados...

Também é importante salientar que quanto mais associações houver em nosso código, de forma geral mais eficiente é a execução do mesmo ao invés de blocos e mais blocos de códigos independentes e repetidos para cada situação, porém, quanto mais robusto for o código, maior também será a suscetibilidade a erro, sendo assim, é interessante testar o código a cada bloco criado para verificar se não há nenhum erro de lógica ou sintaxe.

Agregação e Composição de Classes

Começando o entendimento pela sua lógica, anteriormente vimos o que basicamente é a associação de classes, onde temos mais de uma classe, elas podem compartilhar métodos e atributos entre si ao serem instanciadas, porém uma pode ser executada independentemente da outra.

Já quando falamos em agregação e composição, o laço entre essas classes e suas estruturas passa a ser tão forte que uma depende da outra.

Raciocine que quando estamos realizando a composição de uma classe, uma classe tomará posse dos objetos das outras classes, de modo que se algo corromper essa classe principal, as outras param de funcionar também.

```
1  class Contato:
2      def __init__(self, residencial, celular):
3          self.residencial = residencial
4          self.celular = celular
5
6  class Cliente:
7      def __init__(self, nome, idade, fone=None):
8          self.nome = nome
9          self.idade = idade
10         self.fone = []
11
12     def addFone(self, residencial, celular):
13         self.fone.append(Contato(residencial, celular))
14     def listaFone(self):
15         for fone in self.fone:
16             print(fone.residencial, fone.celular)
17
```

Para esse exemplo, inicialmente criamos uma classe de nome Cliente, ela possui seu método construtor que recebe um nome, uma idade e um ou mais telefones para contato, isto porque como objeto de classe fone inicialmente recebe uma lista em branco.

Em seguida é criado um método de classe responsável por alimentar essa lista com números de telefone.

Porém, repare no diferencial, em self.fone, estamos adicionando dados instanciando como parâmetro outra classe, nesse caso a classe Contato com toda sua estrutura declarada anteriormente.

Como dito anteriormente, este laço, instanciando uma classe dentro de outra classe é um laço forte onde, se houver qualquer exceção no código, tudo irá parar de funcionar uma vez que tudo está interligado operando em conjunto.

```
18  cliente1 = Cliente('Fernando', 32)
19  cliente1.addFone(33221766, 991357258)
20  print(cliente1.nome)
21  print(cliente1.listaFone())
22
```

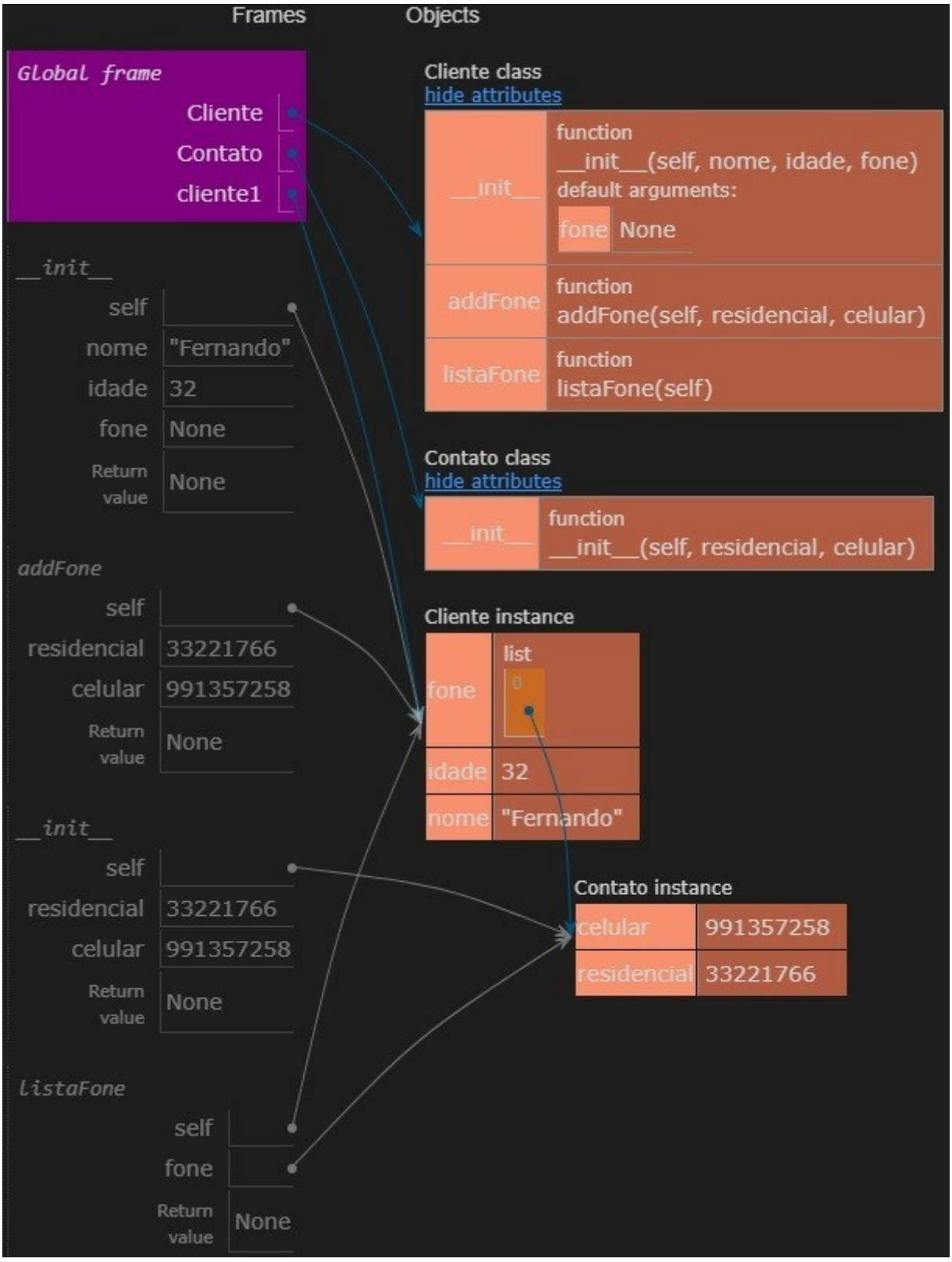
Seguindo com o exemplo, é criada uma variável de nome cliente1 que instancia Cliente passando como atributos de classe Fernando, 32.

Em seguida é chamada a função addFone( ) passando como parâmetros 33221766 e 991357258.

Por fim, por meio do comando print( ) pedimos que sejam exibidos os dados de cliente1.nome de cliente1.listaFone( ).

```
Fernando
33221766 991357258
None
```

Neste caso o retorno é, como esperado, os dados demonstrados na imagem acima.


Frames
Objects
Global frame
Cliente
Contato
cliente1
__init__
self
nome
"Fernando"
idade
32
fone
None
Return value
None
addFone
self
residencial
33221766
celular
991357258
Return value
None
__init__
self
residencial
33221766
celular
991357258
Return value
None
listaFone
self
fone
Return value
None
Cliente class
hide attributes
__init__
function
__init__(self, nome, idade, fone)
default arguments:
fone
None
addFone
function
addFone(self, residencial, celular)
listaFone
function
listaFone(self)
Contato class
hide attributes
__init__
function
__init__(self, residencial, celular)
Cliente instance
fone
list
0
idade
32
nome
"Fernando"
Contato instance
celular
991357258
residencial
33221766

Herança

Na estrutura de código de um determinado programa todo orientado a objetos, é bastante comum que algumas classes em teoria possuam exatamente as mesmas características que outras, porém isso não é nada eficiente no que diz respeito a quantidade de linhas de código uma vez que esses blocos, já que são idênticos, estão repetidos no código. Por Exemplo:

```
1  class Corsa:
2      def __init__(self, nome, ano):
3          self.nome = nome
4          self.ano = ano
5
6  class Gol:
7      def __init__(self, nome, ano):
8          self.nome = nome
9          self.ano = ano
10
```

Dado o exemplo acima, note que temos duas classes para dois carros diferentes, Corsa e Gol, e repare que as características que irão identificar os mesmos, aqui nesse exemplo bastante básico, seria um nome e o ano do veículo. Duas classes com a mesma estrutura de código repetida duas vezes.

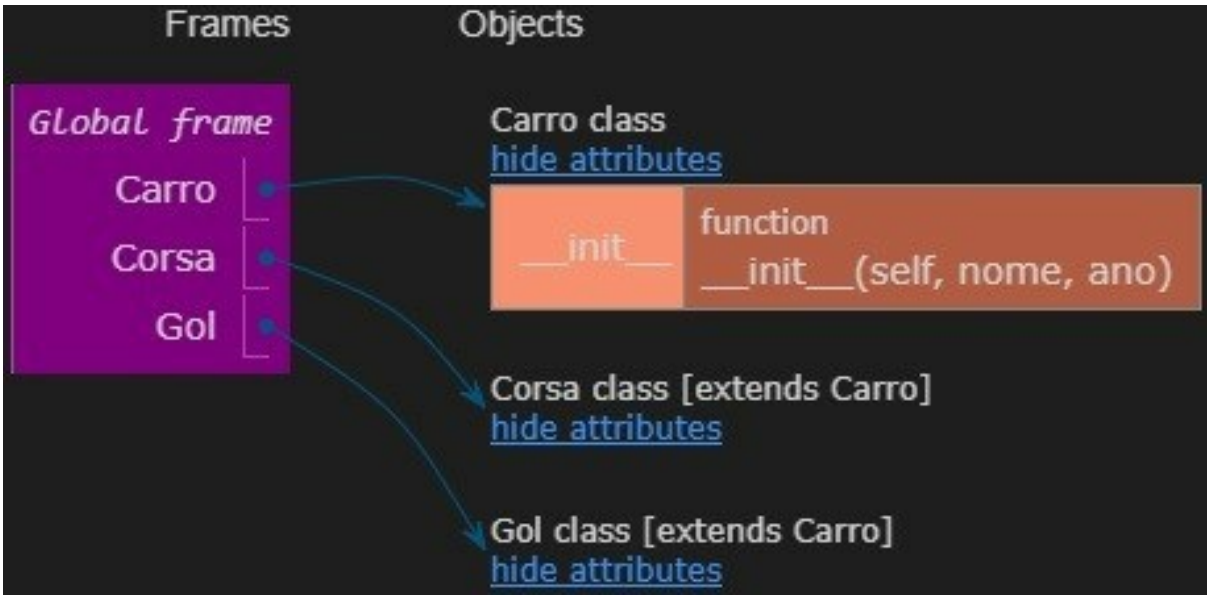
Podemos otimizar esse processo criando uma classe molde de onde serão lidas e interpretadas todas essas características de forma otimizada por nosso interpretador.

Vale lembrar que neste exemplo são apenas duas classes, não impactando na performance, mas numa aplicação real pode haver dezenas ou centenas de classes idênticas em estrutura, o que acarreta em lentidão se lidas uma a uma de forma léxica.

```
11  class Carro:
12      def __init__(self, nome, ano):
13          self.nome = nome
14          self.ano = ano
15
16  class Corsa(Carro):
17      pass
18
19  class Gol(Carro):
20      pass
21
```

Reformulando o código, é criada uma classe de nome Carro que servirá de molde para as outras, dentro de si ela tem um método construtor assim como recebe como atributos de classe um nome e um ano.

A partir de agora as classes Corsa e Gol simplesmente recebem como parâmetro a classe Carro, recebendo toda sua estrutura. Internamente o interpretador não fará distinção e assumirá que cada classe é uma classe normal, independente, inclusive com alocações de memória separadas para cada classe, porém a estrutura do código dessa forma é muito mais enxuta.



Na literatura, principalmente autores que escreveram por versões anteriores do Python costumavam chamar esta estrutura de classes e subclasses, apenas por convenção, para que seja facilitada a visualização da

hierarquia das mesmas.

Vale lembrar também que uma “subclasse” pode ter mais métodos e atributos particulares a si, sem problema nenhum, a herança ocorre normalmente a tudo o que estiver codificado na classe mãe, porém as classes filhas desta tem flexibilidade para ter mais atributos e funções conforme necessidade.

Ainda sob o exemplo anterior, a subclasse Corsa, por exemplo, poderia ser mais especializada tendo maiores atributos dentro de si além do nome e ano herdado de Carro.

Cadeia de Heranças

Como visto no tópico anterior, uma classe pode herdar outra sem problema nenhum, desde que a lógica estrutural e sintática esteja correta na hora de definir as mesmas.

Extrapolando um pouco, vale salientar que essa herança pode ocorrer em diversos níveis, na verdade, não há um limite para isto, porém é preciso tomar bastante cuidado para não herdar características desnecessárias que possam tornar o código ineficiente. Ex:

```
1  class Carro:
2      def __init__(self, nome, ano):
3          self.nome = nome
4          self.ano = ano
5
6  class Gasolina(Carro):
7      def __init__(self, tipogasolina=True, tipoalcool=False):
8          self.tipogasolina = tipogasolina
9          self.tipoalcool = tipoalcool
10
11 class Jeep(Gasolina):
12     pass
13
```

Repare que nesse exemplo inicialmente é criada uma classe Carro, dentro de si ela possui um método construtor onde é repassado como atributo de classe um nome e um ano.

Em seguida é criada uma classe de nome Gasolina que herda toda estrutura de Carro e dentro de si define que o veículo desta categoria será do tipo gasolina.

Por fim é criada uma última classe se referindo a um carro marca Jeep, que herda toda estrutura de Gasolina e de Carro pela hierarquia entre classe e subclasses. Seria o mesmo que:

```
class Jeep:
    def carro():
        def __init__(self, nome, ano):
            self.nome = nome
            self.ano = ano

    def gasolina(self, tipogasolina=True, tipoalcool=False):
        self.tipogasolina = tipogasolina
        self.tipoalcool = tipoalcool

jeep = Jeep()

```

Código usual, porém, ineficiente. Assim como estático, de modo que quando houvesse a criação de outros veículos não seria possível reaproveitar nada de dentro dessa Classe a não ser por instâncias de fora da mesma, o que gera muitos caminhos de leitura para o interpretador, tornando a performance do código ruim.

Lembrando que a vantagem pontual que temos ao realizar heranças entre classes é a reutilização de seus componentes internos no lugar de blocos e mais blocos de código repetidos.

```
class Carro:
    def __init__(self, nome, ano):
        self.nome = nome
        self.ano = ano

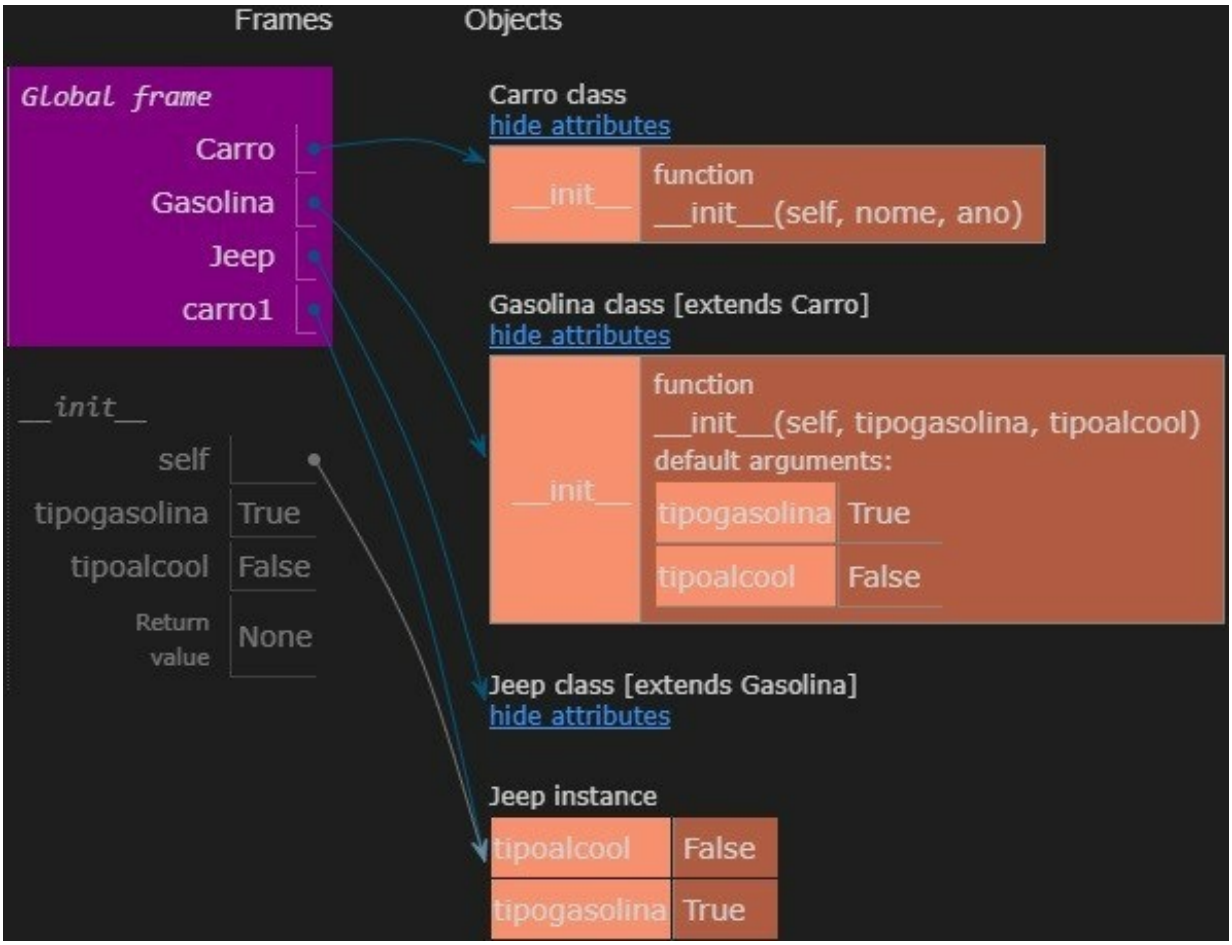
class Gasolina(Carro):
    def __init__(self, tipogasolina=True, tipoalcool=False):
        self.tipogasolina = tipogasolina
        self.tipoalcool = tipoalcool

class Jeep(Gasolina):
    pass

carro1 = Jeep()
print(carro1.tipogasolina)

```

Código otimizado via herança e cadeia de hierarquia simples. Executando nossa função print( ) nesse caso o retorno será: True


Frames
Objects
Global frame
Carro
Gasolina
Jeep
carro1
__init__
self
tipogasolina
True
tipoalcool
False
Return value
None
Carro class
hide attributes
__init__
function
__init__(self, nome, ano)
Gasolina class [extends Carro]
hide attributes
__init__
function
__init__(self, tipogasolina, tipoalcool)
default arguments:
tipogasolina
True
tipoalcool
False
Jeep class [extends Gasolina]
hide attributes
Jeep instance
tipoalcool
False
tipogasolina
True

Herança Múltipla

Entendida a lógica de como uma classe herda toda a estrutura de outra classe, podemos avançar um pouco mais fazendo o que é comumente chamado de herança múltipla.

Raciocine que numa herança simples, criamos uma determinada classe assim como todos seus métodos e atributos de classe, de forma que podíamos instanciar / passar ela como parâmetro de uma outra classe.

Respeitando a sintaxe, é possível realizar essa herança vinda de diferentes classes, importando dessa forma seus conteúdos.

```
1  class Mercadoria:
2      def __init__(self, nome, preco):
3          self.nome = nome
4          self.preco = preco
5
6  class Carnes(Mercadoria):
7      def __init__(self, tipo, peso):
8          self.tipo = tipo
9          self.peso = peso
10
11 class Utensilios:
12     def __init__(self, espetos, carvao):
13         self.espetos = espetos
14         self.carvao = carvao
15
16 class KitChurrasco(Carnes, Utensilios):
17     pass
18
```

Para este exemplo, vamos supor que esse seria o background de um simples sistema de inventário de um mercado, onde as categorias de produtos possuem estrutura em classes distintas, porém que podem ser combinadas para montar “kits” de produtos.

Então note que inicialmente é criada uma classe de nome

Mercadoria, que por sua vez possui um método construtor assim como nome e preco das mercadorias como atributos de classe.

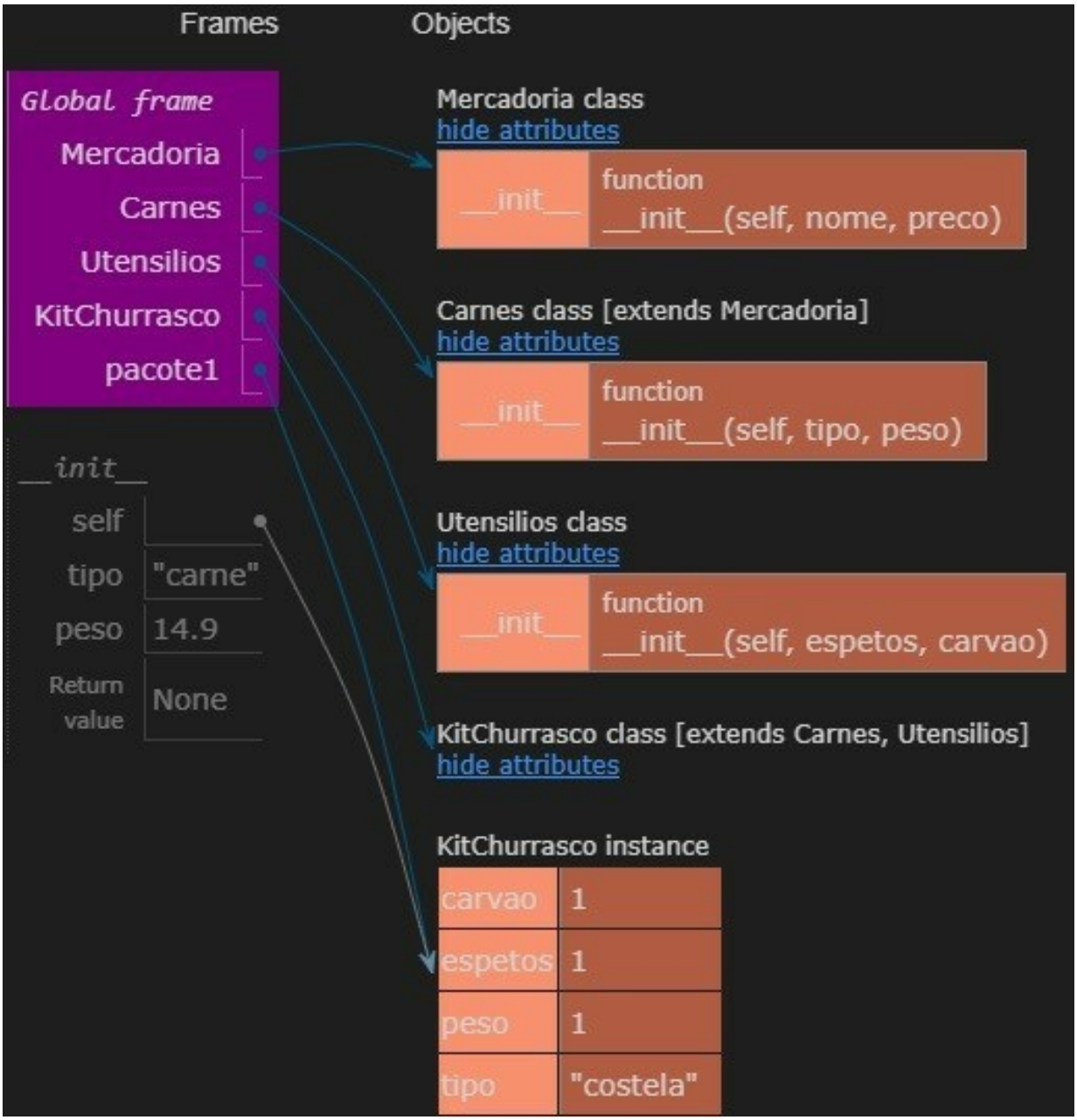
Em seguida é criada uma classe de nome Carnes que herda toda estrutura de Mercadoria e adiciona os atributos de classe referentes ao tipo de carne e seu peso.

Na sequência é criada uma classe de nome Utensilios que não herda nada das classes anteriores, e dentro de si possui estrutura de método construtor assim como seus atributos de classe.

Por fim, é criada uma classe de nome KitChurrasco, que por sua vez herda Carnes (que herda Mercadoria) e Utensilios com todo seu conteúdo.

```
20  pacote1 = KitChurrasco('carne', 14.90)
21  pacote1.tipo = 'costela'
22  pacote1.peso = 1
23  pacote1.espetos = 1
24  pacote1.carvao = 1
25
```

A partir disso, como nos exemplos anteriores, é possível instanciar objetos e definir dados/valores para os mesmos normalmente, uma vez que KitChurrasco agora, devido às características que herdou, possui campos para nome, preco, tipo, peso, espetos e carvao dentro de si.


Frames
Objects
Global frame
Mercadoria
Carnes
Utensilios
KitChurrasco
pacote1
__init__
self
tipo
"carne"
peso
14.9
Return value
None
Mercadoria class
hide attributes
__init__
function
__init__(self, nome, preco)
Carnes class [extends Mercadoria]
hide attributes
__init__
function
__init__(self, tipo, peso)
Utensilios class
hide attributes
__init__
function
__init__(self, espetos, carvao)
KitChurrasco class [extends Carnes, Utensilios]
hide attributes
KitChurrasco instance
carvao
1
espetos
1
peso
1
tipo
"costela"

Sobreposição de Membros

Um dos cuidados que devemos ter ao manipular nossas classes em suas hierarquias e heranças é o fato de apenas realizarmos sobreposições quando de fato queremos alterar/sobrescrever um determinado dado/valor/método dentro da mesma.

Em outras palavras, anteriormente vimos que o interpretador do Python realiza a chamada leitura léxica do código, ou seja, ele lê linha por linha (de cima para baixo) e linha por linha (da esquerda para direita), dessa forma, podemos reprogramar alguma linha ou bloco de código para que na sequência de sua leitura/interpretação o código seja alterado.

```
1  class Pessoa:
2      def __init__(self, nome):
3          self.nome = nome
4
5      def Acao1(self):
6          print(f'{self.nome} está dormindo')
7
8  class Jogador1(Pessoa):
9      def Acao2(self):
10         print(f'{self.nome} está comendo')
11
12 class SaveJogador1(Jogador1):
13     pass
14
```

Inicialmente criamos a classe Pessoa, dentro de si um método construtor que recebe um nome como atributo de classe. Também é definida uma Acao1 que por sua vez por meio da função print( ) exibe uma mensagem.

Em seguida é criada uma classe de nome Jogador1 que herda tudo de Pessoa e por sua vez, apenas tem um método Acao2 que exibe também uma determinada mensagem.

Por fim é criado uma classe SaveJogador1 que herda tudo de

Jogador1 e de Pessoa.

```
16  p1 = SaveJogador1('Fernando')
17  print(p1.nome)
18
```

Trabalhando sobre essa cadeia de classes, criamos uma variável de nome p1 que instancia SaveJogador1 e passa como parâmetro ‘Fernando’, pela hierarquia destas classes, esse parâmetro alimentará self.nome de Pessoa.

```
Fernando
```

Por meio da função print( ) passando como parâmetros p1.nome o retorno é: Fernando.

```
19  p1.Acao1()
20  p1.Acao2()
21
```

Da mesma forma, instanciando qualquer coisa de dentro de qualquer classe da hierarquia, se não houver nenhum erro de sintaxe tudo será executado normalmente.

```
Fernando
Fernando está dormindo
Fernando está comendo
```

Nesse caso, o retorno é o apresentado na imagem acima.

```
class Pessoa:
    def __init__(self, nome):
        self.nome = nome

    def Acao1(self):
        print(f'{self.nome} está dormindo')

class Jogador1(Pessoa):
    def Acao2(self):
        print(f'{self.nome} está comendo')

class SaveJogador1(Jogador1):
    def Acao1(self):
        print(f'{self.nome} está acordado')

```

Por fim, partindo para uma sobreposição de fato, note que agora em SaveJogador1 criamos um método de classe de nome Acao1, dentro de si uma função para exibir uma determinada mensagem.

Repare que Acao1 já existia em Pessoa, mas o que ocorre aqui é que pela cadeia de heranças SaveJogador1 criando um método Acao1 irá sobrepor esse método já criado anteriormente.

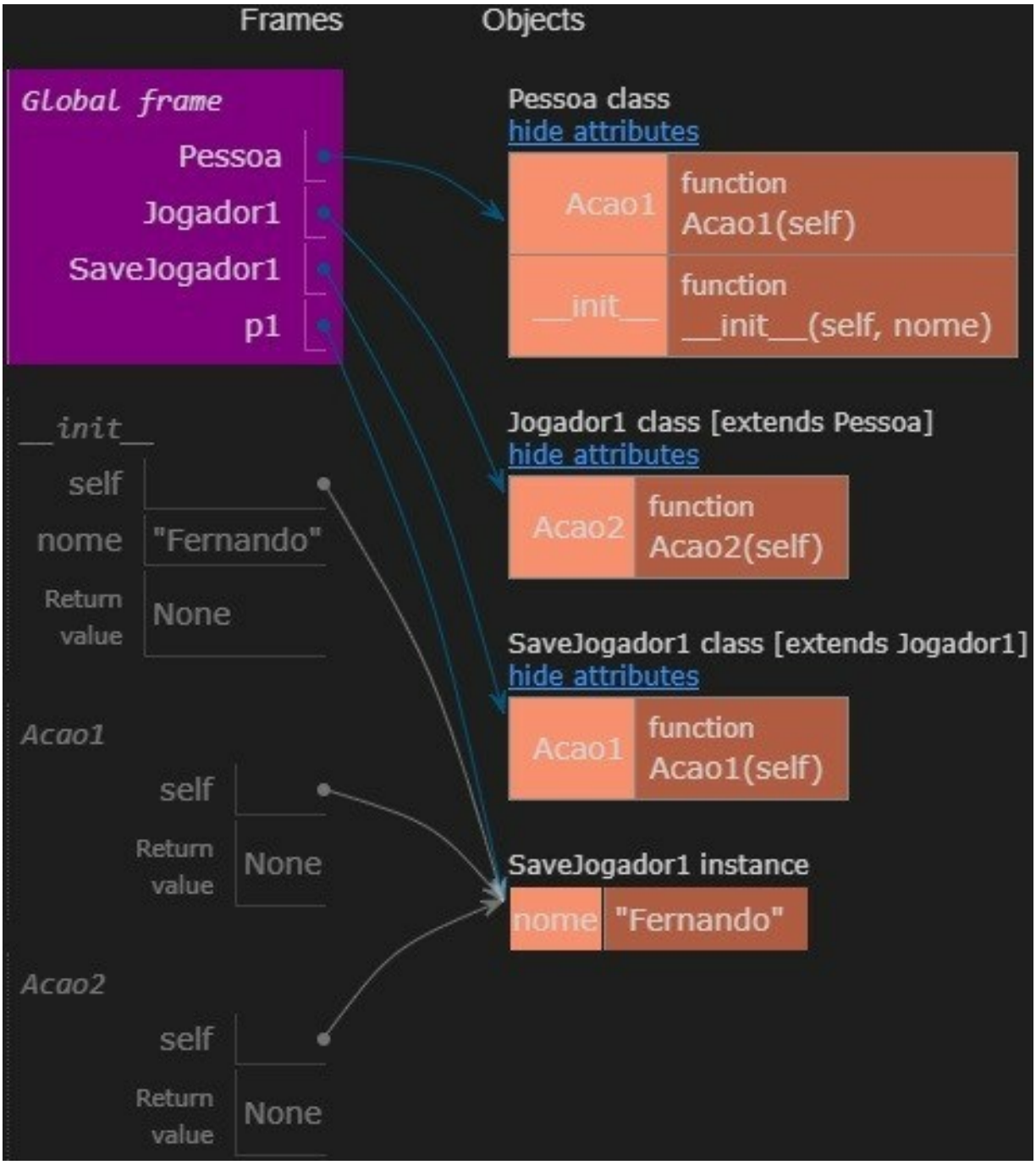
Em outras palavras, dada a hierarquia entre essas classes, o interpretador irá considerar pela sua leitura léxica a última alteração das mesmas, Acao1 presente em SaveJogador1 é a última modificação desse método de classe, logo, a função interna do mesmo irá ser executada no lugar de Acao1 de Pessoa, que será ignorada.

```
p1 = SaveJogador1('Fernando')
p1.Acao1()
p1.Acao2()

```

```
Fernando está acordado
Fernando está comendo
```

Finalmente temos como retorno o resultado apresentado na imagem acima.


Frames
Objects
Global frame
Pessoa
Jogador1
SaveJogador1
p1
__init__
self
nome
"Fernando"
Return value
None
Acao1
self
Return value
None
Acao2
self
Return value
None
Pessoa class
hide attributes
Acao1
function Acao1(self)
__init__
function __init__(self, nome)
Jogador1 class [extends Pessoa]
hide attributes
Acao2
function Acao2(self)
SaveJogador1 class [extends Jogador1]
hide attributes
Acao1
function Acao1(self)
SaveJogador1 instance
nome
"Fernando"

super( )

Outra funcionalidade que eventualmente pode ser utilizada é a de, justamente executar uma determinada ação da classe mãe e posteriormente a sobrescrever, isso é possível por meio da função reservada super( ).

Esta função em orientação a objetos faz com que seja respeitada a hierarquia de classe mãe / super classe em primeiro lugar e posteriormente as classes filhas / sub classes.

```
1  class SaveJogador1(Jogador1):
2      def Acao1(self):
3          super().Acao1()
4          print(f'{self.nome} está acordado')
5
```

Ainda no mesmo exemplo, neste caso, primeiro será executada a Acao1 de Pessoa e em seguida ela será sobrescrita por Acao1 de SaveJogador1.

```
Fernando está dormindo
Fernando está acordado
Fernando está comendo
```

Avançando com nossa linha de raciocínio, ao se trabalhar com heranças deve-se tomar muito cuidado com as hierarquias entre as classes dentro de sua ordem de leitura léxica.

Raciocine que o mesmo padrão usado no exemplo anterior vale para qualquer coisa, até mesmo um método construtor pode ser sobrescrito, logo, devemos tomar os devidos cuidados na hora de usar os mesmos.

Também é importante lembrar que por parte de sobreposições, quando se trata de heranças múltiplas, a ordem como as classes são passadas como parâmetro irá influenciar nos resultados.

```
1  class Pessoa:
2      def acao(self):
3          print('Inicializando o sistema')
4
5  class Acao1(Pessoa):
6      def acao(self):
7          print('Sistema pronto para uso')
8
9  class Acao2(Pessoa):
10     def acao(self):
11         print('Desligando o sistema')
12
13 class SaveJogador1(Acao1, Acao2):
14     pass
15
16 p1 = SaveJogador1()
17 p1.acao()
18
```

Apenas como exemplo, repare que foram criadas 4 classes, Pessoa, Acao1, Acao2 e SaveJogador1 respectivamente, dentro de si apenas existe um método de classe designado a exibir uma mensagem.

Para esse exemplo, o que importa é a forma como SaveJogador1 está herdando características de Acao1 e Acao2, como dito anteriormente, essa ordem ao qual as classes são passadas como parâmetro aqui influenciará os resultados da execução do código.

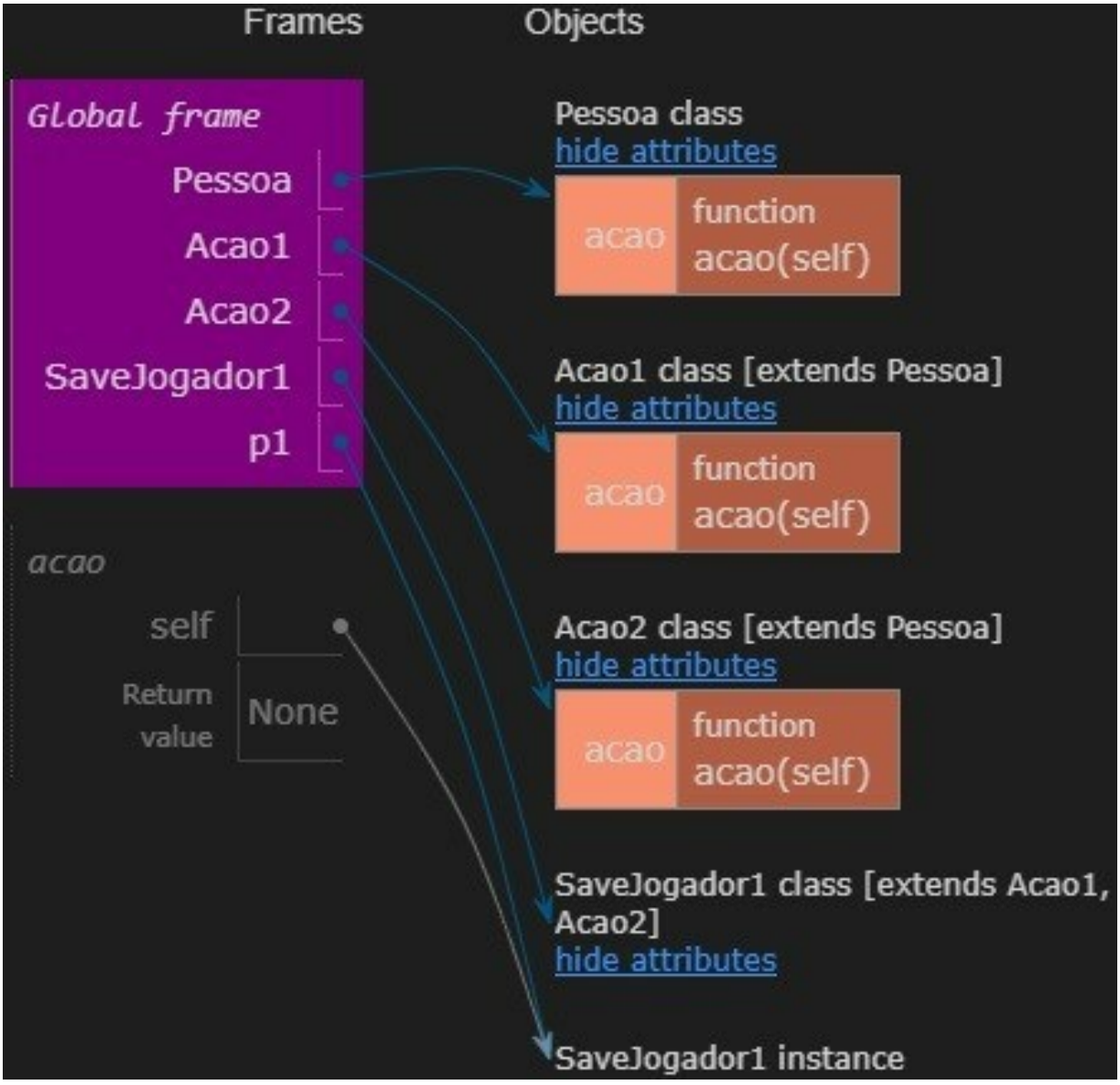
```
Sistema pronto para uso
```

Nesse exemplo, SaveJogador1(Acao1, Acao2), o retorno será: Sistema pronto para uso

```
13 class SaveJogador1(Acao2, Acao1):
14     pass
15
16 p1 = SaveJogador1()
17 p1.acao()
18
```

Exatamente a mesma estrutura anterior, porém com ordem inversa, SaveJogador1(Acao2, Acao1), o retorno será de acordo com esta justaposição forçada.

```
Desligando o sistema
```



Apenas fazendo um adendo, caso estes tópicos ainda gerem alguma dúvida para você. Uma forma de diferenciarmos a forma como tratamos as interações entre classes dentro do Python é a seguinte:

Associação: Se dá quando uma classe usa outro ou algum objeto de

outra.

Agregação: Se dá quando um ou mais objetos de classe é compartilhado, usado por duas ou mais classes (para que não seja necessário programar esse atributo para cada uma delas).

Composição: Se dá quando uma classe é dona de outra pela interligação de seus atributos e a forma sequencial como uns dependem dos outros.

Herança: Se dá quando estruturalmente um objeto é outro objeto, no sentido de que ele literalmente herda todas características e funcionalidades do outro, para que o código fique mais enxuto.

Classes Abstratas

Se você chegou até este tópico do livro certamente não terá nenhum problema para entender este modelo de classe, uma vez que já o utilizamos implicitamente em outros exemplos anteriores.

Mas o fato é que, como dito lá no início, para alguns autores uma classe seria um molde a ser utilizado para se guardar todo e qualquer tipo de dado associado a um objeto de forma que possa ser utilizado livremente de forma eficiente ao longo do código.

Sendo assim, quando criamos uma classe vazia, ou bastante básica, que serve como base estrutural para outras classes, já estamos trabalhando com uma classe abstrata.

O que faremos agora é definir manualmente este tipo de classe, de forma que sua forma estrutural irá forçar as classes subsequentes a criar suas especializações a partir dela. Pode parecer um tanto confuso, mas com o exemplo tudo ficará mais claro.

Como dito anteriormente, na verdade já trabalhamos com este tipo de classe, mas de forma implícita e assim o interpretador do Python não gerava nenhuma barreira quanto a execução de uma classe quando a mesma era abstrata.

Agora, usaremos para este exemplo, um módulo do Python que nos obrigará a abstrair manualmente as classes a partir de uma sintaxe própria.

Sendo assim, inicialmente precisamos importar os módulos ABC e abstractmethod da biblioteca abc.

```
1  from abc import ABC, abstractclassmethod
2
```

Realizadas as devidas importações, podemos prosseguir com o exemplo:

```
3  class Pessoa(ABC):
4      @abstractclassmethod
5      def logar(self):
6          pass
7
8  class Usuario(Pessoa):
9      def logar(self):
10         print('Usuario logado no sistema')
11
```

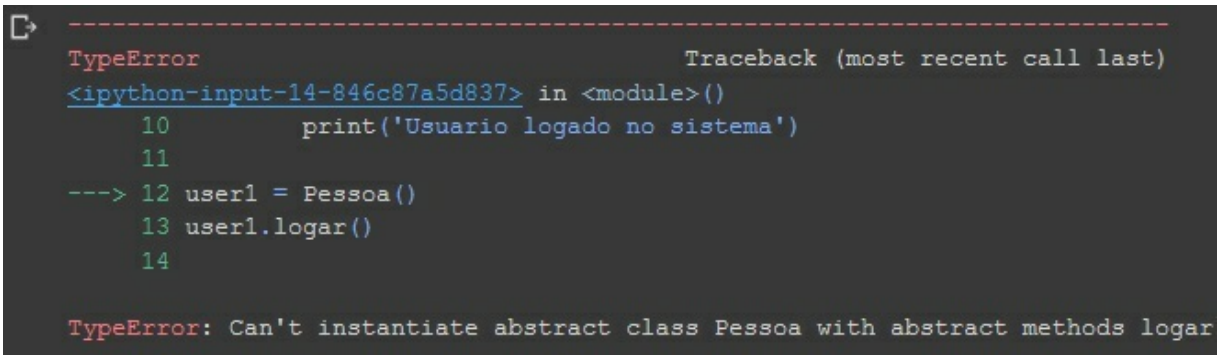
Inicialmente criamos uma classe Pessoa, que já herda ABC em sua declaração, dentro de si, note que é criado um decorador @abstractclassmethod, que por sua vez irá sinalizar ao interpretador que todos os blocos de código que vierem na sequência dessa classe, devem ser sobrescritas em suas respectivas classes filhas dessa hierarquia.

Em outras palavras, Pessoa no momento dessa declaração, possui um método chamado logar que obrigatoriamente deverá ser criado em uma nova classe herdeira, para que possa ser instanciada e realizar suas devidas funções.

Dando sequência, repare que em seguida criamos uma classe Usuario, que herda Pessoa, e dentro de si cria o método logar para sobrepor/sobrescrever o método logar de Pessoa. Este, por sua vez, exibe uma mensagem pré-definida por meio da função print( ).

```
12 user1 = Pessoa()
13 user1.logar()
14
```

A partir do momento que Pessoa é identificada como uma classe abstrata por nosso decorador, a mesma passa a ser literalmente apenas um molde. Seguindo com o exemplo, tentando fazer a associação que estamos acostumados a fazer, atribuindo essa classe a um objeto qualquer, ao tentar executar a mesma será gerado um erro de interpretação.



```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-14-846c87a5d837> in <module>()
     10         print('Usuario logado no sistema')
     11
---> 12 user1 = Pessoa()
     13 user1.logar()
     14

TypeError: Can't instantiate abstract class Pessoa with abstract methods logar
```

Em tradução livre: Não é possível inicializar a classe abstrata Pessoa com o método abstrato logar.



```
#user1 = Pessoa()
#user1.logar()

user1 = Usuario()
user1.logar()

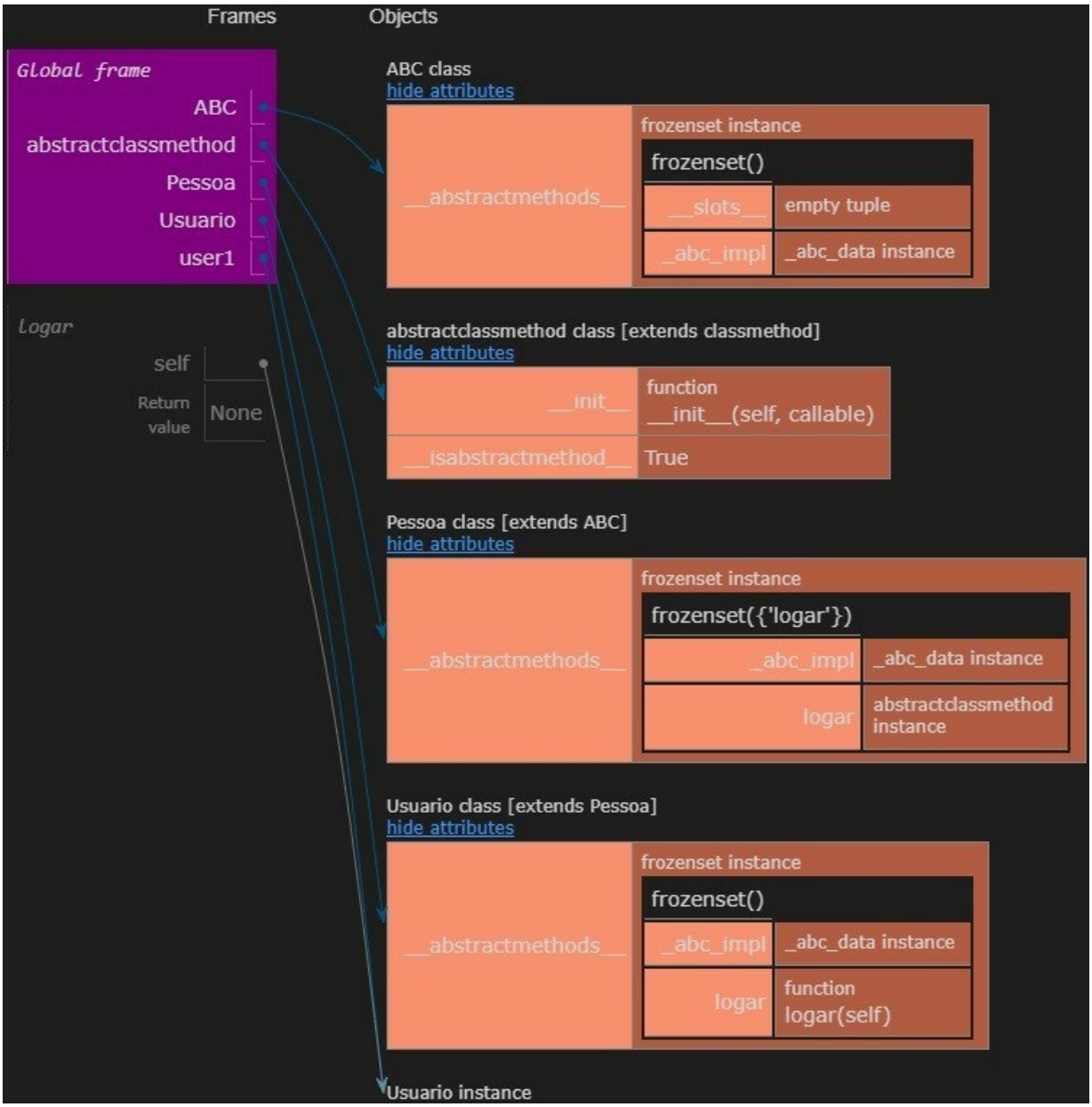
```

Porém instanciando Usuario, que herda Pessoa, ao tentar executar o método de classe logar que consta no corpo do mesmo, este funciona perfeitamente.



```
Usuario logado no sistema
```

Nesse caso o retorno será, como esperado: Usuario logado no sistema



Sendo assim, o que precisa ficar bem claro, inicialmente, é que uma das formas que temos de proteger uma classe que apenas nos serve de molde, para que essa seja herdada, mas não sobrescrita, é definindo a mesma manualmente como uma classe abstrata.

Polimorfismo

Avançando mais um pouco com nossos estudos, hora de abordar um princípio da programação orientada a objetos chamada polimorfismo.

Como o próprio nome já sugere, polimorfismo significa que algo possui muitas formas (ou ao menos mais que duas) em sua estrutura.

Raciocine que é perfeitamente possível termos classes derivadas de uma determinada classe mãe / super classe que possuem métodos iguais, porém comportamentos diferentes dependendo sua aplicação no código.

Alguns autores costumam chamar esta característica de classe de “assinatura”, quando se refere a mesma quantidade e tipos de parâmetros, porém com fins diferentes.

Último ponto a destacar antes de partirmos para o código é que, polimorfismo normalmente está fortemente ligado a classes abstratas, sendo assim, é de suma importância que a lógica deste tópico visto no capítulo anterior esteja bem clara e entendida, caso contrário poderá haver confusão na hora de identificar tais estruturas de código para criar suas devidas associações.

Entenda que uma classe abstrata, em polimorfismo, obrigatoriamente será um molde para criação das demais classes onde será obrigatório realizar a sobreposição do(s) método(s) de classe que a classe mãe possuir.

```
1  from abc import ABC, abstractclassmethod
2
3  class Pessoa(ABC):
4      @abstractclassmethod
5      def logar(self, chaveseguranca):
6          pass
7
8  class Usuario(Pessoa):
9      def logar(self, chaveseguranca):
10         print('Usuario logado no sistema')
11
12 class Bot(Pessoa):
13     def logar(self, chaveseguranca):
14         print('Sistema rodando em segundo plano')
15
```

Seguindo com um exemplo muito próximo ao anterior, apenas para melhor identificação, note que inicialmente são feitas as importações dos módulos ABC e abstractclassmethod da biblioteca abc.

Em seguida é criada uma classe de nome Pessoa que herda ABC, dentro de si existe um decorador e um método de classe que a caracteriza como uma classe abstrata.

Em outras palavras, toda classe que herdar a estrutura de Pessoa terá de redefinir o método logar, fornecendo uma chavedeseguranca que aqui não está associada a nada, mas na prática seria parte de um sistema de login.

Na sequência é criada a classe Usuário que herda Pessoa, e pela sua estrutura, repare que ela é polimórfica, possuindo a mesma assinatura de objetos instanciados ao método logar, porém há uma função personalizada que está programada para exibir uma mensagem.

O mesmo é feito com uma terceira classe chamada Bot que herda Pessoa, que por sua vez herda ABC, tudo sob a mesma “assinatura”.

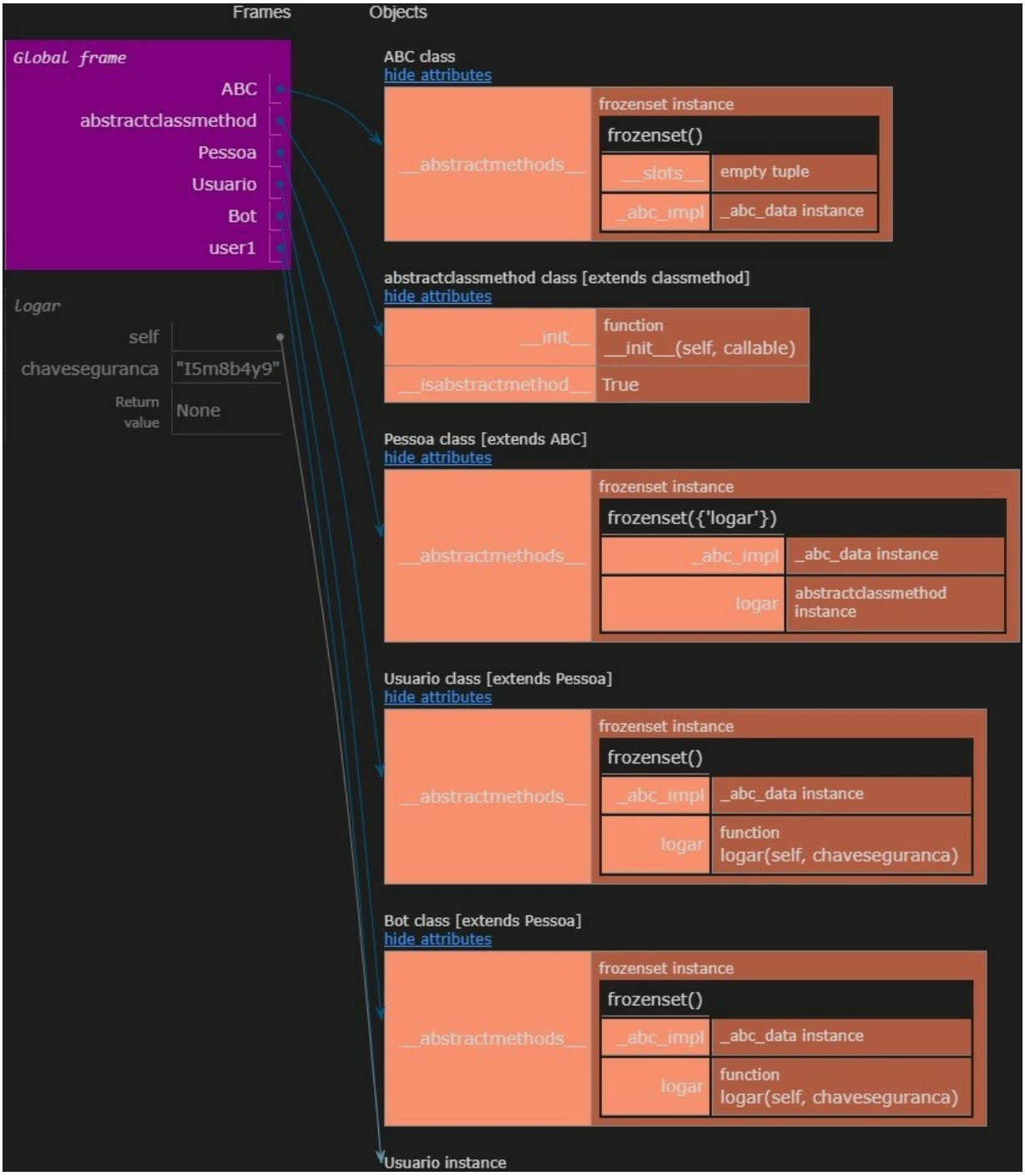
```
16 user1 = Usuario()
17 user1.logar('I5m8b4y9')
18
```

Instanciando Usuario a um objeto qualquer e aplicando sobre si

parâmetros (neste caso, fornecendo uma senha para chaveseguranca), ocorrerá a execução da respectiva função.

```
Usuario logado no sistema
```

Neste caso o retorno será: Usuario logado no sistema



Sobrecarga de Operadores

Quando damos nossos primeiros passos no aprendizado do Python, em sua forma básica, programando de forma estruturada, um dos primeiros tópicos que nos é apresentado são os operadores.

Estes por sua vez são símbolos reservados ao sistema que servem para fazer operações lógicas ou aritméticas, de forma que não precisamos programar do zero tais ações, pela sintaxe, basta usar o operador com os tipos de dados aos quais queremos a interação e o interpretador fará a leitura dos mesmos normalmente.

Em outras palavras, se declaramos duas variáveis com números do tipo inteiro, podemos por exemplo, via operador de soma + realizar a soma destes valores, sem a necessidade de programar do zero o que seria uma função que soma dois valores.

Se tratando da programação orientada a objetos, o que ocorre é que quando criamos uma classe com variáveis/objetos dentro dela, estes dados passam a ser um tipo de dado novo, independente, apenas reconhecido pelo interpretador como um objeto.

Para ficar mais claro, raciocine que uma variável normal, com valor atribuído de 4 por exemplo, é automaticamente interpretado por nosso interpretador como int (número inteiro, sem casas decimais), porém a mesma variável, exatamente igual, mas declarada dentro de uma classe, passa a ser apenas um objeto geral para o interpretador.

Números inteiros podem ser somados, subtraídos, multiplicados, elevados a uma determinada potência, etc... um objeto dentro de uma classe não possui essas funcionalidades a não ser que manualmente realizemos a chamada sobrecarga de operadores. Por exemplo:

```
1  class Caixa:
2      def __init__(self, largura, altura):
3          self.largura = largura
4          self.altura = altura
5
6  caixa1 = Caixa(10,10)
7  caixa2 = Caixa(10,20)
8
```

Aqui propositalmente simulando este erro lógico, inicialmente criamos uma classe de nome Caixa, dentro dela um método construtor que recebe um valor para largura e um para altura, atribuindo esses valores a objetos internos.

No corpo de nosso código, criando variáveis que instanciam nossa classe Caixa e atribuem valores para largura e altura, seria normal raciocinar que seria possível, por exemplo, realizar a soma destes valores.

```
 9  print(caixa1 + caixa2)
10
```

Porém o que ocorre ao tentar realizar uma simples operação de soma entre tais valores o retorno que temos é um traceback.

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent c
<ipython-input-17-b5b182d8e6c6> in <module>()
      7 caixa2 = Caixa(10,20)
      8
----> 9 print(caixa1 + caixa2)

TypeError: unsupported operand type(s) for +: 'Caixa' and 'Caixa'
```

Em tradução livre: Erro de Tipo: tipo de operador não suportado para +: Caixa e Caixa.

Repare que o erro em si é que o operador de soma não foi reconhecido como tal, pois esse símbolo de + está fora de contexto uma vez que estamos trabalhando com orientação a objetos.

Sendo assim, o que teremos de fazer é simplesmente buscar os operadores que sejam reconhecidos e processados neste tipo de programação.

Segue uma lista com os operadores mais utilizados, assim como o seu método correspondente para uso em programação orientada a objetos.

Operador	Método	Operação
+	__add__	Adição
-	__sub__	Subtração
*	__mul__	Multiplicação
/	__div__	Divisão
//	__floordiv__	Divisão inteira
%	__mod__	Módulo
**	__pow__	Potência
<	__lt__	Menor que
>	__gt__	Maior que
<=	__le__	Menor ou igual a
>=	__ge__	Maior ou igual a
==	__eq__	Igual a
!=	__ne__	Diferente de

Dando continuidade a nosso entendimento, uma vez que descobrimos que para cada tipo de operador lógico/aritmético temos um método de classe correspondente, tudo o que teremos de fazer é de fato criar este método dentro de nossa classe, para forçar o interpretador a reconhecer que, nesse exemplo, é possível realizar a soma dos dados/valores atribuídos aos objetos ali instanciados.

```
1  class Caixa:
2      def __init__(self, largura, altura):
3          self.largura = largura
4          self.altura = altura
5
6      def __add__(self, other):
7          pass
8
```

Retornando ao código, note que simplesmente é criado um método

de classe de nome __add__ que recebe como atributos um valor para si e um valor a ser somado neste caso. __add__ é uma palavra reservada ao sistema, logo, o interpretador sabe sua função e a executará normalmente dentro desse bloco de código.

```
1  class Caixa:
2      def __init__(self, largura, altura):
3          self.largura = largura
4          self.altura = altura
5
6      def __add__(self, other):
7          largura1 = self.largura + other.largura
8          altura1 = self.altura + other.altura
9          return Caixa(largura1, altura1)
10
11     def __repr__(self):
12         return f"<class 'Caixa({self.largura}, {self.altura})'>"
13
```

Dessa forma, podemos criar normalmente uma função de soma personalizada, note que no método __add__ foi criada uma variável de nome largura1 que guardará o valor da soma entre as larguras dos objetos.

Da mesma forma, altura1 receberá o valor da soma das alturas, tudo isto retornando para a classe tais valores.

Na sequência, apenas para facilitar a visualização, foi criada um método de classe __repr__ (também reservado ao sistema) que irá retornar os valores das somas explicitamente ao usuário (lembre-se de que como estamos trabalhando dentro de uma classe, por padrão esta soma seria uma mecanismo interno, se você tentar printar esses valores sem esta última função, você terá como retorno apenas o objeto, e não seu valor).

```
14  caixa1 = Caixa(10,10)
15  caixa2 = Caixa(10,20)
16  caixa3 = caixa1 + caixa2
17
18
19  print(caixa3)
20
```

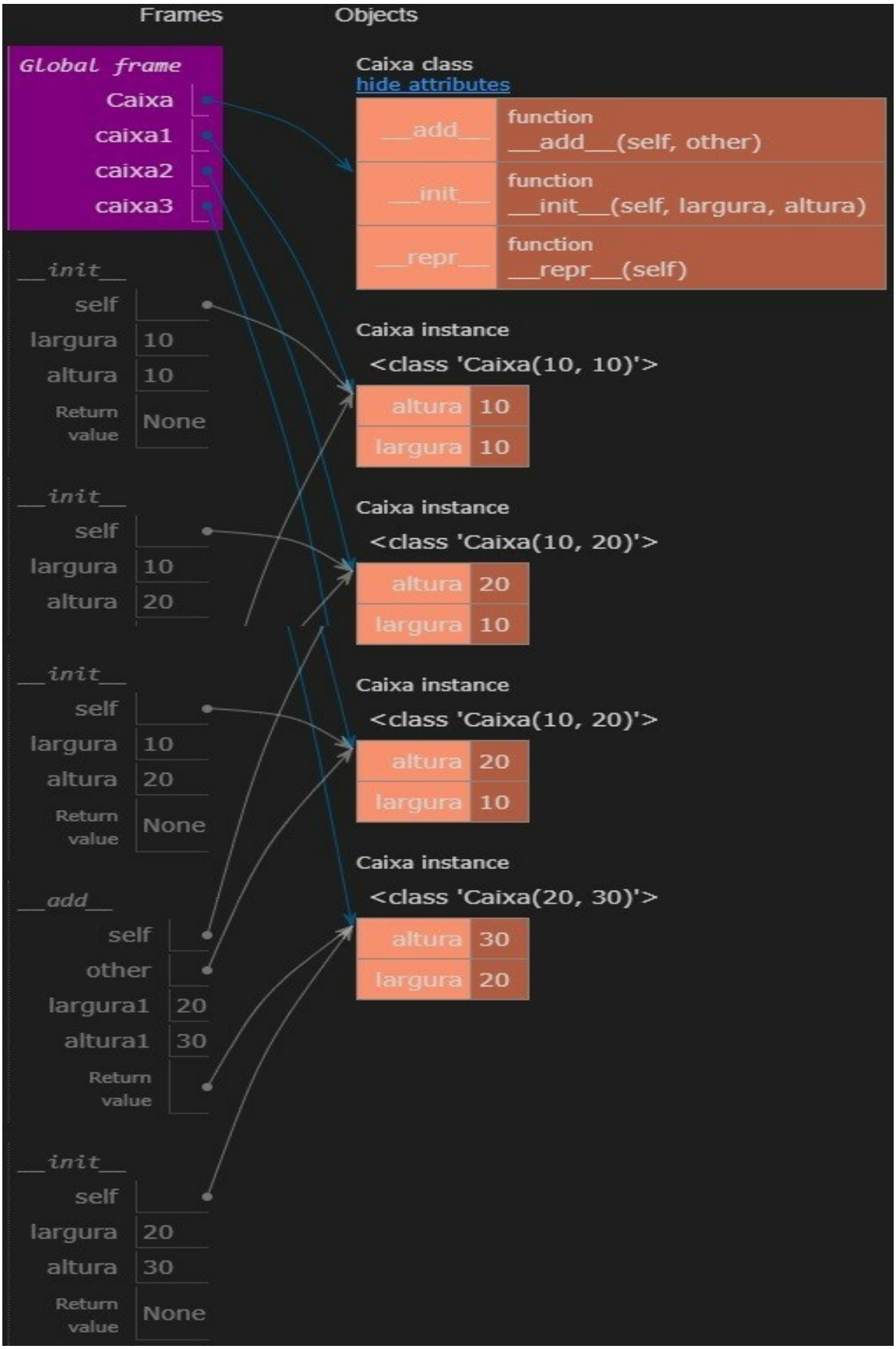
Por fim, agora instanciando a classe, atribuindo seus respectivos

valores, finalmente é possível realizar normalmente a operação de soma entre as variáveis que instanciam essa classe.

Nesse exemplo, criando uma variável de nome caixa3 que recebe como atributo a soma dos valores de largura e altura de caixa1 e caixa2.

```
<class 'Caixa(20, 30)'>
```

O retorno, como esperado, será: <class 'Caixa(20, 30)'>.


Frames
Objects
Global frame
Caixa
caixa1
caixa2
caixa3
Caixa class
hide attributes
__add__
function
__add__(self, other)
__init__
function
__init__(self, largura, altura)
__repr__
function
__repr__(self)
__init__
self
largura 10
altura 10
Return value None
Caixa instance
<class 'Caixa(10, 10)'>
altura 10
largura 10
__init__
self
largura 10
altura 20
Caixa instance
<class 'Caixa(10, 20)'>
altura 20
largura 10
__init__
self
largura 10
altura 20
Return value None
Caixa instance
<class 'Caixa(10, 20)'>
altura 20
largura 10
__add__
self
other
largura1 20
altura1 30
Return value
Caixa instance
<class 'Caixa(20, 30)'>
altura 30
largura 20
__init__
self
largura 20
altura 30
Return value None

Encerrando, apenas tenha em mente que, de acordo com a funcionalidade que quer usar em seu código, deve procurar o método equivalente e o declarar dentro da classe como um método de classe qualquer, para “substituir” o operador lógico/aritmético de forma a ser lido e interpretador por nosso interpretador.

Filas (Queues) e Nodes

Uma prática bastante comum, embora demande a criação de um grande bloco de código de funções validadoras, é gerenciar filas via OO, fazendo uso de Nodes (nós que são dinâmicos tanto para inserção de dados quanto para atualização dos mesmos).

Raciocine que via OO podemos criar toda uma estrutura base que terá performance superior se comparado a um modelo equivalente estruturado, ou em outras palavras, melhor gerenciamento de listas do que pela forma básica por meio de seus índices, já que neste tipo de arquitetura de dados, temos elementos de uma lista como objetos dinamicamente alocados, permitindo a manipulação dos mesmos de forma semi automatizada assim como suporte a grandes volumes de dados.

Tenha em mente que em uma lista estruturada, a cada atualização de elementos em fila, é necessário internamente mover o elemento e atualizar seu índice um a um, enquanto nos exemplos que você verá usando de OO, este processo é internamente dinâmico, de modo que todos elementos assim como seus índices (se houver) estarão dentro de um objeto.

Para isso, você notará que os blocos exibidos a seguir foram criados de modo a criar uma cadeia de dependências, ou seja, espaços alocados esperados para serem alimentados com dados, e ao mesmo tempo, tais estruturas realizarão uma série de validações para que os elementos da fila tenham o comportamento esperado, atualizando sua posição na fila conforme a fila anda.

Sendo assim, teremos “nós” em uma fila que respeitam a lógica first in first out, e que podem trabalhar com qualquer elemento, sem restrições de tipo ou número.

Como mencionado anteriormente, o que justifica a criação desta estrutura de gerenciamento de filas é uma questão de performance em função da maneira como os dados serão tratados via OO.

Apesar da estrutura ser um tanto quanto grande, procure se familiarizar com a mesma, principalmente se seu objetivo é posteriormente

aprender sobre redes neurais artificiais.

Este sistema de dependências e validações é muito comum em arquiteturas como as de redes neurais artificiais intuitivas, onde uma IA realiza processamento em tempo real se retroalimentando, atualizando e validando seus estados constantemente.

```
1  from typing import Any
2  EMPTY_NODE_VALUE = '__EMPTY_NODE_VALUE__'
3
4  class Erro(Exception):
5      ...
6
```

Como de costume, em casos onde se faz o uso de bibliotecas, módulos e pacotes, logo no início do nosso código realizamos as devidas importações.

Nesse caso, da biblioteca typing importamos Any, que por sua vez é uma estrutura de dado que pode ser inserida em qualquer contexto, guardando o lugar de algum tipo de dado que a venha substituir.

Em seguida é criada uma constante de nome EMPTY_NODE_VALUE que recebe como atributo uma string homônima. Por mais estranho que isso pareça uma vez que em Python não temos constantes, todos tipos de dados tanto estruturados quanto orientados a objeto são sempre dinâmicos.

Aqui, apenas por convenção, criamos uma variável equivalente a uma constante, com a simples função de atribuir algum dado onde possa não existir, evitando erros de interpretação.

Também é criada uma classe Erro, que levanta uma exceção via parâmetro Exception, esta estrutura, que também será usada ao longo de nosso código, da mesma forma que a constante EMPTY_NODE_VALUE, evitando que o programa pare de funcionar em função de um erro de interpretação.

```
 7  class Node:
 8    def __init__(self, value: Any) -> None:
 9      self.value = value
10      self.next = Node
11
12    def __repr__(self) -> str:
13      return f'{self.value}'
14
15    def __bool__(self) -> bool:
16      return bool(self.value != EMPTY_NODE_VALUE)
17
```

Na sequência criamos a nossa classe Node, dentro da mesma temos um método construtor __init__( ) que recebe como atributo um valor qualquer via value: Any e retorna um None.

Por hora raciocine que um dos fatores que devem ser considerados dentro desta lógica é que existirá um momento inicial onde a fila pode estar vazia, assim como um momento final onde a fila novamente pode ficar vazia, em ambos os casos, os espaços alocados por value podem (Any) ou não (None) tem algum dado atribuído a si.

Como esperado, são criados os objetos value que recebe um valor qualquer, da mesma forma é criado o objeto next que por sua vez recebe Node.

De forma opcional podem ser criados um método de classe chamado __repr__( ) que retorna value em forma de string, simplesmente para que se possa realizar a leitura do dado/valor atribuído a value.

Também é criado um método de classe __bool__( ) que irá retornar um valor True quando o valor de value for diferente de EMPTY_NODE_VALUE, o que em outras palavras nos diz que value não está mais em seu estado inicial vazio, possuindo agora dados atribuídos a si.

```
18  class Queue:
19    def __init__(self) -> None:
20      self.first: Node = Node(EMPTY_NODE_VALUE)
21      self.last: Node = Node(EMPTY_NODE_VALUE)
22      self._count = 0
23
```

Na sequência é criada a classe Queue que em seu método construtor retorna None, dentro deste método são criados os objetos first e last, ambos recebendo como atributos Node, que chama a classe Node parametrizando a mesma com EMPTY_NODE_VALUE. Em outras palavras, aqui definimos uma posição inicial e uma final da fila, inicialmente vazias.

Por fim, é criado um contador inicialmente zerado, lembrando que aqui, apenas por controle, toda vez que for adicionado um elemento a fila, _count será incrementado, da mesma forma, toda vez que um elemento for removido da fila, _count será decrementado.

```
24    def enqueue(self, node_value: Any) -> None:
25        novo_node = Node(node_value)
26
27        if not self.first:
28            self.first = novo_node
29        if not self.last:
30            self.last = novo_node
31        else:
32            self.last.next = novo_node
33            self.last = novo_node
34        self._count += 1
35
```

Ainda dentro de Queue, criamos o método de classe enqueue, função que ficará responsável por adicionar elementos na fila e gerenciar a mesma na sequência lógica esperada.

Inicialmente dentro de seu corpo é criado um objeto de nome novo_node que instancia Node repassando para o mesmo o valor que estiver atribuído a node_value.

Em seguida é criada uma estrutura condicional onde inicialmente se first ainda não tiver sido construído e alocado na memória, é criado o mesmo com os dados iniciais de novo_node. Da mesma forma é criado last com valor inicial oriundo de novo_node.

O importante a se entender nesta etapa é que pensando em evitar erros, esses objetos são criados mesmo que não tenham dados inicialmente, sendo apenas espaços reservados para que sejam alimentados posteriormente.

Caso as condições impostas anteriormente não sejam alcançadas, last.next recebe os dados de novo_node, atualizando last e incrementando o contador em 1.

Raciocine que aqui nesse processo estamos forçando para que o elemento que entrar na fila ocupe a última posição da fila, claro que quando é adicionado o primeiro elemento ele ocupa a primeira posição da fila, mas o importante é que novos elementos que sejam adicionados à fila entrem na ordem correta, ao final da fila.

```
36    def pop(self) -> Node:
37        if not self.first:
38            raise Erro('Sem elementos na fila')
39        first = self.first
40
41        if hasattr(self.first, 'next'):
42            self.first = self.first.next
43        else:
44            self.first = Node(EMPTY_NODE_VALUE)
45        self._count =+ 1
46        return first
47
```

Criada a função que permitirá a adição de novos elementos na fila, hora de criar a função de objetivo oposto, uma vez que uma das funcionalidades que não podem faltar em um programa desses é justamente a de poder remover elementos da lista.

Sendo assim, criamos o método de classe pop( ) que retorna um Node. Em seu corpo é criada uma estrutura condicional onde caso não exista nenhum dado em first, e o usuário ainda assim tentar remover algum elemento desta posição, é levantado um erro dizendo ao usuário que não existem elementos a serem removidos da lista.

Da mesma forma é criado o método hasattr( ) palavra reservada do sistema que nada mais é do que uma simples função validadora, verificando se de fato existem ou não dados em first.

Note que neste caso, first é atualizado com o dado que houver em first.next, uma vez que quando a fila anda, o segundo elemento passa a ser o

primeiro, o terceiro o segundo, e assim para todos elementos sucessivamente.

Aqui também ocorre a validação onde caso não existam elementos para ocupar a primeira posição da fila, será adicionado um EMPTY_NODE_VALUE para que não ocorram erros de interpretação.

Por fim é retornado o valor atualizado de first.

```
48    def peek(self) -> Node:
49      return self.first
50
51    def __len__(self) -> int:
52      return - self._count
53
54    def __bool__(self) -> bool:
55      return bool(self._count)
56
57    def __iter__(self) -> Queue:
58      return self
59
60    def __next__(self) -> Any:
61      try:
62        next_value = self.pop()
63        return next_value
64      except Erro:
65        raise StopIteration
66
```

Dando prosseguimento, é possível criar de forma totalmente opcional alguns métodos de classe validadores ou verificadores. Por exemplo peek( ) pode simplesmente pegar o dado/valor do primeiro elemento e exibir para o usuário; __len__( ) pode ler o último valor de _count e retornar o tamanho da fila; __bool__( ) pode retornar True caso o valor de _count seja 1 ou maior que 1, significando que existe ao menos um elemento na fila; __iter__( ) que retorna toda a lista; __next__( ) que retorna qualquer coisa, validando se não existem mais elementos a serem removidos da lista.

```
67  fila1 = Queue()
68  fila1.enqueue('Maria')
69  fila1.enqueue('Carlos')
70  fila1.pop()
71
```

Por fim, é perfeitamente possível criar um objeto qualquer que instancia nossa classe Queue, e a partir disso se pode adicionar elementos a lista via função enqueue( ) assim como remover elementos da mesma via função pop( ).

Como dito no início desse capítulo, essa é uma estrutura grande e um pouco confusa, porém de performance e usabilidade muito mais eficiente do que se trabalhar com o gerenciamento de filas de forma estruturada.

Gerenciamento de filas via Deque

Uma vez entendida a lógica de gerenciamento de filas a partir do exemplo anterior, hora de conhecer um método alternativo baseado em uma biblioteca nativa do Python chamada Deque.

Raciocine que em programação você dificilmente irá reinventar a roda, mas existe uma série de possibilidades de melhorar a mesma.

Conferindo repositórios de bibliotecas é possível ver que existem centenas de milhares de bibliotecas, módulos e pacotes criados pela comunidade a fim de otimizar ou implementar funcionalidades no Python.

```
1  from typing import Deque, Any
2  from collections import deque
3
```

Como sempre, todo processo se inicia pela importação das bibliotecas, módulos e pacotes que serão utilizados ao longo de nosso código. Nesse caso, da biblioteca typing importamos Deque e Any. Da mesma forma da biblioteca collections importamos deque.

```
 4  fila2: Deque[Any] = deque()
 5
 6  fila2.append('Ana')
 7  fila2.append('Carlos')
 8  fila2.append('Fernando')
 9  fila2.append('Maria')
10
```

Em seguida criamos uma variável de nome fila2 que por sua vez instancia Deque com um objeto qualquer em sua posição inicial e recebe como atributo a função deque( ), sem parâmetros mesmo.

Na sequência já podemos, como de costume, manipular dados da mesma forma que costumamos fazer em listas básicas, por meio da função append( ) podemos adicionar elementos a nossa fila.

```
11  for pessoas in fila2:
12    print(pessoas)
13
```

Por meio do laço for podemos fazer a varredura dessa lista e verificar quantos e quais elementos compõem essa fila.

```
Ana
Carlos
Fernando
Maria
```

Como esperado, os elementos foram adicionados sequencialmente conforme ordem de adição.

```
14  fila2.popleft()
15
```

Para removermos elementos de nossa fila, via deque( ) usamos da função popleft( ) uma vez que queremos remover o primeiro elemento da fila, sendo o segundo assumindo a posição de primeiro, o terceiro a do segundo, respectivamente independente da quantidade de elementos na fila.

```
16  for pessoas in fila2:
17    print(pessoas)
18
```

Da mesma forma, como no bloco anterior, via laço for podemos percorrer a fila após a atualização da mesma.

```
Carlos
Fernando
Maria
```

E como esperado, a nova leitura da fila nos mostra que Carlos agora ocupa a primeira posição da fila e os demais nas suas respectivas posições.

Apenas uma observação, para concluirmos nossa linha de raciocínio, filas geradas via Deque possuem índice, você pode usar de qualquer função ao qual está habituado a usar em uma lista comum, porém você deve tomar o cuidado de não tentar remover, por exemplo, o elemento 0 via índice, nesse caso irá gerar um erro de interpretação, haja visto que essa biblioteca não possui internamente todos validadores que criamos no

exemplo anterior manualmente.

PEPS

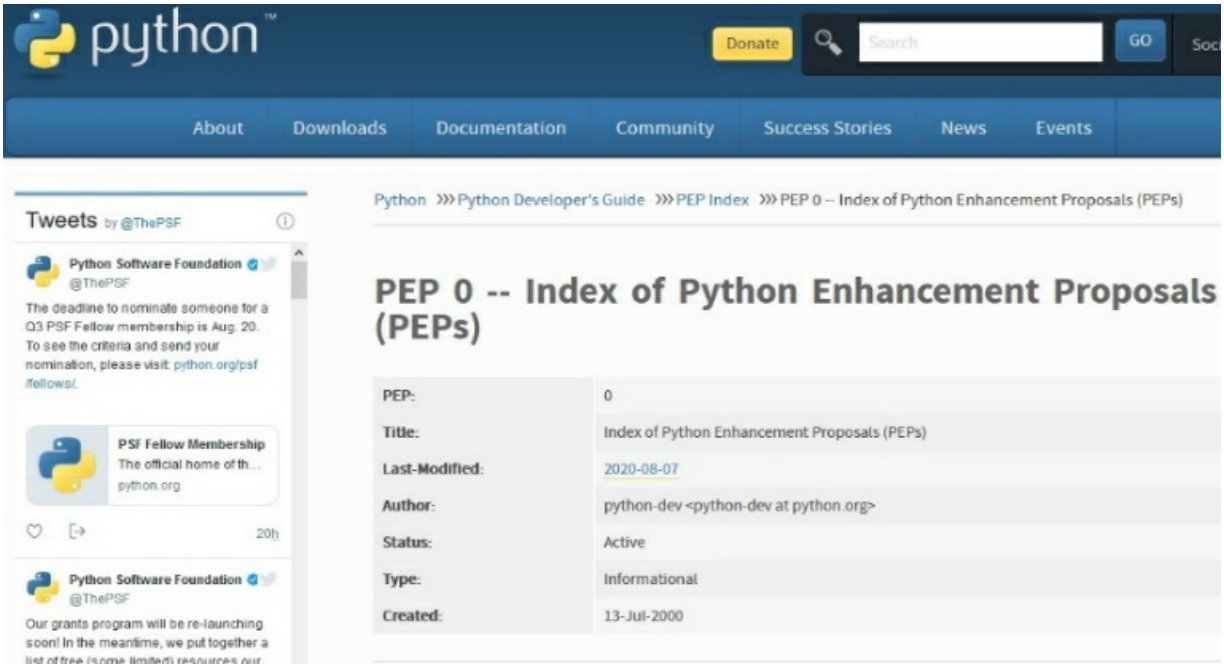
Encerrados os tópicos deste livro você deve estar se perguntando quais são os próximos passos.

Pois bem, recomendo fortemente que você explore mais a fundo a própria documentação da linguagem Python.

Como dito logo no início desse livro, existe uma infinidade de possibilidades fazendo uso do Python, e na condição de estudantes, grande parte das vezes nem temos noção de tantas ferramentas que essa linguagem pode oferecer.

Outra forma bastante interessante de se aprofundar em tópicos específicos é consultando a documentação PEPS do Python.

Contextualizando, PEPS em tradução livre seria algo como Índice de Propostas e Melhorias do Python. Nesta documentação em específico você verá tudo o que já foi implementado em Python ao longo de suas versões, assim como ferramentas em processo de implementação e projetos de ferramentas a serem implementadas em versões futuras.



Tudo relacionado ao PEPS do Python está disponível em https://www.python.org/dev/peps/

REPOSITÓROS

Avançando um pouco mais em Python, proporcional a complexidade de um projeto está a suscetibilidade a erros, o que é perfeitamente normal. O importante é sempre saber onde buscar informações relevantes ao seu problema, e nesse contexto Python está muito bem servido graças a sua comunidade.

A fonte recomendada por todos autores, e também por mim, é o famoso Stackoverflow (acessível em: https://stackoverflow.com/), aqui temos um ambiente que conta com fóruns ativos onde é possível buscar informações diretamente sobre os erros mais comuns entre os usuários da comunidade.

O Stackoverflow se tornou tão importante em meio a comunidade de programadores que boa parte das IDEs ao gerar um traceback já oferece a opção de consultar a possível solução do problema com um click.

De forma manual, certamente é uma excelente ferramenta quando se tem dúvidas sobre certos pontos, podendo ver de forma prática o código original com as devidas soluções fornecidas pelos usuários.

Outra possibilidade é quando estamos trabalhando sobre uma biblioteca específica e tendo algumas dificuldades com a mesma. Por meio do site Pypi (acessível em: https://pypi.org/) podemos não só baixar as bibliotecas/módulos como ter acesso a documentação oficial das mesmas.

Por fim, outra alternativa interessante é realizar a consulta via GitHub (acessível em: https://github.com/), aqui temos uma vasta gama de repositórios de código, onde boa parte deles, senão tdos, estão disponibilizados para livre uso, assim como um bom percentual são de códigos comentados pelo próprio criador.

PYTHON + NUMPY



Quando estamos usando da linguagem Python para criar nossos códigos, é normal que usemos de uma série de recursos internos da própria linguagem, uma vez que como diz o jargão, Python vem com baterias inclusas, o que em outras palavras pode significar que Python mesmo em seu estado mais básico já nos oferece uma enorme gama de funcionalidades prontas para implementação.

No âmbito de nichos específicos como, por exemplo, computação científica, é bastante comum o uso tanto das ferramentas nativas da linguagem Python como de bibliotecas desenvolvidas pela comunidade para propósitos bastante específicos.

Com uma rápida pesquisa no site https://pypi.org é possível ver que existem, literalmente, milhares de bibliotecas, módulos e pacotes desenvolvidos pela própria comunidade, com licença de uso e distribuição livre, a fim de adicionar novas funcionalidades ou aprimorar funcionalidades já existentes no núcleo da linguagem Python, tornando esta linguagem ainda melhor.

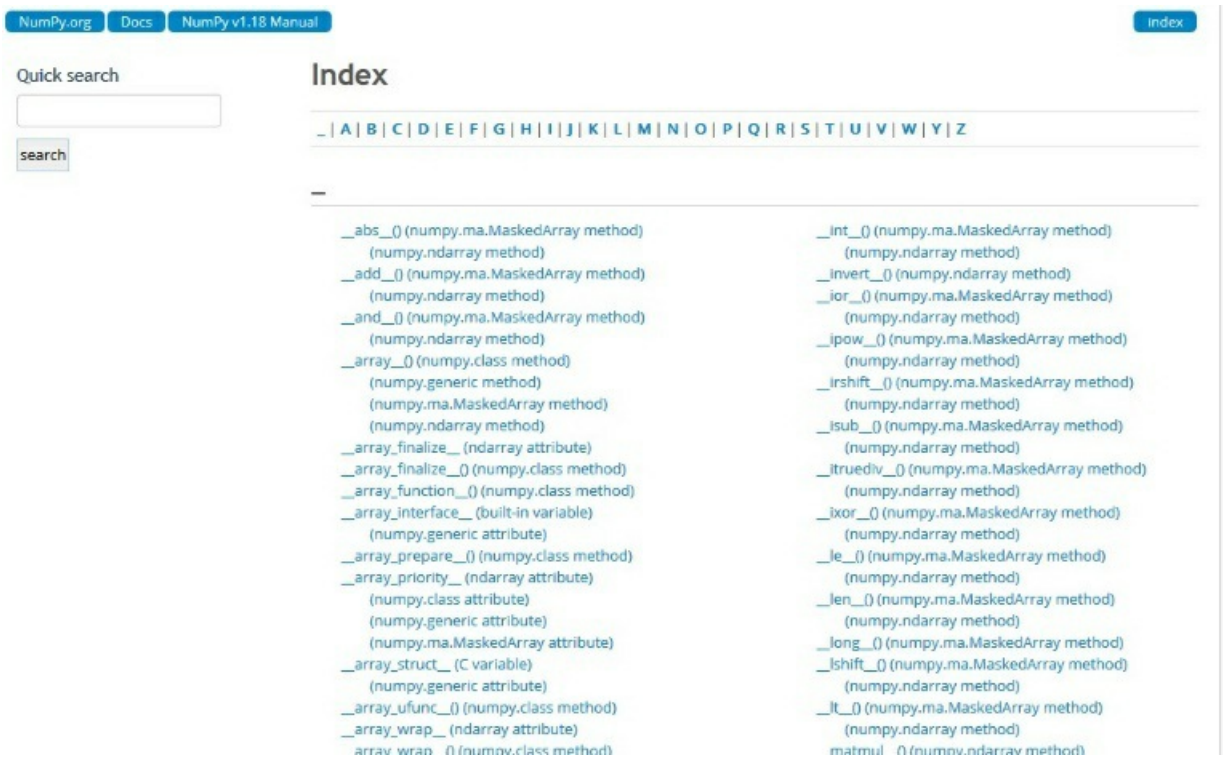
Uma das bibliotecas mais utilizadas para o tratamento de dados em formato vetorial e matricial, devida a sua facilidade de implementação e uso, é a biblioteca Numpy. Esta por sua vez oferece uma grande variedade de funções aritméticas de fácil aplicação, de modo que para certos tipos de operações as funções

da biblioteca Numpy são muito mais intuitivas e eficientes do que as funções nativas de mesmo propósito presentes nas built-ins do Python para as chamadas arrays.

Sobre a biblioteca Numpy

Como mencionado anteriormente, a biblioteca Numpy, uma das melhores, se não a melhor, quando falamos de bibliotecas dedicadas a operações aritméticas sobre dados em forma de matriz, e isso se estende por uma enorme gama de aplicações de estruturas de dados matemáticos aplicados.

De acordo com a própria apresentação da biblioteca em seu site oficial numpy.org, a biblioteca Numpy oferece uma estrutura de dados pela qual é possível realizar operações em áreas como análise matemática e estatística, processamento de imagens, bioinformática, computação simbólica, processamento em redes neurais artificiais além de uma grande integração com outras bibliotecas e ferramentas científicas externas.



Tudo isso se dá por meio de uma enorme variedade de funções pré-definidas para cada tipo de aplicação, explorando parte da documentação é possível ver que temos literalmente algumas centenas delas prontas para uso, bastando chamar a função específica e a parametrizar como é esperado.

De fato, é fácil compreender como tal biblioteca, dada sua estrutura e características, aliada a facilidade de uso da linguagem Python, se tornou uma referência para computação científica.

Neste pequeno compendio estaremos fazendo uma rápida leitura sobre as principais aplicações da biblioteca Numpy, desde sua estrutura de dados mais básica, funções mais comumente usadas até um exemplo de aplicação de arrays do tipo Numpy como estrutura de dados de uma rede neural artificial.

Toda documentação se encontra disponível no site oficial numpy.org até o momento sem tradução para português brasileiro.

Site oficial:

https://numpy.org/

GitHub oficial:

https://github.com/numpy/numpy

Instalação e Importação das Dependências

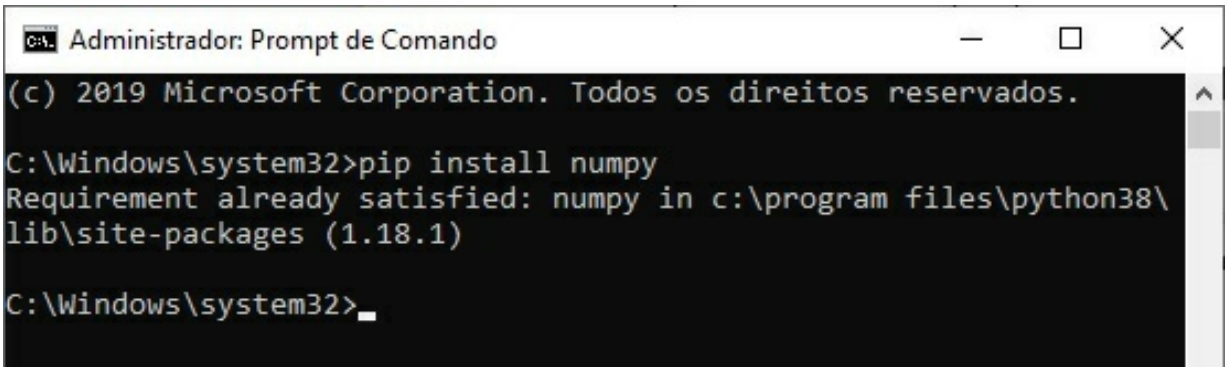
Para os exemplos mostrados nos capítulos subsequentes, estaremos usando do Google Colab, IDE online muito semelhante ao Jupyter Notebook, porém rodando em nuvem. No exemplo final o código exibido está rodando na IDE Spyder, ambas IDEs bastante familiares para quem trabalha com redes neurais artificiais, aqui usadas apenas pela comodidade de podermos executar blocos de código isoladamente.

Para podemos realizar a instalação e atualização da biblioteca Numpy por meio do gerenciador de pacotes **pip** no Colab, de forma muito parecida como quando instalamos qualquer outra biblioteca localmente, basta executarmos o comando **!pip install numpy**, uma vez que tal biblioteca será instalada em um espaço alocado em seu Drive.

```
!pip install numpy
```

Uma vez que o processo de instalação tenha sido finalizado, sem erros como esperado, podemos imediatamente fazer o uso das ferramentas disponíveis da biblioteca Numpy.

O processo de instalação local, caso você queira fazer o uso da biblioteca Numpy em sua IDE de preferência, também pode ser feito via Prompt de Comando, por meio de **pip install numpy**. Lembrando que caso você use ambientes virtualizados deve realizar a instalação dentro do environment do mesmo.



Para darmos início a nossos exemplos, já com nossa IDE aberta, primeiramente é necessário realizar a importação da biblioteca Numpy, uma vez que a mesma é uma biblioteca externa que por padrão não vem pré-carregada na maioria das IDEs.

O processo de importação é bastante simples, bastando criar a linha de código referente a importação da biblioteca logo no início de nosso código para que a biblioteca seja carregada em prioritariamente antes das demais estruturas de código, de acordo com a leitura léxica de nosso interpretador.

```
import numpy as np
```

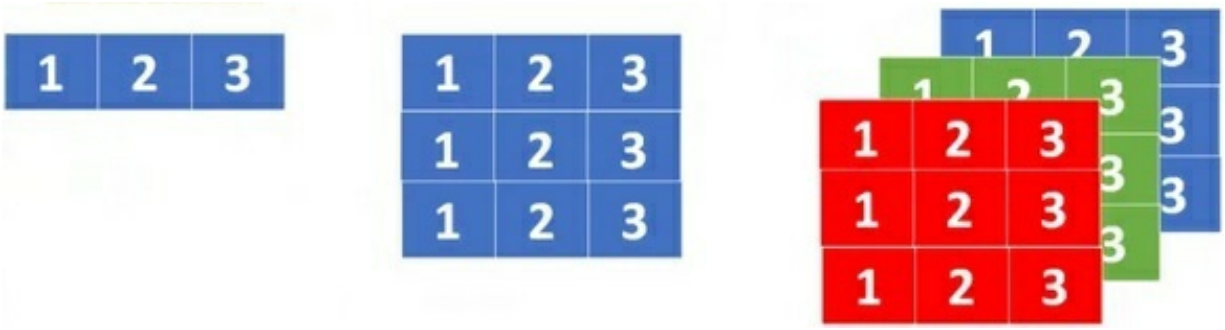
Uma prática comum é referenciar nossas bibliotecas por meio de alguma sigla, para que simplesmente fique mais fácil instanciar a mesma ao longo de nosso código. Neste caso, importamos a biblioteca numpy e a referenciamos como np apenas por convenção.

Trabalhando com Arrays

Em Python, assim como em outras linguagens de programação de alto nível, temos algumas estruturas de dados específicas quando queremos guardar mais de um dado/valor atribuído a uma variável/objeto.

Para aplicações comuns temos tipos de dados como listas, tuplas, sets, dicionários, etc... cada um com suas particularidades tanto na forma sintática quanto da maneira como tais dados são indexados e lidos por nosso interpretador.

No âmbito da computação científica uma prática comum é a do uso de arrays, estruturas de dados onde podemos guardar dados numéricos dispostos em forma de vetores e/ou matrizes, sendo assim ampliada a gama de aplicações de funções sobre estes tipos de dados em particular.



A partir de uma representação visual fica um tanto quanto mais fácil entendermos de fato o que seriam dados em forma vetorial/matricial.

Dos exemplos acima, à esquerda temos uma array unidimensional, estrutura de dados composta por campos preenchidos por dados numéricos dispostos ao longo de uma

simples linha. Cada elemento da mesma possui um índice interno ao qual podemos fazer uso para alguns tipos de funções básicas de exploração e tratamento dos dados.

Da mesma forma, ao meio temos uma representação de uma array bidimensional, onde os dados agora estão dispostos em linhas e colunas, o que nos amplia um pouco a gama de possibilidades sobre tais dados, haja visto que em muitas áreas trabalhamos diariamente com dados neste formato, comumente associado a tabelas Excel.

Por fim, à direita, temos a representação visual de uma array multidimensional, com dados dispostos em linhas, colunas e camadas, algo equivalente ao mapeamento de pixels e voxels em uma imagem, onde inúmeras dimensões podem ser adicionadas aumentando exponencialmente a complexidade de tais dados assim como as possibilidades sobre os mesmos.

Para computação científica, no que diz respeito ao tratamento de dados desta complexidade, a biblioteca Numpy nos oferecerá uma extensa biblioteca de funções onde a aplicação de conversões, funções, equações e outras explorações será feita de forma fácil e intuitiva, algo que quando realizado de forma manual pode ser bem maçante e complexo.

Desde a simples criação de uma estrutura de dados para armazenamento de dados/valores em forma matricial até aplicações mais avançadas, como o uso de matrizes como camadas de neurônios em redes neurais convolucionais*[1] (arquiteturas de rede neural dedicadas a conversão de dados a partir de mapeamento de imagens), tenha em mente que o uso de arrays do tipo Numpy facilitarão, e muito nossa vida.

*[1] Para ver tal aplicação, veja meu artigo “Detecção de doenças pulmonares a partir de radiografias de tórax, via rede neural artificial convolucional”, disponível em https://medium.com/@fernando2rad/rede-neural-artificial-convolucional-fernando-b-feltrin-6620064d6b71

Criando uma array numpy

Estando todas as dependências devidamente instaladas e carregadas, podemos finalmente dar início ao entendimento das estruturas de dados da biblioteca Numpy. Basicamente, já que estaremos trabalhando com dados por meio de uma biblioteca científica, é necessário contextualizarmos os tipos de dados envolvidos assim como as possíveis aplicações dos mesmos.

Sempre que estamos trabalhando com dados por meio da biblioteca Numpy, a primeira coisa que devemos ter em mente é que todo e qualquer tipo de dado em sua forma mais básica estará no formato de uma Array Numpy.

Quando estamos falando de arrays de forma geral basicamente estamos falando de dados em formato de vetores / matrizes, representados normalmente com dimensões bem definidas, alocando dados nos moldes de uma tabela com suas respectivas linhas, colunas e camadas.

A partir do momento que estamos falando de uma array “do tipo numpy”, basicamente estamos falando que tais dados estão carregados pela biblioteca Numpy de modo que possuem um sistema de indexação próprio da biblioteca e algumas métricas de mapeamento dos dados para que se possam ser aplicados sobre os mesmos uma série de funções, operações lógicas e aritméticas dependendo a necessidade.

```
data = np.array([2, 4, 6, 8, 10])
```

Inicialmente criamos uma variável / objeto de nome data, que recebe como atributo a função np.array( ) parametrizada com uma

simples lista de caracteres. Note que instanciamos a biblioteca numpy por meio de np, em seguida chamando a função np.array( ) que serve para transformar qualquer tipo de dado em array do tipo numpy.

```
print(data)
print(type(data))
print(data.shape)
```

Uma vez criada a array data com seus respectivos dados/valores, podemos realizar algumas verificações simples.

```
[ 2  4  6  8 10]
<class 'numpy.ndarray'>
(5,)
```

Na primeira linha vemos o retorno da função print( ) simplesmente exibindo o conteúdo de data. Na segunda linha o retorno referente a função print( ) parametrizada para verificação do tipo de dado por meio da função type( ) por sua vez parametrizado com data, note que, como esperado, trata-se de uma ‘numpy.ndarray’.

Por fim, na terceira linha temos o retorno referente ao tamanho da array por meio do método .shape, nesse caso, 5 elementos em uma linha, o espaço vazio após a vírgula indica não haver nenhuma coluna adicional.

Criando uma array gerada com números ordenados

Dependendo o propósito, pode ser necessário a criação de uma array composta de números gerados aleatoriamente, nesse contexto, a função np.arange( ) foi criada para este fim, bastando especificar o número de elementos que irão compor a array.

```
data = np.arange(15)

print(data)
```

Nesse caso, quando estamos falando em números ordenados estamos falando em números inteiros gerados de forma sequencial, do zero em diante.

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]
```

Novamente por meio da função print( ) parametrizada com data, é possível ver o retorno, nesse caso, uma array sequencial de 15 elementos gerados de forma ordenada.

Array gerada com números do tipo float

```
data = np.random.rand(15)

print(data)
```

Outra possibilidade que temos é a de criar uma array numpy de dados aleatórios tipo float, escalonados entre 0 e 1, por meio da função np.random.rand( ), novamente parametrizada com o número de elementos a serem gerados.

```
[0.18087199 0.32988976 0.73241758 0.5836709  0.02009198 0.42970031
 0.0398033  0.28778931 0.29019795 0.29858361 0.30491469 0.68747359
 0.99062792 0.71303427 0.16030855]
```

Da mesma forma, por meio da função print( ) é possível visualizar tais números gerados randomicamente. Note que neste caso os números são totalmente aleatórios, não sequenciais, dentro do intervalo de 0 a 1.

Array gerada com números do tipo int

```
data = np.random.randint(10, size = 10)

print(data)
```

Outro meio de criar uma array numpy é gerando números inteiros dentro de um intervalo específico definido manualmente para a função np.random.randint( ). Repare que o primeiro parâmetro da função, nesse caso, é o número 10, o que em outras palavras nos diz que estamos gerando números de 0 até 10, já o segundo parâmetro, size, aqui definido em 10, estipula que 10 elementos serão gerados aleatoriamente.

```
[4 9 0 9 8 2 2 3 5 6]
```

Mais uma vez por meio da função print( ) podemos visualizar a array gerada, como esperado, 10 números inteiros gerados aleatoriamente.

```
data = np.random.random((2, 2))

print(data)
```

De forma parecida com o que fizemos anteriormente por meio da função np.random.rand( ), podemos via np.random.random( ) gerar arrays de dados tipo float, no intervalo entre 0 e 1, aqui com mais de uma dimensão. Note que para isso é necessário colocar o número de linhas e colunas dentro de um segundo par de parênteses.

```
[[0.10310535 0.87117662]
 [0.4818465  0.4363957 ]]
```

Como de costume, por meio da função print( ) visualizamos os dados gerados e atribuídos a data.

Array gerada com números zero

```
data = np.zeros((3, 4))

print(data)
```

Dependendo da aplicação, pode ser necessário gerar arrays numpy previamente preenchidas com um tipo de dado específico. Em machine learning, por exemplo, existem várias aplicações onde se cria uma matriz inicialmente com valores zerados a serem substituídos após algumas funções serem executadas e gerarem retornos.

Aqui, por meio da função np.zeros( ) podemos criar uma array com as dimensões que quisermos, totalmente preenchida com números 0. Nesse caso, uma array com 3 linhas e 4 colunas deverá ser gerada.



Por meio da função print( ) novamente vemos o resultado, neste caso, uma matriz composta apenas de números 0.

Array gerada com números um

```
data = np.ones((3, 4))

print(data)
```

Da mesma forma como realizado no exemplo anterior, de acordo com as particularidades de uma aplicação pode ser necessário realizar a criação de uma array numpy inicialmente preenchida com números 1. Apenas substituindo a função np.zeros( ) por np.ones( ) criamos uma array nos mesmos moldes da anterior.

```
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]
```

Por meio da função print( ) visualizamos o resultado, uma matriz de números 1.

Array gerada com espaços vazios

```
data = np.empty((3, 4))

print(data)
```

Apenas como curiosidade, teoricamente é possível criar uma array numpy de dimensões predefinidas e com valores vazios por meio da função np.empty( ).

```
[[4.9e-323 1.5e-323 9.9e-324 1.5e-323]
 [2.0e-323 3.0e-323 2.0e-323 0.0e+000]
 [0.0e+000 9.9e-324 2.0e-323 0.0e+000]]
```

Porém o retorno gerado, como você pode visualizar na imagem acima, é uma array com números em notação científica, representando algo muito próximo de zero, porém não zero absoluto.

```
data = np.arange(10) * -1

print(data)
```

Concluindo essa linha de raciocínio, novamente dependendo da aplicação, pode ser necessário a geração de uma array composta de números negativos. Nesse caso não há uma função específica para tal fim, porém há o suporte da linguagem Python em permitir o uso do operador de multiplicação por um número negativo.

```
[ 0 -1 -2 -3 -4 -5 -6 -7 -8 -9]
```

Por meio da função print( ), como esperado, dessa vez temos uma array composta de 10 elementos negativos.

Criando uma array de números aleatórios, mas com tamanho predefinido em variável

Uma última possibilidade a ser explorada é a de associar a criação de uma array com um tamanho predefinido através de uma variável. Por meio da função np.random.permutation( ) podemos definir um tamanho que pode ser modificado conforme a necessidade,

```
tamanho = 10

data6 = np.random.permutation(tamanho)

print(data6)
```

Para esse fim, inicialmente criamos uma variável de nome tamanho que recebe como atributo o valor 10, este será o número a ser usado como referência de tamanho para a array, em outras palavras, o número de elementos da mesma.

Na sequência criamos nossa array data6 que recebe a função np.random.permitation( ) por sua vez parametrizada com o valor de tamanho. Por meio da função print( ) podemos ver o resultado desse gerador.

```
[1 8 6 7 0 3 9 5 4 2]
```

O resultado, como esperado, é uma array unidimensional composta por 10 elementos aleatoriamente gerados.

Criando arrays de dimensões específicas

Uma vez que estamos trabalhando com dados em forma matricial (convertidos e indexados como arrays numpy), uma característica importante a observar sobre os mesmos é sua forma representada, seu número de dimensões.

Posteriormente estaremos entendendo como realizar diversas operações entre arrays, mas por hora, raciocine que para que seja possível o cruzamento de dados entre arrays, as mesmas devem possuir formato compatível, nessa lógica, posteriormente veremos que para determinadas situações estaremos inclusive realizando a alteração do formato de nossas arrays para aplicação de certas funções.

Por hora, vamos entender da maneira correta como são definidas as dimensões de uma array numpy.

Array unidimensional

```
data = np.random.randint(5, size = 10

print(data)
print(data.shape)
```

Como já feito anteriormente, por meio da função np.random.randint( ) podemos gerar uma array, nesse caso, de 10 elementos com valores distribuídos entre 0 a 5.

```
[4 4 2 0 4 4 3 0 2 1]
(10,)
```

Por meio da função print( ) parametrizada com data vemos os valores da array, e parametrizando a mesma com data.shape podemos inspecionar de forma simples o formato de nossa array. Neste caso, [10,] representa uma array com 10 elementos em apenas uma linha. Uma array unidimensional.

Array bidimensional

```
data2 = np.random.randint(5, size = (3, 4))

print(data2)
print(data2.shape)
```

Para o próximo exemplo criamos uma variável de nome data2 que recebe, da mesma forma que o exemplo anterior, a função np.random.randint( ) atribuída para si, agora substituindo o valor de size por dois valores (3, 4), estamos definindo manualmente que o tamanho dessa array deverá ser de 3 linhas e 4 colunas, respectivamente. Uma array bidimensional.



Como esperado, por meio da função print( ) vemos a array em si com seu conteúdo, novamente via data2.shape vemos o formato esperado.

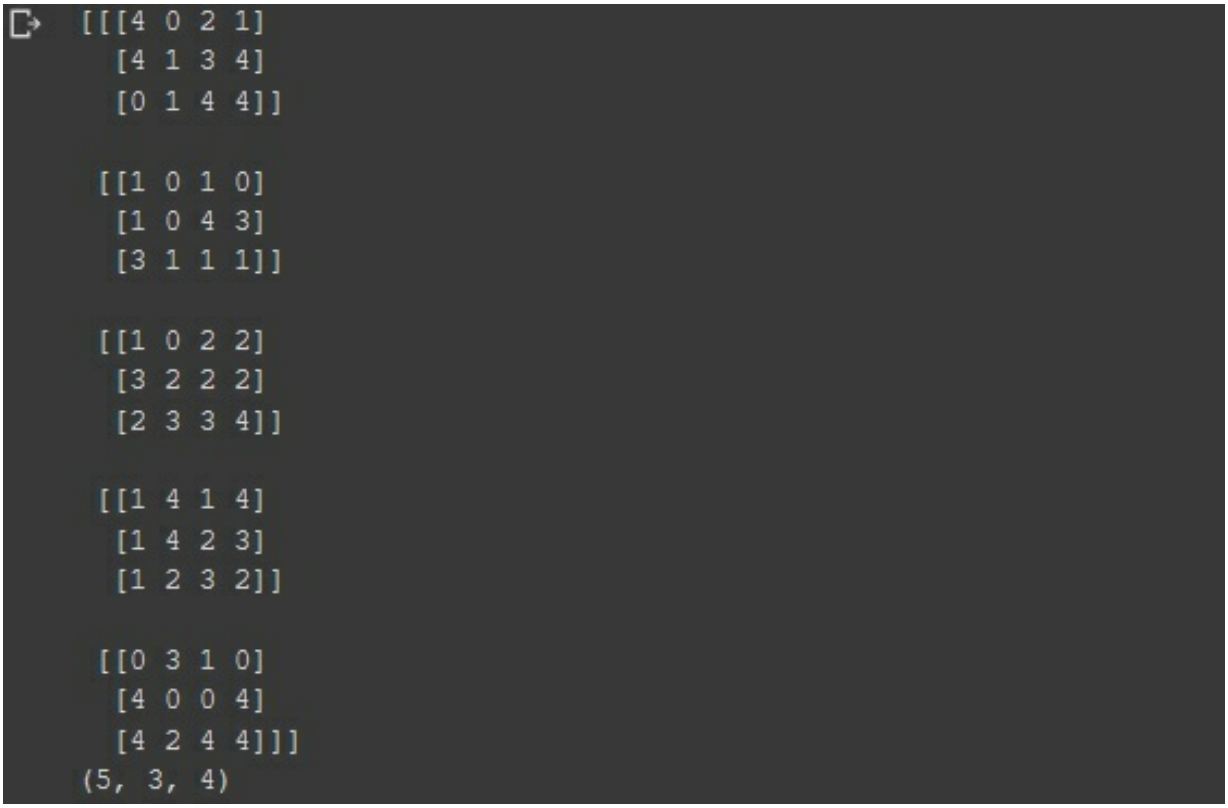
Array tridimensional

```
data3 = np.random.randint(5, size = (5, 3, 4))

print(data3)
print(data3.shape)
```

Seguindo com a mesma lógica, criamos a variável data3 que chama a função np.random.randint( ), alterando novamente o número definido para size, agora com 3 parâmetros, estamos definindo uma array de formato tridimensional.

Nesse caso, importante entender que o primeiro dos três parâmetros se refere ao número de camadas que a array terá em sua terceira dimensão, enquanto o segundo parâmetro define o número de linhas e o terceiro parâmetro o número de colunas.



```
[[[4 0 2 1]
  [4 1 3 4]
  [0 1 4 4]]

 [[1 0 1 0]
  [1 0 4 3]
  [3 1 1 1]]

 [[1 0 2 2]
  [3 2 2 2]
  [2 3 3 4]]

 [[1 4 1 4]
  [1 4 2 3]
  [1 2 3 2]]

 [[0 3 1 0]
  [4 0 0 4]
  [4 2 4 4]]]
(5, 3, 4)
```

Como esperado, 5 camadas, cada uma com 3 linhas e 4

colunas, preenchida com números gerados aleatoriamente entre 0 e 5.

```
data4 = np.array([[1, 2, 3, 4], [1, 3, 5, 7]])

print(data4)
```

Entendidos os meios de como gerar arrays com valores aleatórios, podemos agora entender também que é perfeitamente possível criar arrays manualmente, usando de dados em formato de lista para a construção da mesma. Repare que seguindo uma lógica parecida com o que já foi visto anteriormente, aqui temos duas listas de números, consequentemente, teremos uma array bidimensional.

```
[[1 2 3 4]
 [1 3 5 7]]
```

Exibindo em tela o resultado via função print( ) temos nossa array com os respectivos dados declarados manualmente. Uma última observação a se fazer é que imprescindível que se faça o uso dos dados declarados da maneira correta, inclusive na sequência certa.

Convertendo uma array multidimensional para unidimensional

Posteriormente veremos algumas possibilidades no que diz respeito a alterar a forma de uma array por meio de algumas funções específicas. Por hora, raciocine que uma prática comum é realizar a conversão de uma array multidimensional para unidimensional, de modo que se mantenha a indexação interna da mesma para que possa ser posteriormente convertida novamente para seu formato original, e isso é feito por meio da função flatten( ).

```
data4 = np.array([(1,2,3,4),(3,1,4,2)])

print(data4)

data4_flat = data4.flatten()

print(data4_flat)
```

Inicialmente criamos uma simples array numpy atribuída a variável data4. Em seguida é criada a variável data4_flat que recebe como atributo os dados de data4 processados pela função flatten( ).

```
[[1 2 3 4]
 [3 1 4 2]]
[1 2 3 4 3 1 4 2]
```

O retorno, como esperado, para primeira função print( ) é uma array bidimensional de 2 linhas e 4 colunas, assim como para segunda execução da função print( ) é exibida uma array unidimensional de 1 linhas e 8 colunas.

Verificando o tamanho e formato de uma array

Quando estamos trabalhando com dados em formato de vetor ou matriz, é muito importante eventualmente verificar o tamanho e formato dos mesmos, até porquê para ser possível realizar operações entre tais tipos de dados os mesmos devem ter um formato padronizado.

```
data3 = np.random.randint(5, size = (5, 3, 4))

print(data3.ndim)
print(data3.shape)
print(data3.size)
```

Indo diretamente ao ponto, nossa variável data3 é uma array do tipo numpy como pode ser observado. A partir daí é possível realizar algumas verificações, sendo a primeira delas o método .ndim, que por sua vez retornará o número de dimensões presentes em nossa array. Da mesma forma .shape nos retorna o formato de nossa array (formato de cada dimensão da mesma) e por fim .size retornará o tamanho total dessa matriz.

```
3
(5, 3, 4)
60
```

Como retorno das funções print( ) criadas, logo na primeira linha temos o valor 3 referente a .ndim, que em outras palavras significa tratar-se de uma array tridimensional. Em seguida, na segunda linha temos os valores 5, 3, 4, ou seja, 5 camadas, cada uma com 3 linhas e 4 colunas. Por fim na terceira linha temos o valor 60, que se refere a soma de todos os elementos que compõe essa array / matriz.

Verificando o tamanho em bytes de um item e de toda a array

Uma das questões mais honestas que todo estudante de Python (não somente Python, mas outras linguagens) possui é do por quê utilizar de bibliotecas externas quando já temos a disposição ferramentas nativas da linguagem.

Pois bem, toda linguagem de programação de código fonte aberto tende a receber forte contribuição da própria comunidade de usuários, que por sua vez, tentam implementar novos recursos ou aperfeiçoar os já existentes.

Com Python não é diferente, existe uma infinidade de bibliotecas para todo e qualquer tipo de aplicação, umas inclusive, como no caso da Numpy, mesclam implementar novas ferramentas assim como reestruturar ferramentas padrão em busca de maior performance.

Uma boa prática é manter em nosso código apenas o necessário, assim como levar em consideração o impacto de cada elemento no processamento de nossos códigos. Não é porque você vai criar um editor de texto que você precise compilar junto todo o sistema operacional. Inclusive se você já tem alguma familiaridade com a programação em Python sabe muito bem que sempre que possível importamos apenas os módulos e pacotes necessários de uma biblioteca, descartando todo o resto.

```
print(f'Cada elemento possui {data3.itemsize} bytes')
```

```
print(f'Toda a matriz possui {data3.nbytes} bytes')
```

Dado o contexto acima, uma das verificações comumente realizada é a de consultar o tamanho de cada elemento que faz

parte da array assim como o tamanho de toda a array. Isso é feito diretamente por meio dos métodos .itemsize e .nbytes, respectivamente, Aqui, apenas para variar um pouco, já implementamos essas verificações diretamente como parâmetro em nossa função print( ), porém é perfeitamente possível associar esses dados a uma variável.



Como esperado, junto de nossa mensagem declarada manualmente por meio de f’Strings temos os respectivos dados de cada elemento e de toda a matriz

Verificando o elemento de maior valor de uma array

Iniciando o entendimento das possíveis verificações que podemos realizar em nossas arrays, uma das mais comuns é buscar o elemento de maior valor em relação aos demais da array.

```
data = np.arange(15)

print(data)
print(data.max())
```

Criada a array data com 15 elementos ordenados, por meio da função print( ) parametrizando a mesma com os dados de data veremos tais elementos. Aproveitando o contexto, por meio da função max( ) teremos acesso ao elemento de maior valor associado.

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]
14
```

Como esperado, na primeira linha temos todos os elementos de data, na segunda linha temos em destaque o número 14, lembrando que a array tem 15 elementos, mas os mesmos iniciam em 0, logo, temos números de 0 a 14, sendo 14 o maior deles.

Consultando um elemento por meio de seu índice

Como visto anteriormente, a organização e indexação dos elementos de uma array se assemelha muito com a estrutura de dados de uma lista, e como qualquer lista, é possível consultar um determinado elemento por meio de seu índice, mesmo que esse índice não seja explícito.

Inicialmente vamos ver alguns exemplos com base em array unidimensional.

```
data = np.arange(15)

print(data)
print(data[8])
```

Criada a array data da mesma forma que o exemplo anterior, por meio da função print( ) parametrizada com data[ ] especificando um número de índice, é esperado que seja retornado o respectivo dado/valor situado naquela posição.

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]
8
```

Se não houver nenhum erro de sintaxe, é retornado o dado/valor da posição 8 do índice de nossa array data. Nesse caso, o retorno é o próprio número 8 uma vez que temos dados ordenados nesta array.

```
data

print(data)
print(data[-5])
```

Da mesma forma, passando como número de índice um valor negativo, será exibido o respectivo elemento a contar do final para o começo dessa array.

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]
10
```

Nesse caso, o 5º elemento a contar do fim para o começo é o número 10.

Partindo para uma array multidimensional, o processo para consultar um elemento é feito da mesma forma dos exemplos anteriores, bastando agora especificar a posição do elemento levando em conta a linha e coluna onde é esperado.

```
data2 = np.random.randint(5, size = (3, 4))

print(data2)
print(data2[0, 2])
```

Note que é criada a array data2, de elementos aleatórios entre 0 e 5, tendo 3 linhas e 4 colunas em sua forma. Parametrizando print com data2[0, 2] estamos pedindo que seja exibido o elemento situado na linha 0 (primeira linha) e coluna 2 (terceira coluna pois a primeira coluna é 0).

```
[[0 1 4 2]
 [4 3 4 0]
 [1 0 1 1]]
4
```

Podemos visualizar via console a array em si, e o respectivo elemento situado na primeira linha, terceira coluna. Neste caso, o número 4.

Consultando elementos dentro de um intervalo

```
data

print(data)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]
```

Reutilizando nossa array data criada anteriormente, uma simples array unidimensional de 15 elementos ordenados, podemos avançar com o entendimento de outras formas de consultar dados/valores, dessa vez, definindo intervalos específicos.

```
print(data)
print(data[:5])
print(data[3:])
print(data[4:8])
print(data[::2])
print(data[3::])
```

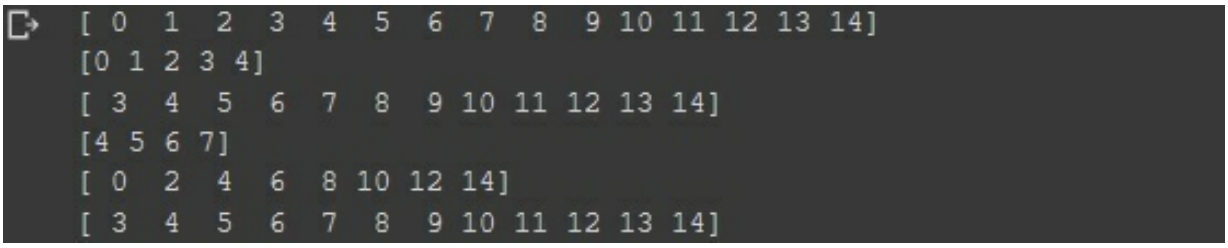
Como mencionado anteriormente, a forma como podemos consultar elementos via índice é muito parecida (senão igual) a forma como realizávamos as devidas consultas em elementos de uma lista. Sendo assim, podemos usar das mesmas notações aqui.

Na primeira linha simplesmente por meio da função print( ) estamos exibindo em tela os dados contidos em data.

Na sequência, passando data[:5] (mesmo que data[0:5]) como parâmetro para nossa função print( ) estaremos exibindo os 5 primeiros elementos dessa array. Em seguida via data[3:] estaremos exibindo do terceiro elemento em diante todos os demais. Na sequência, via data[4:8] estaremos exibindo do quarto ao oitavo elemento.

Da mesma forma, porém com uma pequena diferença de

notação, podemos por meio de data[::2] exibir de dois em dois elementos, todos os elementos. Por fim, via data[3::] estamos pulando os 3 primeiros elementos e exibindo todos os demais.



Como esperado, na primeira linha temos todos os dados da array, na segunda linha apenas os 5 primeiros, na terceira linha do terceiro elemento em diante, na quarta linha os elementos entre 4 e 8, na quinta linha os elementos pares e na sexta linha todos os dados exceto os três primeiros.

Modificando manualmente um dado/valor de um elemento por meio de seu índice.

Assim como em listas podíamos realizar modificações diretamente sobre seus elementos, bastando saber seu índice, aqui estaremos realizando o mesmo processo da mesma forma.

```
data2

print(data2)

data2[0, 0] = 10

print(data2)
```

Reutilizando nossa array data 2 criada anteriormente, podemos realizar a alteração de qualquer dado/valor desde que se faça a referência correta a sua posição na array de acordo com seu índice. Nesse caso, apenas como exemplo, iremos substituir o valor do elemento situado na linha 0 e coluna 0 por 10.

```
[[1 3 2 3]
 [4 0 4 0]
 [0 2 4 0]]
[[10  3  2  3]
 [ 4  0  4  0]
 [ 0  2  4  0]]
```

Repare que como esperado, o valor do elemento [0, 0] foi substituído de 1 para 10 como mostra o retorno gerado via print( ).

```
data2

data2[1, 1] = 6.82945

print(data2)
```

Apenas como curiosidade, atualizando o valor de um elemento

com um número do tipo float, o mesmo será convertido para int para que não hajam erros por parte do interpretador.

```
[[10  3  2  3]
 [ 4  6  4  0]
 [ 0  2  4  0]]
```

Elemento [1, 1] em formato int, sem as casas decimais declaradas manualmente.

```
data2 = np.array([8, -3, 5, 9], dtype = 'float')

print(data2)
print(type(data2[0]))
```

Supondo que você realmente precise dos dados em formato float dentro de uma array, você pode definir manualmente o tipo de dado por meio do parâmetro dtype = ‘float’. Dessa forma, todos os elementos desta array serão convertidos para float.

Aqui como exemplo estamos criando novamente uma array unidimensional, com números inteiros em sua composição no momento da declaração.

```
[ 8. -3.  5.  9.]
<class 'numpy.float64'>
```

Analisando o retorno de nossa função print( ) podemos ver que de fato são números, agora com casas decimais, de tipo float64.

Criando uma array com números igualmente distribuídos

Dependendo mais uma vez do contexto, pode ser necessária a criação de uma array com dados/valores distribuídos de maneira uniforma na mesma. Tenha em mente que os dados gerados de forma aleatória não possuem critérios definidos quanto sua organização. Porém, por meio da função linspace( ) podemos criar dados uniformemente arranjados dentro de uma array.

```
data = np.linspace(0, 1, 7)

print(data)
```

Criamos uma nova array de nome data, onde por meio da função np.linspace( ) estamos gerando uma array composta de 7 elementos igualmente distribuídos, com valores entre 0 e 1.

```
[0.         0.16666667 0.33333333 0.5        0.66666667 0.83333333
 1.        ]
```

O retorno como esperado é uma array de números float, para que haja divisão igual dos valores dos elementos.

Redefinindo o formato de uma array

Um dos recursos mais importantes que a biblioteca Numpy nos oferece é a de poder livremente alterar o formato de nossas arrays. Não que isso não seja possível sem o auxílio da biblioteca Numpy, mas a questão mais importante aqui é que no processo de reformatação de uma array numpy, a mesma se reajustará em todos os seus níveis, assim como atualizará seus índices para que não se percam dados entre dimensões.

```
data5 = np.arange(8)

print(data5)
```

Inicialmente apenas para o exemplo criamos nossa array data5 com 8 elementos gerados e ordenados por meio da função np.arange( ).

```
data5 = data5.reshape(2, 4)

print(data5)
```

Em seguida, por meio da função reshape( ) podemos alterar livremente o formato de nossa array. Repare que inicialmente geramos uma array unidimensional de 8 elementos, agora estamos transformando a mesma para uma array bidimensional de 8 elementos, onde os mesmos serão distribuídos em duas linhas e 4 colunas.

Importante salientar que você pode alterar livremente o formato de uma array desde que mantenha sua proporção de distribuição de dados. Aqui, nesse exemplo bastante simples, 8 elementos dispostos em uma linha passarão a ser 8 elementos dispostos em duas linhas e 4 colunas. O número de elementos em si

não pode ser alterado.



Na primeira linha, referente ao primeiro print( ) temos a array data5 em sua forma inicial, na segunda linha o retorno obtido pós reshape.

Usando de operadores lógicos em arrays

Dependendo o contexto pode ser necessário fazer uso de operadores lógicos mesmo quando trabalhando com arrays numpy. Respeitando a sintaxe Python, podemos fazer uso de operadores lógicos sobre arrays da mesma forma como fazemos com qualquer outro tipo de dado.

```
data7 = np.arange(10)

print(data7)
print(data7 > 3)
```

Para esse exemplo criamos nossa array data7 de 10 elementos ordenados gerador aleatoriamente. Por meio da função print( ) podemos tanto exibir seu conteúdo quanto realizar proposições baseadas em operadores lógicos. Note que no exemplo a seguir, estamos verificando quais elementos de data7 tem seu valor maior que 3.

```
[0 1 2 3 4 5 6 7 8 9]
[False False False False  True  True  True  True  True  True]
```

Na primeira linha todo o conteúdo de data7, na segunda marcados como False os elementos menores que 3, assim como marcados como True os de valor maior que 3.

```
print(data7)
print(data7[4] > 5)
```

Da mesma forma como visto anteriormente, fazendo a seleção de um elemento por meio de seu índice, também podemos aplicar um operador lógico sobre o mesmo.

```
[0 1 2 3 4 5 6 7 8 9]
False
```

Nesse caso, o elemento da posição 4 do índice, o número 3, não é maior que 5, logo, o retorno sobre essa operação lógica é False.

Usando de operadores aritméticos em arrays

Dentro das possibilidades de manipulação de dados em arrays do tipo numpy está a de realizar, como esperado, operações aritméticas entre tais dados. Porém recapitulando o básico da linguagem Python, quando estamos trabalhando com determinados tipos de dados temos de observar seu tipo e compatibilidade com outros tipos de dados, sendo em determinados casos necessário a realização da conversão entre tipos de dados.

Uma array numpy permite realizar determinadas operações que nos mesmos tipos de dados em formato não array numpy iriam gerar exceções ou erros de interpretação.

```
data8 = [1, 2, 3, 4, 5, 6, 7, 8]

print(data8 + 10)
```

Para entendermos melhor, vamos usar do seguinte exemplo, temos uma array data8 com uma série de elementos dispostos em formato de lista, o que pode passar despercebido aos olhos do usuário, subentendendo que para este tipo de dado é permitido realizar qualquer tipo de operação. Por meio da função print( ) parametrizada com data8 + 10, diferentemente do esperado, irá gerar um erro.

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-8-61e14f649c93> in <module>()
      1 data8 = [1, 2, 3, 4, 5, 6, 7, 8]
----> 2 print(data8 + 10)

TypeError: can only concatenate list (not "int") to list
```

Repare que o interpretador lê os dados de data8 como uma

simples lista, tentando por meio do operador + concatenar os dados com o valor 10 ao invés de realizar a soma.

```
data8 = np.array(data8)

print(data8 + 10)
```

Realizando a conversão de data8 para uma array do tipo numpy, por meio da função np.array( ) parametrizada com a própria variável data8, agora é possível realizar a operação de soma dos dados de data8 com o valor definido 10.

```
[1, 2, 3, 4, 5, 6, 7, 8]
[11 12 13 14 15 16 17 18]
```

Como retorno da função print( ), na primeira linha temos os valores iniciais de data8, na segunda linha os valores após somar cada elemento ao número 10.

Criando uma matriz diagonal

Aqui mais um dos exemplos de propósito específico, em algumas aplicações de machine learning é necessário a criação de uma matriz diagonal, e a mesma pode ser feita por meio da função np.diag( ).

```
data9 = np.diag((1, 2, 3, 4, 5, 6, 7, 8))

print(data9)
```

Para esse exemplo criamos a array data9 que recebe como atributo uma matriz diagonal de 8 elementos definidos manualmente, gerada pela função np.diag( ).



```
[[1 0 0 0 0 0 0 0]
 [0 2 0 0 0 0 0 0]
 [0 0 3 0 0 0 0 0]
 [0 0 0 4 0 0 0 0]
 [0 0 0 0 5 0 0 0]
 [0 0 0 0 0 6 0 0]
 [0 0 0 0 0 0 7 0]
 [0 0 0 0 0 0 0 8]]
```

O retorno é uma matriz gerada com os elementos predefinidos compondo uma linha vertical, sendo todos os outros espaços preenchidos com 0.

Algo parecido pode ser feito através da função np.eye( ), porém, nesse caso será gerada uma matriz diagonal com valores 0 e 1, de tamanho definido pelo parâmetro repassado para função.

```
data9 = np.eye(4)

print(data9)
```

Neste caso, para a variável data9 está sendo criada uma array diagonal de 4 linhas e colunas conforme a parametrização realizada em np.eye( ).



O retorno é uma matriz diagonal, com números 1 dispostos na coluna diagonal

Criando padrões duplicados

Por meio da simples notação de “encapsular” uma array em um novo par de colchetes e aplicar a função np.tile( ) é possível realizar a duplicação/replicação da mesma quantas vezes for necessário.

```
data10 = np.tile(np.array([[9, 4], [3, 7]]), 4)

print(data10)
```

Aqui criada a array data10, note que atribuído para a mesma está a função np.tile( ) parametrizada com uma array definida manualmente assim como um último parâmetro 4, referente ao número de vezes a serem replicados os dados/valores da array.

```
[[9 4 9 4 9 4 9 4]
 [3 7 3 7 3 7 3 7]]
```

No retorno podemos observar os elementos declarados na array, em sua primeira linha 9 e 4, na segunda, 3 e 7, repetidos 4 vezes.

```
data10 = np.tile(np.array([[9, 4], [3, 7]]), (2,2))

print(data10)
```

Mesmo exemplo anterior, agora parametrizado para replicação em linhas e colunas.

```
[[9 4 9 4]
 [3 7 3 7]
 [9 4 9 4]
 [3 7 3 7]]
```

Note que, neste caso, a duplicação realizada de acordo com a parametrização ocorre sem sobrepor a ordem original dos elementos.

Somando um valor a cada elemento da array

Como visto anteriormente, a partir do momento que uma matriz passa a ser uma array do tipo numpy, pode-se perfeitamente aplicar sobre a mesma qualquer tipo de operador lógico ou aritmético. Dessa forma, é possível realizar tais operações, desde que leve em consideração que a operação realizada se aplicará a cada um dos elementos da array.

```
data11 = np.arange(0, 15)

print(data11)

data11 = np.arange(0, 15) + 1

print(data11)
```

Para esse exemplo criamos uma nova array de nome data11, por sua vez gerada com números ordenados. Em seguida é realizada uma simples operação de somar 1 a própria array, e como dito anteriormente, essa soma se aplicará a todos os elementos.

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]
[ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15]
```

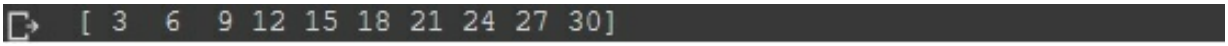
Observando os retornos das funções print( ) na primeira linha temos a primeira array com seus respectivos elementos, na segunda, a mesma array onde a cada elemento foi somado 1 ao seu valor original.

```
data11 = np.arange(0, 30, 3) + 3

print(data11)
```

Apenas como exemplo, usando da notação vista anteriormente é possível realizar a soma de um valor a cada elemento de uma array, nesse caso, somando 3 ao valor de cada elemento, de 3 em 3

elementos de acordo com o parâmetro passado para função np.arange( ).



Como esperado, o retorno é uma array gerada com números de 0 a 30, com intervalo de 3 entre cada elemento.

Realizando soma de arrays

Entendido boa parte do processo lógico das possíveis operações em arrays não podemos nos esquecer que é permitido também a soma de arrays, desde que as mesmas tenham o mesmo formato ou número de elementos.

```
d1 = np.array([ 3,  6,  9, 12, 15])
d2 = np.array([ 1,  2,  3,  4,  5])

d3 = d1 + d2

print(d3)
```

Para esse exemplo criamos duas arrays d1 e d2, respectivamente, cada uma com 5 elementos distintos. Por meio de uma nova variável de nome e3 realizamos a soma simples entre d1 e d2, e como esperado, cada elemento de cada array será somado ao seu elemento equivalente da outra array.

```
[ 4  8 12 16 20]
```

O resultado obtido é a simples soma do primeiro elemento da primeira array com o primeiro elemento da segunda array, assim como todos os outros elementos com seu equivalente.

```
d1 = np.array([ 3,  6,  9, 12, 15])
d2 = np.array([ 1,  2,  3,  4,  5])

d3 = d1 - d2

print(d3)
```

Subtração entre arrays

Exatamente da mesma forma é possível realizar a subtração entre arrays.



Como esperado, é obtido os valores das subtrações entre cada elemento da array d1 com o seu respectivo elemento da array d2.

Multiplicação e divisão entre arrays

```
d1 = np.array([ 3,  6,  9, 12, 15])
d2 = np.array([ 1,  2,  3,  4,  5])

d3 = d1 / d2

d4 = d1 * d2

print(d3)
print(d4)
```

Exatamente da mesma forma é possível realizar a divisão e a multiplicação entre os elementos das arrays.

```
[3. 3. 3. 3. 3.]
[ 3 12 27 48 75]
```

Obtendo assim, na primeira linha o retorno da divisão entre os elementos das arrays, na segunda linha o retorno da multiplicação entre os elementos das mesmas.

Realizando operações lógicas entre arrays

Da mesma forma que é possível realizar o uso de operadores aritméticos para seus devidos cálculos entre elementos de arrays, também é uma prática perfeitamente possível fazer o uso de operadores lógicos para verificar a equivalência de elementos de duas ou mais arrays.

```
d1 = np.array([ 3,  6,  9, 12, 15, 18, 21])
d2 = np.array([ 1,  2,  3,  4,  5, 20, 25])

d3 = d2 > d1

print(d3)
```

Neste exemplo, duas arrays d1 e d2 novamente, agora com algumas pequenas mudanças em alguns elementos, apenas para tornar nosso exemplo mais interessante. Em nossa variável d3 estamos usando um operador para verificar se os elementos da array d2 são maiores que os elementos equivalentes em d1. Novamente, o retorno será gerado mostrando o resultado dessa operação lógica para cada elemento.

```
[False False False False False  True  True]
```

Repare que por exemplo o 4º elemento da array d2 se comparado com o 4º elemento da array d1 retornou False, pois 4 não é maior que 12. Da mesma forma o último elemento de d2 se comparado com o último elemento de d1 retornou True, pois obviamente 25 é maior que 22.

Transposição de arrays

Em determinadas aplicações, principalmente em machine learning, é realizada a chamada transposição de matrizes, que nada mais é do que realizar uma alteração de forma da mesma, transformando linhas em colunas e vice versa por meio da função transpose( ).

```
arr1 = np.array([[1, 2, 3], [4, 5, 6]])

print(arr1)
print(arr1.shape)

arr1_transposta = arr1.transpose()

print(arr1_transposta)
print(arr1_transposta.shape)
```

Note que inicialmente é criada uma array de nome arr1, bidimensional, com elementos distribuídos em 2 linhas e 3 colunas. Na sequência é criara uma nova variável de nome arr1_transposta que recebe o conteúdo de arr1, sob a função transpose( ). Como sempre, por meio da função print( ) realizaremos as devidas verificações.

```
[[1 2 3]
 [4 5 6]]
(2, 3)
[[1 4]
 [2 5]
 [3 6]]
(3, 2)
```

Como esperado, na primeira parte de nosso retorno temos a array arr1 em sua forma original, com shape (2, 3) e logo abaixo a array arr1_transposta, que nada mais é do que o conteúdo de arr1 reorganizados no que diz respeito as suas linhas e colunas, nesse

caso com o shape (3, 2).

Salvando uma array no disco local

```
data = np.array([ 3,  6,  9, 12, 15])
```

```
np.save('minha_array', data)
```

Encerrando nossos estudos, vamos ver como é possível salvar nossas arrays localmente para reutilização. Para isso simplesmente criamos uma array comum atribuída a variável data.

Em seguida por meio da função np.save( ) podemos de fato exportar os dados de nossa array para um arquivo loca. Note que como parâmetros da função np.save( ) em primeiro lugar repassamos em forma de string um nome para nosso arquivo, seguido do nome da variável que possui a array atribuída para si.

O arquivo gerado será, nesse caso, minha_array.npy.

Carregando uma array do disco local

Da mesma forma que era possível salvar nossa array em um arquivo no disco local, outra possibilidade que temos é de justamente carregar este arquivo para sua utilização.

```
np.load('minha_array.npy')
```

Através da função np.load( ), bastando passar como parâmetro o nome do arquivo com sua extensão, o carregamento será feito, inclusive já alocando em memória a array para que se possam realizar operações sobre a mesma.

CIÊNCIA DE DADOS E APRENDIZADO DE MÁQUINA

Nota do Autor

Seja muito bem vindo(a) ao mundo das Redes Neurais Artificiais, em tradução livre, Neural Networks. Ao longo deste pequeno livro introdutório estaremos entendendo de forma simples e objetiva o que é essa área da computação, quais suas particularidades e aplicações. Assim como de forma gradual estaremos pondo em prática a codificação de modelos e estruturas de redes neurais artificiais, aprendizado de máquina e tudo o que está ao seu entorno.

Tenha em mente que este livro tem o intuito principal de desenvolver bases bastante sólidas sobre essa que em minha humilde opinião, é uma das mais incríveis áreas da computação. Independentemente se você já atua nessa área, ou se já é um programador, ou se é um entusiasta que por seus motivos está começando agora nessa área, fique tranquilo pois todo conteúdo será abordado de uma forma bastante simples, didática e prática, de modo a desmistificar muita informação errônea que existe nesse meio assim como de forma gradativa aprender de fato o que é ciência de dados por meio de suas aplicações.

Com dedicação e muita prática tenho certeza que ao final desse livro você terá plenas condições de, seja para fins de estudo ou profissionais, criar seus próprios modelos e/ou adaptar modelos já existentes contextualizando-os para seus devidos fins.

Um forte abraço.

Fernando Feltrin

O que é Ciência de Dados e Aprendizado de Máquina?

Na área de tecnologia temos muitas áreas de especialização, o que normalmente é desconhecido do público comum e até mesmo de alguns profissionais da área, pois dependendo da sua área de formação e atuação, costumam se ater a apenas o próprio nicho se especializando a fundo apenas nele. A grande variedade de subdivisões se dá não só pela grande variedade de cursos de nível técnico, superior e/ou especializações que existem nesse meio formando especialistas de cada área, mas principalmente porque cada segmento da tecnologia possui muitas áreas de especialização dentro de cada nicho e um universo de informação ainda a ser explorado.

Especificamente dentro da área da ciência da computação, nicho da programação, existem também muitas subdivisões onde encontramos desde a criação de programas e ferramentas que possuem propósito final de resolver algum problema de forma computacional até a análise e manipulação de dados de forma científica. Uma destas áreas específicas é a que trata das particularidades da chamada Inteligência Artificial, sub área esta que trata desde processos de automação até modelos que simulam tipos de inteligência para os mais diversos propósitos.

Na verdade, existem muitas áreas que podem englobar ou compartilhar conceitos umas com as outras, o que nos dias atuais torna difícil até mesmo tentar dividir as áreas da tecnologia ou categorizá-las individualmente de acordo com suas particularidades. Uma destas áreas da computação, independentemente de qual classe, chamada Data Science, ou em tradução livre, Ciência de Dados, está muito em alta nos dias atuais, porém sua idealização se deu em meados da década de 50 e ainda estamos longe de explorar todo seu real potencial.

Basicamente se idealizaram na década de 50 rotinas (algoritmos) e modelos computacionais para se trabalhar processando grandes volumes de dados fazendo o uso do poder computacional da época para seu devido processamento. Você já deve imaginar que naquela época não tínhamos poder computacional para pôr em prática tais conceitos, logo, muitas dessas

ideias acabaram sendo engavetadas.

Porém décadas depois tivemos enormes saltos tecnológicos e finalmente hoje temos poder de processamento de sobra, fazendo o uso de GPUs de alta performance, clusters de processamento paralelo e métodos de processamento onde realmente conseguimos trabalhar com enormes volumes de dados para fins de se analisar e obter informações a partir dos mesmos, sem falar de outras tecnologias que foram criadas em paralelo que nos permitiram avançar de forma exponencial. Hoje já é realidade termos também automação para praticamente qualquer coisa, assim como hardware e software desenvolvidos para aumentar e melhorar processos até então realizados puramente por pessoas com base em sua baixa eficiência se comparado a um computador.

Sendo assim retomamos e aperfeiçoamos aqueles conceitos lá da longínqua década de 50 para hoje termos diferentes áreas de processamento de dados sendo cada vez mais usadas de forma a criar impacto no mundo real. Muitos autores inclusive veem essa área da computação como uma evolução natural da ciência estatística que, combinada a toda automação que desenvolvemos, resultou num salto tecnológico sem precedentes. A tecnologia está infiltrada em tudo e todas as outras áreas, colaborando para que nestes dias atribulados consigamos criar e produzir de modo a suprir as demandas desse mundo moderno.

Como comentado anteriormente, o conceitual teórico de criar formas e métodos de fazer com que computadores “trabalhem por nós” já existe de muito tempo. Hoje conseguimos concretizar várias dessas ideias graças aos avanços tecnológicos que obtivemos, seja em estrutura física ou lógica, a ponto de literalmente estarmos criando literalmente inteligência artificial, sistemas embarcados, integrados na chamada internet das coisas, softwares que automatizam processos e não menos importante que os exemplos anteriores, a digitalização de muita informação analógica que ainda hoje trabalhamos, mesmo numa fase de grandes avanços tecnológicos, ainda vivemos essa transição.

Falando da aplicação de “dados”, raciocine que em outras épocas, ou pelo menos em outras eras tecnológicas, independente do mercado, se produzia com base na simples aplicação de modelos administrativos, investimento e com base em seu retorno, com um pouco de estatística se

criavam previsões para o mercado a médio prazo. Nesse processo quase que de tentativa e erro tínhamos modelos de processos com baixa eficiência, alto custo e consequentemente desgaste por parte dos colaboradores.

Hoje independentemente de que área que estivermos abordando, existem ferramentas computacionais de automação que, sem grandes dificuldades conseguem elevar muito a eficiência dos processos num mundo cada vez mais acelerado e interconectado, sedento por produtividade, com enormes volumes de dados gerados e coletados a cada instante que podemos usar de forma a otimizar nossos processos e até mesmo prever o futuro dos mercados.

Para ficar mais claro, imagine o processo de desenvolvimento de uma vacina, desde sua lenta pesquisa até a aplicação da mesma, tudo sob enormes custos, muito trabalho de pesquisa, tentativa e erro para poucos e pífios resultados. Hoje, falando a nível de ciência, se evolui mais em um dia do que no passado em décadas, e cada vez mais a tecnologia irá contribuir para que nossos avanços em todas as áreas sejam de forma mais acelerada e precisa.

Seguindo com o raciocínio, apenas por curiosidade, um exemplo bastante interessante é o da criação da vacina contra o vírus da raiva. Idealizada e pesquisada por Louis Pasteur, posteriormente publicada e liberada em 1885, na época de sua pesquisa a história nos conta que em seu laboratório quando ele ia coletar amostras de tecidos, de animais em diferentes estágios da infecção para que fossem feitas as devidas análises e testes. Para realização dos devidos procedimentos iam até o ambiente das cobaias ele, um assistente e era levado consigo uma espingarda… A espingarda era para matar ele caso houvesse sido contaminado, uma vez que naquela época esta ainda era uma doença fatal, de quadro clínico irreversível uma vez que a pessoa fosse contaminada.

Hoje conseguimos usar de poder computacional para reduzir tempo e custos (principalmente em pesquisa e no âmbito de criar modelos onde se usa de ferramentas computacionais para simular os cenários e as aplicações e seus respectivos resultados, em milhões de testes simulados), aplicar a vacina de forma mais adaptada às necessidades de uma população ou área geográfica e ainda temos condições de aprender com esses dados para melhorar ainda mais a eficiência de todo este processo para o futuro. Da mesma forma, existe muita pesquisa científica em praticamente todas as áreas de conhecimento

usando das atuais ferramentas de ciência de dados e das técnicas de aprendizagem de máquina para criar novos prospectos e obter exponenciais avanços na ciência.

Neste pequeno livro iremos abordar de forma bastante prática os principais conceitos desta gigante área de conhecimento, a fim de desmistificar e principalmente de elucidar o que realmente é esta área de inteligência artificial, ciência de dados e aprendizagem de máquina, além de suas vertentes, subdivisões e aplicações. Estaremos criando de forma simples e procedural uma bagagem de conhecimento sobre os principais tópicos sobre o assunto assim como estaremos pondo em prática linha a linha de código com suas devidas explicações para que ao final desse livro você tenha condições não só de trabalhar em cima de ferramentas de computação científica, mas criar seus próprios modelos de análise de dados e ter condições de os adaptar para suas aplicações diárias.

O que são Redes Neurais Artificiais

Desde os primeiros estudos por parte biológico e psicológico dos mecanismos que temos de aprendizado, descobrindo nossa própria biologia (no sentido literal da palavra), criamos modelos lógicos estruturados buscando abstrair tais mecanismos para que pudéssemos fazer ciência sobre os mesmos.

À medida que a neurociência evoluía, assim como áreas abstratas como psicologia também, pudemos cada vez melhor entender o funcionamento de nosso cérebro, desde os mecanismos físicos e biológicos até os que nos tornam autoconscientes e cientes de como aprendemos a medida que experenciamos esse mundo.

Não por acaso, as primeiras tentativas de abstrairmos tais mecanismos de forma lógico-computacional se deram literalmente, recriando tais estruturas anatômicas e mecanismos de aprendizado de forma artificial, por meio das chamadas redes neurais artificiais, com seu sistema de neuronal sináptico..

Existe uma grande variedade de tipos/modelos de redes neurais artificiais de acordo com sua estrutura e propósito, em comum a todas elas estão por exemplo conceitos de perceptrons, onde literalmente criamos neurônios artificiais e interação entre os mesmos para que possam encontrar padrões em dados, usando de poder computacional como vantagem no que diz respeito ao número de informações a serem processadas e o tempo de processamento muito superior aos limites humanos.

Dessa forma, partindo de preceitos de problemas computacionais a serem resolvidos (ou problemas reais abstraídos de forma computacional), temos modelos de redes neurais artificiais que conseguem ler e interpretar dados a partir de praticamente todo tipo de arquivo (inclusive imagens e sons), realizar classificações para os mais diversos fins, assim como realizar previsões com base em histórico de dados, simular algoritmos genéticos, também realizar processamento de linguagem natural, geração de novos dados a partir de amostras, reconhecimento por conversão, simulação de sentidos biológicos como visão, audição e outros com base em leitura de sensores, aprendizado com base em experiência, tomadas de decisão com

base em memória, etc... Uma verdadeira infinidade de possibilidades.

Raciocine que nesse processo de criarmos arquiteturas de redes neurais artificiais, poderemos simular praticamente qualquer tipo de inteligência de um ser vivo biológico de menor complexidade. Ainda estamos engatinhando nessa área, mesmo com décadas de evolução, porém ao final desse livro procure ver à frente, todo o potencial que temos para as próximas décadas à medida que aperfeiçoamos nossos modelos.

O que é Aprendizado de Máquina

Entendendo de forma simples, e ao mesmo tempo separando o joio do trigo, precisamos de fato contextualizar o que é uma rede neural artificial e como se dá o mecanismo de aprendizado de máquina. Raciocine que, como mencionado anteriormente, temos um problema computacional a ser resolvido. Supondo que o mesmo é um problema de classificação, onde temos amostras de dois ou mais tipos de objetos e precisamos os classificar conforme seu tipo.

Existe literalmente uma infinidade de algoritmos que poderiam realizar essa tarefa, mas o grande pulo do gato aqui é que por meio de um modelo de rede neural, podemos treinar a mesma para que reconheça quais são os objetos, quais são suas características relevantes e assim treinar e testar a mesma para que com algumas configurações consiga realizar essa tarefa de forma rápida e precisa, além de que uma vez aprendida essa classificação, essa rede neural pode realizar novas classificações com base nos seus padrões aprendidos previamente.

Para esse processo estaremos criando toda uma estrutura de código onde a rede terá dados de entrada, oriundos das amostras a serem classificadas, processamento via camadas de neurônios artificiais encontrando padrões e aplicando funções matemáticas sobre os dados, para ao final do processo separar tais dados das amostras conforme o seu tipo.

Neste processo existe o que é chamado de aprendizado de máquina supervisionado, onde literalmente temos que mostrar o caminho para a rede, mostrar o que é o resultado final correto para que ela aprenda os padrões intermediários que levaram até aquela resolução. Assim como também teremos modelos de aprendizado não supervisionado, onde a rede com base em alguns algoritmos conseguirá sozinha identificar e reconhecer os padrões necessários.

Como se trabalha com Ciência de Dados? Quais ferramentas utilizaremos?

Se tratando de ciência de dados, e como já mencionado anteriormente, estamos falando de um dos nichos da área de programação, logo, estaremos trabalhando em uma camada intermediária entre o usuário e o sistema operacional. Usando uma linguagem de programação e ferramentas criadas a partir dela para facilitar nossas vidas automatizando uma série de processos, que diga-se de passagem em outras épocas era necessário se criar do zero, teremos a faca e o queijo nas mãos para criar e pôr em uso nossos modelos de análise e processamento de dados.

Falando especificamente de linguagem de programação, estaremos usando uma das linguagens mais “queridinhas” dentro deste meio que é a linguagem Python. Python é uma linguagem de sintaxe simples e grande eficiência quando se trata de processamento em tempo real de grandes volumes de dados, dessa forma, cada tópico abordado e cada linha de código que iremos criar posteriormente será feito em Python. Usando dessa linguagem de programação e de uma série de ferramentas criadas a partir dela, como mencionei anteriormente, estaremos desmistificando muita coisa que é dita erroneamente sobre essa área, assim como mostrar na prática que programação voltada a ciência de dados não necessariamente é, ou deve ser, algo extremamente complexo, maçante de difícil entendimento e aplicabilidade.

Caso você seja mais familiarizado com outra linguagem de programação você não terá problemas ao adaptar seus algoritmos para Python, já que aqui estaremos trabalhando com a mesma lógica de programação e você verá que cada ferramenta que utilizaremos ainda terá uma sintaxe própria que indiretamente independe da linguagem abordada. Usaremos da linguagem Python, mas suas bibliotecas possuem ou podem possuir uma sub-sintaxe própria.

Especificamente falando das ferramentas que serão abordadas neste curso, estaremos usando de bibliotecas desenvolvidas e integradas ao Python como, por exemplo, a biblioteca Pandas, uma das melhores e mais usadas no

que diz respeito a análise de dados (leitura, manipulação e visualização dos mesmos), outra ferramenta utilizada será o Numpy, também uma das melhores no que tange a computação científica e uso de dados por meio de operações matemáticas. Também faremos o uso da biblioteca SKLearn que dentro de si possui uma série de ferramentas para processamento de dados já numa sub área que trabalharemos que é a de aprendizagem de máquina.

Outro exemplo é que estaremos usando muito da biblioteca Keras, que fornece ferramentas e funções para a fácil criação de estruturas de redes neurais. Por fim também estaremos fazendo o uso do TensorFlow, biblioteca esta desenvolvida para redes neurais artificiais com grande eficiência e facilidade de uso, além de uma forma mais abstraída (e lógica) de se analisar dados.

Importante apenas complementar que todas ferramentas usadas ao longo desse livro são de distribuição livre, baseadas em políticas opensource, ou seja, de uso, criação, modificação e distribuição livre e gratuita por parte dos usuários. Existem ferramentas proprietárias de grandes empresas, mas aqui vamos nos ater ao que nos é fornecido de forma livre.

Quando e como utilizaremos tais ferramentas?

Se você está começando do zero nesta área fique tranquilo pois a sua enorme complexidade não necessariamente significará ter que escrever centenas de milhares de linhas de código, escritos em linguagem próxima a linguagem de máquina em algo extremamente complexo nem nada do tipo. Basicamente iremos definir que tipo de situação ou problema abordaremos de forma computacional e a partir deste, escolhemos o tipo de ferramenta mais eficiente para aquele fim. Em seguida, usando da linguagem Python dentro de uma IDE ou interpretador, criar e entender passo a passo, linha a linha de código de como estamos abstraindo um problema ou situação real para que, de forma computacional, possamos criar modelos para contextualizar, analisar, aprender com esses dados e criar previsões a partir dos mesmos..

Apenas para fins de exemplo, também, em ciência de dados basicamente trabalharemos com modelos lógicos onde faremos a leitura de dados (em certos casos, grandes volumes de dados), o polimento, a manipulação e a classificação dos mesmos. Também desenvolveremos ferramentas e até mesmo modelos de redes neurais artificiais para que a partir de etapas de treinamento e aprendizagem de máquina elas possam ler, interpretar dados, aprender com seus erros, chegar a resultados satisfatórios de aprendizado e fornecer previsões a partir destes, tudo de forma simples e objetiva.

Um dos exemplos práticos que estaremos abordando em detalhes é a criação e uso de uma rede neural que com base num banco de dados consegue identificar em exames de mamografia a probabilidade de uma determinada lesão identificada a partir do exame se tornar câncer de mama. Raciocine primeiramente que dentro desta área da Radiologia Médica (Diagnóstico por Imagem) existe o profissional técnico que realiza tal tipo de exame, posicionando as estruturas anatômicas a fim de obter imagens da anatomia por meio de radiações ionizantes, um médico especialista que interpreta a imagem com base em sua bagagem de conhecimento de anatomia e patologia e por fim equipamentos analógicos e digitais que produzem imagens da anatomia para fins diagnósticos.

Fazendo o uso de ferramentas de ciência de dados podemos criar um modelo de rede neural que irá aprender a reconhecer padrões de características patológicas a partir de um banco de dados de imagens de exames de raios-x. Dessa forma os profissionais desta área podem ter um viés de confirmação ou até mesmo chegar a taxas de precisão muito maiores em seu diagnóstico, agora possíveis de serem alcançadas por meio destas ferramentas. Vale lembrar que, neste exemplo, assim como em outros, o uso de poder computacional nunca substituirá o(s) profissional(is) em questão, a tecnologia na verdade sempre vem como um grande aliado para que se melhore este e outros processos, no final, todos saem ganhando.

Então, para resumir, se você já está habituado a trabalhar com ferramentas voltadas a programação ou se você está começando do zero, fique tranquilo pois basicamente estaremos usando a combinação de linguagem de programação e seu software interpretador para criar modelos de análise de dados, posteriormente poderemos trabalhar em cima desses modelos e/ou até mesmo distribuí-los para livre utilização de terceiros.

Qual a Abordagem que Será Utilizada?

Se tratando de uma área com tantas subdivisões, cada uma com suas grandes particularidades, poderíamos ir pelo caminho mais longo que seria entender de forma gradual o que é em teoria cada área, como uma se relaciona com outra, como umas são evoluções naturais de outras, etc… Porém, abordaremos o tema da forma mais simples, direta, didática e prática possível. Tenha em mente que devido a nossas limitações, neste caso, de um livro, não teremos como abordar com muita profundidade certos pontos, mas tenha em mente que tudo o que será necessário estará devidamente explicado no decorrer dessas páginas, assim como as referências necessárias para que se busque mais conhecimento caso haja necessidade.

Para se ter uma ideia, poderíamos criar extensos capítulos tentando conceituar programação em Python, inteligência artificial, ciência de dados e aprendizagem de máquina quanto aos tipos de problemas e respectivamente os tipos de soluções que conseguimos de forma computacional.

Poderíamos dividir o tema quanto às técnicas que usamos mais comumente (análise, classificação, regressão, previsão, visão computacional, etc…), ou pelo tipo de rede neural artificial mais eficiente para cada caso (de uma camada, múltiplas camadas, densas (deep learning), convolucionais, recorrentes, etc...), assim como poderíamos separar esse tema de acordo com as ferramentas que são mais usadas por entusiastas, estudantes e profissionais da área (SciKitLearn, Pandas, Numpy, TensorFlow, Keras, OpenCV, etc…).

Ainda poderíamos também fazer uma abordagem de acordo com os métodos aplicados sobre dados (classificação binária, classificação multiclasse, regressão linear, regressão logística, balanço viés-variança, KNN, árvores de decisão, SVMs, clusterização, sistemas de recomendação, processamento de linguagem natural, deep learning, etc…) mas ainda assim seria complicado ao final de todo know hall teórico conseguir fazer as interconexões entre todos conceitos de forma clara.

Sendo assim, a maneira como iremos abordar cada tópico de cada tema de cada nicho será totalmente prático. Usaremos de exemplos reais, de aplicação real, para construir passo a passo, linha a linha de código o

conhecimento necessário para se entender de fato o que é ciência de dados e os métodos de treinamento, aprendizagem de máquina para podermos criar e adaptar nossas próprias ferramentas para estes exemplos e para os próximos desafios que possam surgir.

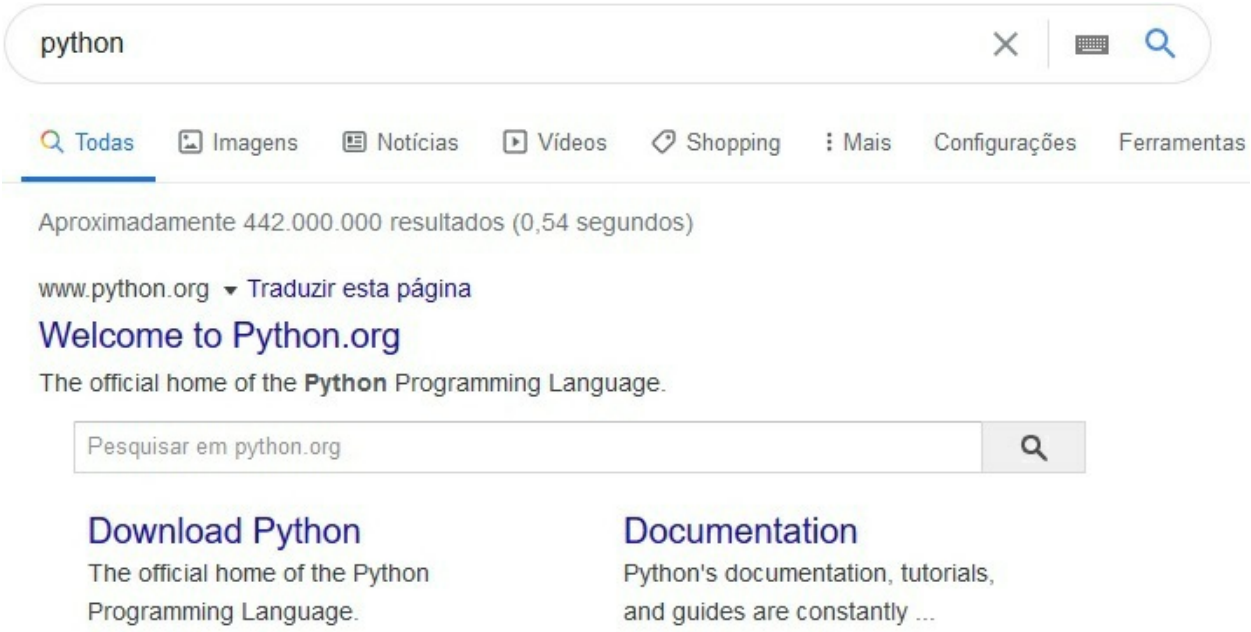
Ao final deste breve livro sobre o assunto você terá bagagem de conhecimento suficiente para criar suas próprias ferramentas de análise de dados e de fato, a partir desses, transformar dados em informação.

PREPARAÇÃO DO AMBIENTE DE TRABALHO

Antes de efetivamente começarmos a implementar nossos códigos, devemos realizar algumas preparações para que possamos avançar para prática. Para nossos devidos fins, estaremos utilizando a linguagem de programação Python em sua versão 3.7, assim como de algumas IDEs inclusas na suíte Anaconda, como o Visual Studio Code e o Spyder. Também faremos isso em um ambiente isolado das demais instalações, para que possamos posteriormente transferir nossos códigos para outros projetos.

Instalação da Linguagem Python

Se você é usuário do Windows, tenha em mente que o núcleo da linguagem Python não vem inclusa no sistema, sendo necessário realizar a instalação da mesma, que por sua vez, é igual a instalação de qualquer software comum em nosso sistema.



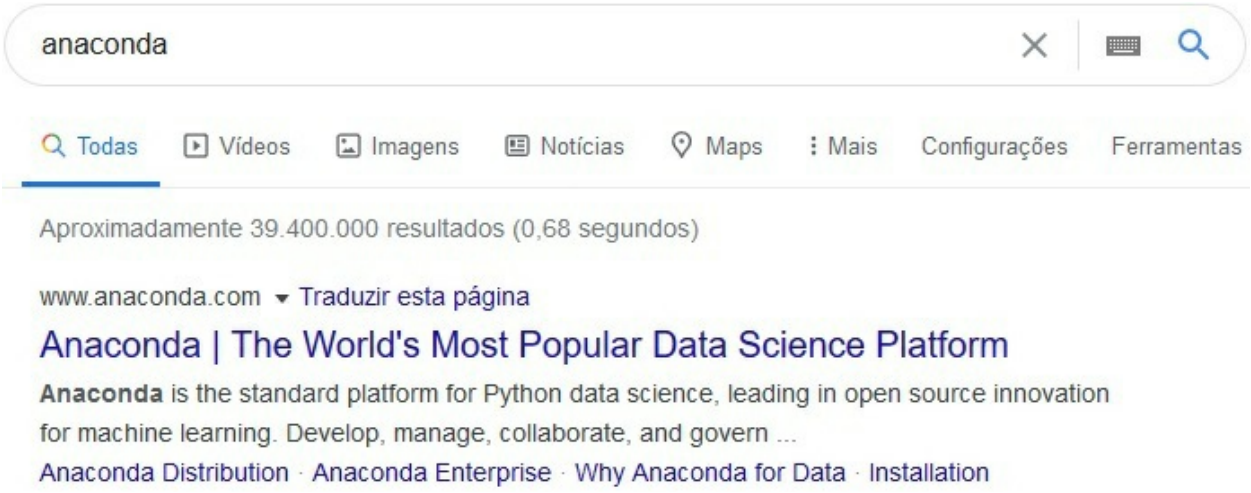
Simplesmente pesquisando no Google por Python, o primeiro resultado instanciado é o do site oficial da linguagem Python. Também acessível via www.python.org



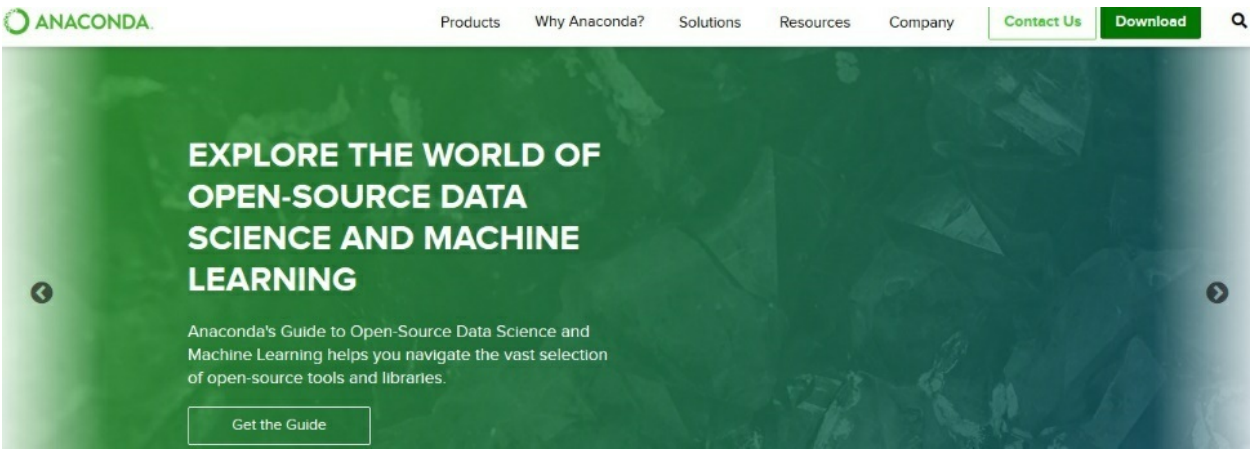
Na página inicial do Python, passando o mouse sob o botão Downloads é possível ter acesso ao link para download da última versão estável, assim como versões passadas caso for necessário.

Instalação da Suite Anaconda

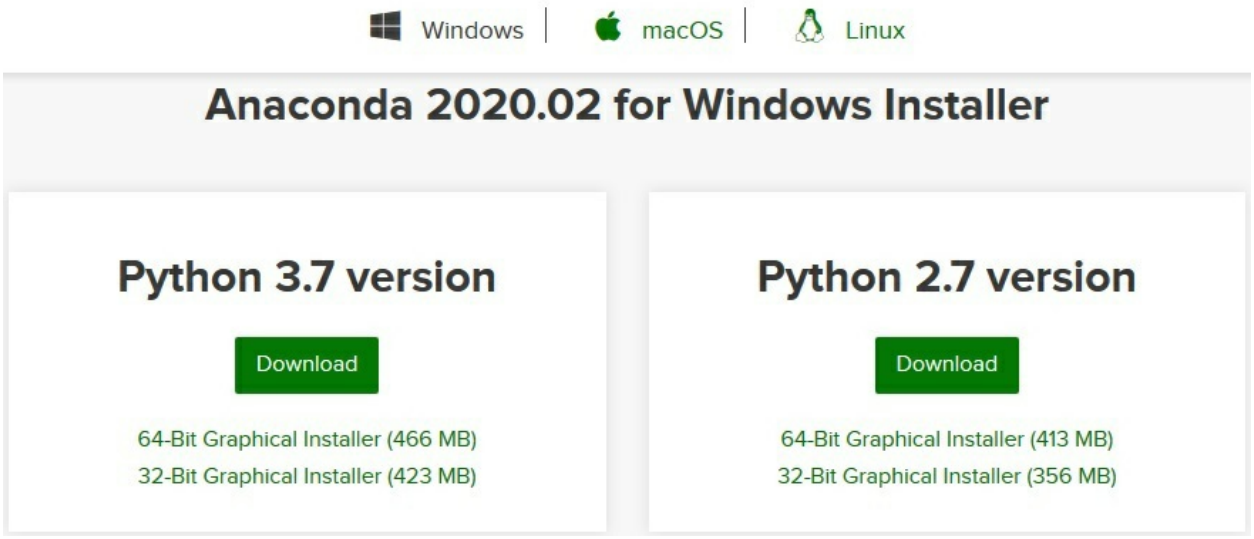
Uma vez instalado o núcleo de nossa linguagem Python, composta pelo núcleo em si, IDE Idle e terminal. Iremos realizar a instalação de um pacote de softwares e ferramentas que nos auxiliarão no que diz respeito a parte de programação.



Da mesma forma, seja pesquisando no Google ou via www.anaconda.com teremos acesso a página oficial dessa suíte de aplicativos.



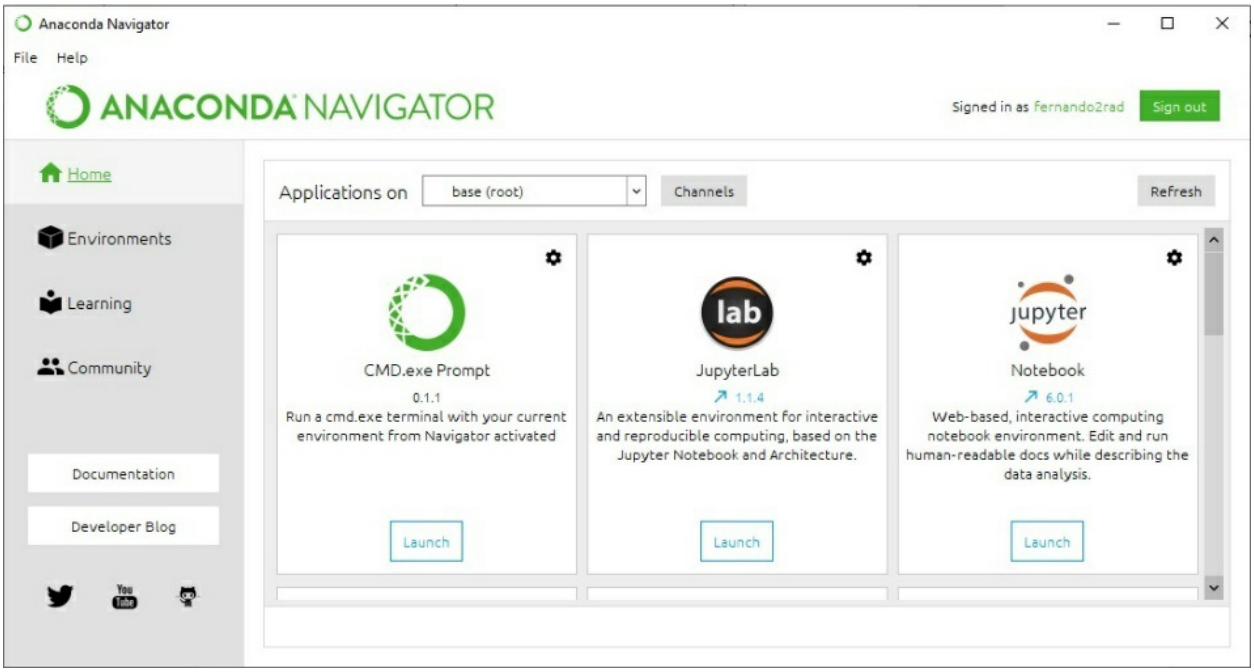
No canto superior direito do site é possível encontrar em destaque o botão que leva para a página de downloads.



Nesta etapa apenas é necessário verificar se você foi redirecionado para a página correspondente ao seu sistema operacional, assim como a versão compatível com a versão da linguagem Python (2.7 ou 3.7) e sua arquitetura (32 ou 64 bits). Após baixada, a instalação da Suite Anaconda se dá da mesma forma como se instala qualquer software no sistema.

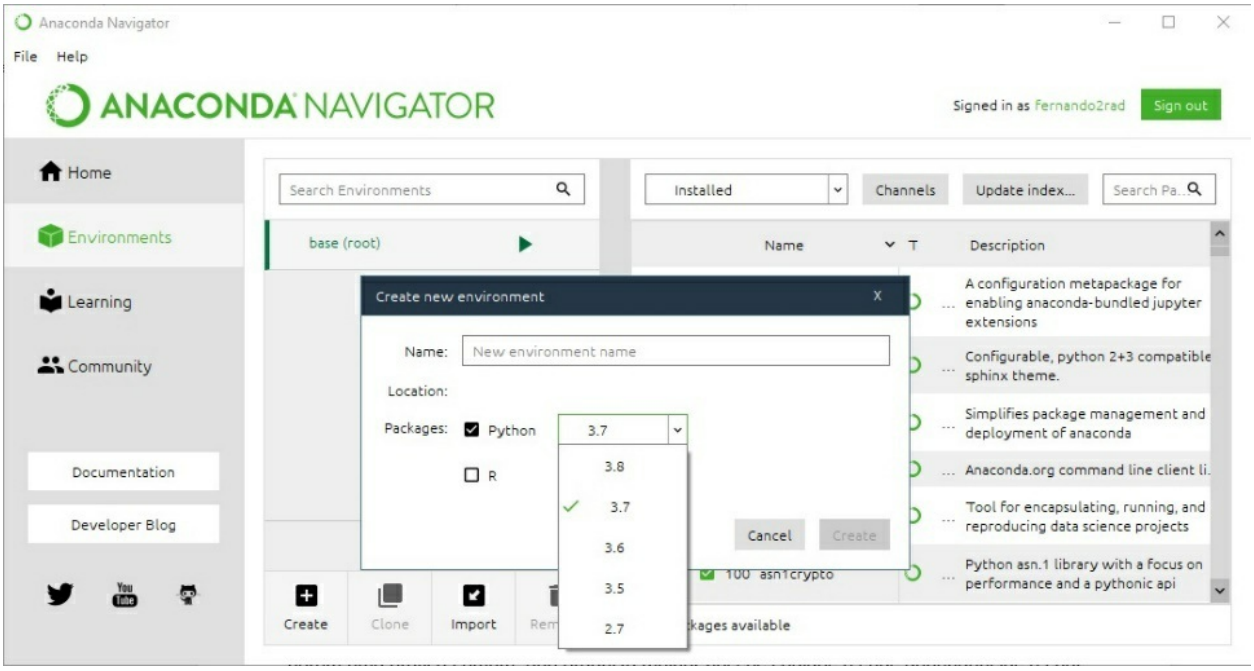
Ambiente de Desenvolvimento

Muito bem, instalada a biblioteca Python assim como a suíte Anaconda, podemos avançar e criar nosso ambiente isolado, onde serão instaladas todas as dependências de nossos códigos. Para isso, inicialmente abrimos o Anaconda e esperamos o término de seu carregamento.

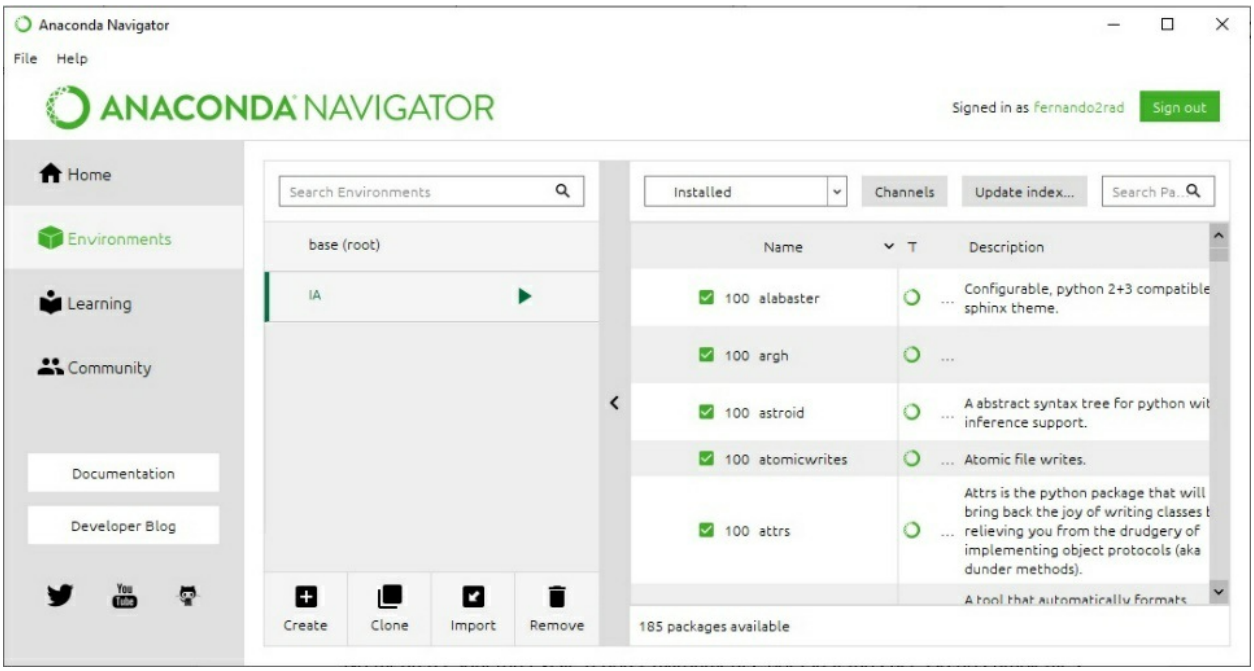


Note que na página inicial da suíte temos um campo chamado Applications on “base(root)”, isso se dá porque até o momento simplesmente instalamos o básico necessário, que por sua vez vem pré-configurado para rodar sob o ambiente em comum com os demais aplicativos do sistema. Não há problema nenhum em programar diretamente sobre esse ambiente, porém uma prática comum, que organiza melhor nossos códigos e suas dependências é criar ambientes virtuais isolados para cada projeto.

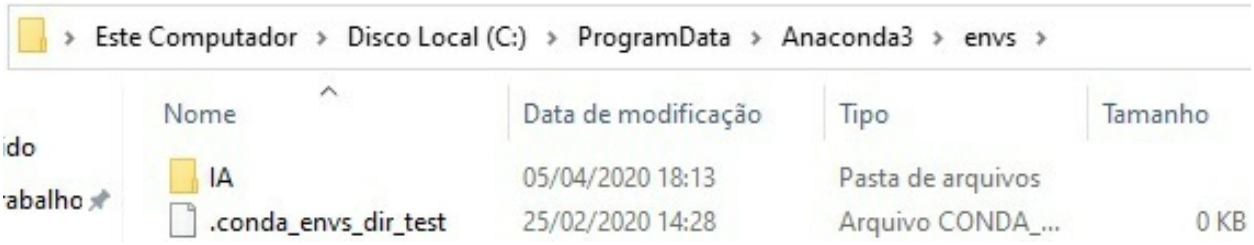
No menu à esquerda existe a aba Environments, por ela temos acesso aos ambientes já criados em nosso sistema e também, de forma intuitiva, o menu para criar novos ambientes a partir daqui.



Logo, clicando no botão Create abrirá uma nova janela pedindo que você dê um nome ao seu ambiente virtual isolado, assim como selecione que linguagem de programação estará usando (nesse caso Python) e por fim que versão da linguagem (nesse caso, 3.7 mesmo).

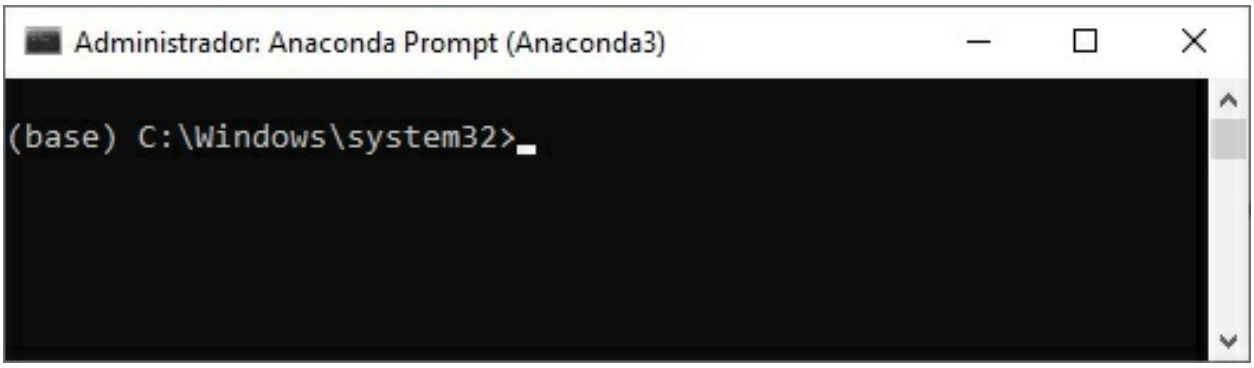


Uma vez criado nosso ambiente virtual isolado, todas demais dependências ficarão salvas em seu diretório, facilitando a transferência desse projeto para outra máquina.

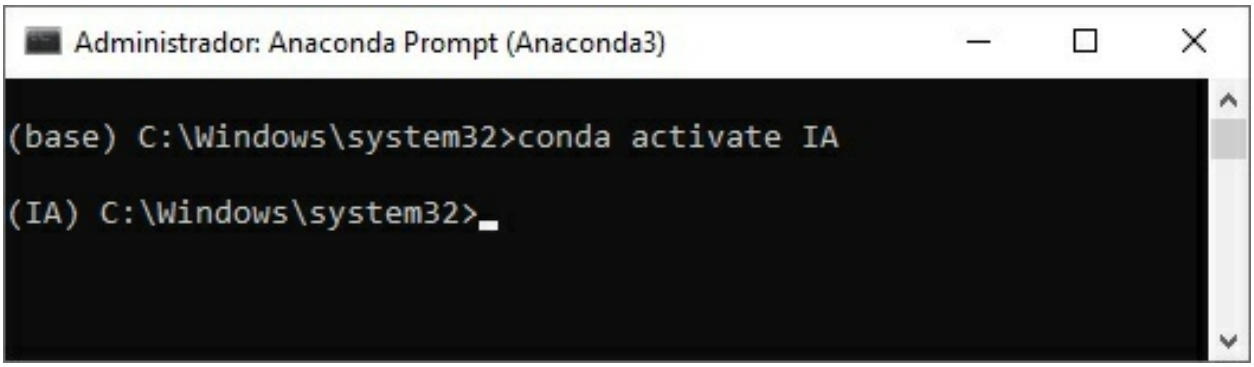


Instalação das Dependências

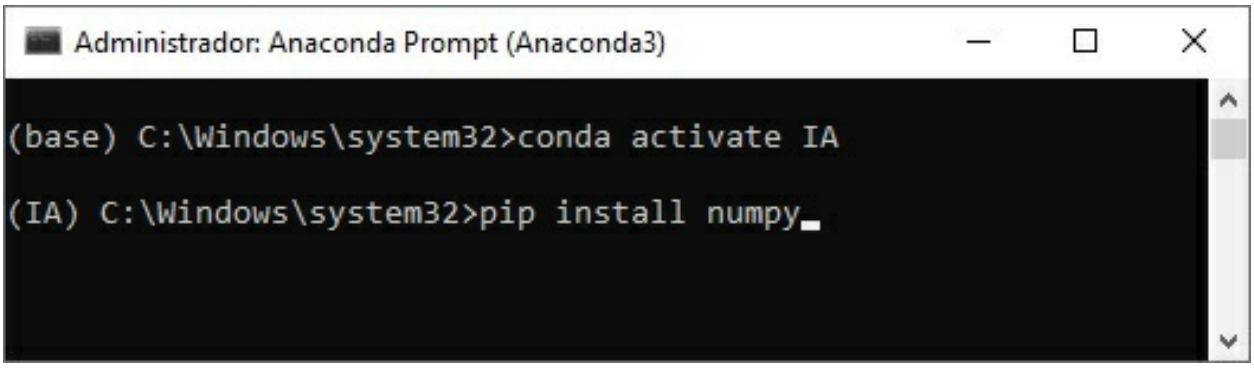
Finalizando a preparação de nosso ambiente, serão necessárias algumas últimas instalações, dessa vez de bibliotecas, módulos e ferramentas que não vem pré-instaladas na linguagem Python. Para isso, inicialmente devemos via prompt de comando ativar o ambiente isolado que criamos e executar alguns códigos básicos.



Note que ao abrir um terminal, o mesmo estará mostrando que todas instâncias estão (como é de se esperar) rodando diretamente no ambiente base do sistema.



Por meio do comando conda activate “nome do ambiente virtual isolado” é feita de fato a ativação da mesma.



E por fim via comando pip install numpy (apenas como exemplo), será baixada a respectiva biblioteca e instalada dentro de nosso ambiente isolado IA, caso não haja nenhum erro durante o processo de download, uma vez terminada a execução do comando pip a devida biblioteca já estará pronta para ser importada e instanciada em nosso código.

Sendo assim, para nossos projetos serão necessárias as respectivas instalações:

pip install numpy

pip install pandas

pip install tensorflow

pip install matplotlib

pip install scikit-learn

pip install scikit-image

pip install keras

pip install kivy

pip install tqdm

pip install pandas_datareader

pip install seaborn

pip install minisom

pip install gym

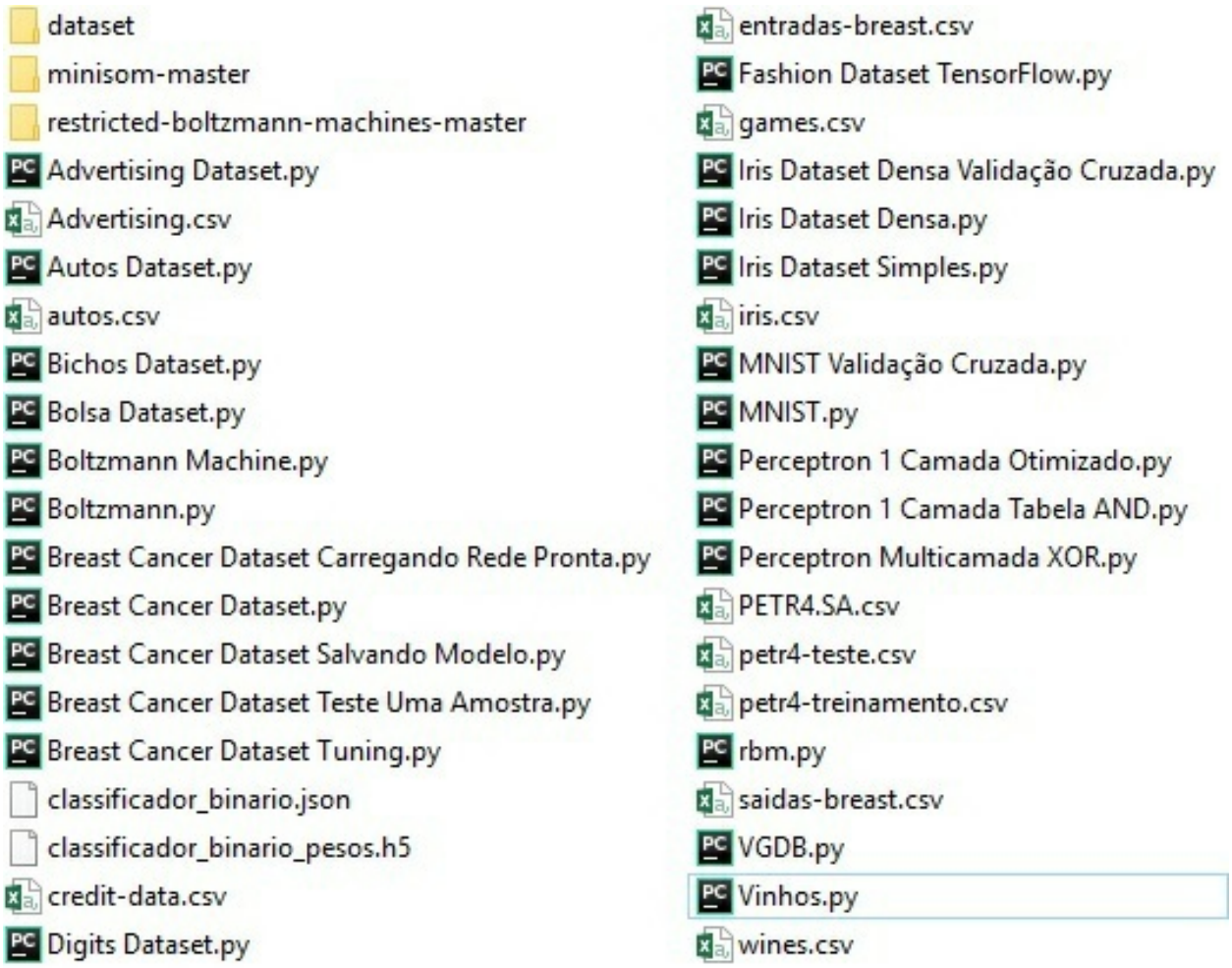
pip install pybullet

Caso você tenha algum problema ao realizar essas importações via CMD uma alternativa é usar o terminal instalador do Anaconda, substituindo pip por conda via CMD você estará instanciando o terminal do Anaconda. Ex: conda install scikit-learn.

Se mesmo assim ocorreram erros no processo de importação, todas essas bibliotecas em seu site possuem seu devido instalador de acordo com o sistema operacional, apenas ficar atento ao instalar manualmente a definição correta dos diretórios.

Todos os demais códigos e ferramentas podem ser baixados diretamente em:

https://drive.google.com/drive/folders/1efxypm6Hh_GcNWQIMltp47OFyqVl
usp=sharing



Ou por meio de meu repositório no GitHub:

https://github.com/fernandofeltrin/Ciencia-de-Dados-e-Aprendizado-de-Maquina

 **fernandofeltrin** Add files via upload

- Advertising Dataset (Previsão Quantitativa).py
- Autos Dataset (Regressão de Dados de Planilias Excel).py
- Bichos Dataset (Classificação de Imagens de Animais).py
- Bolsa Dataset (Previsão de Séries Temporais).py
- Boltzmann Machine (Sistemas de Recomendação).py
- Breast Cancer Dataset (Classificação Binária Via Rede Neural).py
- Breast Cancer Dataset Carregando Rede Pronta.py
- Breast Cancer Dataset Salvando Modelo.py
- Breast Cancer Dataset Teste Uma Amostra.py
- Breast Cancer Dataset Tuning.py
- Digits Dataset (Reconhecimento de Caracteres).py
- Fashion Dataset TensorFlow (Classificação a Partir de Imagens).py
- Iris Dataset Densa (Classificação Multiclasse Via Rede Neural Densa).py
- Iris Dataset Densa Validação Cruzada.py
- Iris Dataset Simples (Classificação Multiclasse Via Rede Neural Simples).py
- MNIST Validação Cruzada.py
- MNIST.py
- Perceptron 1 Camada Otimizado.py
- Perceptron 1 Camada Tabela AND.py
- Perceptron Multicamada XOR.py
- VGDB (Regressão com Múltiplas Saídas).py
- Vinhos (Mapas Auto Organizáveis).py

TEORIA E PRÁTICA EM CIÊNCIA DE DADOS

O que são Redes Neurais Artificiais

Desde os primeiros estudos por parte biológico e psicológico dos mecanismos que temos de aprendizado, descobrindo nossa própria biologia (no sentido literal da palavra), criamos modelos lógicos estruturados buscando abstrair tais mecanismos para que pudéssemos fazer ciência sobre os mesmos.

À medida que a neurociência evoluía, assim como áreas abstratas como psicologia também, pudemos cada vez melhor entender o funcionamento de nosso cérebro, desde os mecanismos físicos e biológicos até os que nos tornam autoconscientes e cientes de como aprendemos a medida que experenciamos esse mundo.

Não por acaso, as primeiras tentativas de abstrairmos tais mecanismos de forma lógico-computacional se deram literalmente, recriando tais estruturas anatômicas e mecanismos de aprendizado de forma artificial, por meio das chamadas redes neurais artificiais, com seu sistema de neuronal sináptico..

Existe uma grande variedade de tipos/modelos de redes neurais artificiais de acordo com sua estrutura e propósito, em comum a todas elas estão por exemplo conceitos de perceptrons, onde literalmente criamos neurônios artificiais e interação entre os mesmos para que possam encontrar padrões em dados, usando de poder computacional como vantagem no que diz respeito ao número de informações a serem processadas e o tempo de processamento muito superior aos limites humanos.

Dessa forma, partindo de preceitos de problemas computacionais a serem resolvidos (ou problemas reais abstraídos de forma computacional), temos modelos de redes neurais artificiais que conseguem ler e interpretar dados a partir de praticamente todo tipo de arquivo (inclusive imagens e sons), realizar classificações para os mais diversos fins, assim como realizar previsões com base em histórico de dados, simular algoritmos genéticos, também realizar processamento de linguagem natural, geração de novos dados a partir de amostras, reconhecimento por conversão, simulação de sentidos biológicos como visão, audição e outros com base em leitura de sensores, aprendizado com base em experiência, tomadas de decisão com

base em memória, etc... Uma verdadeira infinidade de possibilidades.

Raciocine que nesse processo de criarmos arquiteturas de redes neurais artificiais, onde podemos simular praticamente qualquer tipo de inteligência de um ser vivo biológico de menor complexidade. Ainda assim estamos engatinhando nessa área, mesmo com décadas de evolução, porém procure ver à frente, todo o potencial que temos para as próximas décadas à medida que aperfeiçoamos nossos modelos.

Teoria Sobre Redes Neurais Artificiais

Um dos métodos de processamento de dados que iremos usar no decorrer de todos exemplos deste livro, uma vez que sua abordagem será prática, é o de redes neurais artificiais. Como o nome já sugere, estamos criando um modelo computacional para abstrair uma estrutura anatômica real, mais especificamente um ou mais neurônios, células de nosso sistema nervoso central e suas interconexões.

Nosso sistema nervoso, a nível celular, é composto por neurônios e suas conexões chamadas sinapses. Basicamente um neurônio é uma estrutura que a partir de reações bioquímicas que o estimular geram um potencial elétrico para ser transmitido para suas conexões, de forma a criar padrões de conexão entre os mesmos para ativar algum processo de outro subsistema ou tecido ou simplesmente guardar padrões de sinapses no que convencionalmente entendemos como nossa memória.

Na eletrônica usamos de um princípio parecido quando temos componentes eletrônicos que se comunicam em circuitos por meio de cargas de pulso elétrico gerado, convertido e propagado de forma controlada e com diferentes intensidades a fim de desencadear algum processo ou ação.

Na computação digital propriamente dita temos modelos que a nível de linguagem de máquina (ou próxima a ela) realizam chaveamento traduzindo informações de código binário para pulso ou ausência de pulso elétrico sobre portas lógicas para ativação das mesmas dentro de um circuito, independentemente de sua função e robustez.

Sendo assim, não diferentemente de outras áreas, aqui criaremos modelos computacionais apenas como uma forma de abstrair e simplificar a codificação dos mesmos, uma vez que estaremos criando estruturas puramente lógicas onde a partir delas teremos condições de interpretar dados de entrada e saídas de ativação, além de realizar aprendizagem de máquina por meio de reconhecimento de padrões e até mesmo desenvolver mecanismos de interação homem-máquina ou de automação.

A nível de programação, um dos primeiros conceitos que

precisamos entender é o de um Perceptron. Nome este para representar um neurônio artificial, estrutura que trabalhará como modelo lógico para interpretar dados matemáticos de entrada, pesos aplicados sobre os mesmos e funções de ativação para ao final do processo, assim como em um neurônio real, podermos saber se o mesmo é ativado ou não em meio a um processo.

Aproveitando o tópico, vamos introduzir alguns conceitos que serão trabalhados posteriormente de forma prática, mas que por hora não nos aprofundaremos, apenas começaremos a construir aos poucos uma bagagem de conhecimento sobre tais pontos.

Quando estamos falando em redes neurais artificiais, temos de ter de forma clara que este conceito é usado quando estamos falando de toda uma classe de algoritmos usados para encontrar e processar padrões complexos a partir de bases de dados usando estruturas lógicas chamadas perceptrons. Uma rede neural possui uma estrutura básica de entradas, camadas de processamento, funções de ativação e saídas.

De forma geral é composta de neurônios artificiais, estruturas que podem ser alimentadas pelo usuário ou retroalimentadas dentro da rede por dados de suas conexões com outros neurônios, aplicar alguma função sobre esses dados e por fim gerar saídas que por sua vez podem representar uma tomada de decisão, uma classificação, uma ativação ou desativação, etc...

Outro ponto fundamental é entender que assim como eu uma rede neural biológica, em uma rede neural artificial haverão meios de comunicação entre esses neurônios e podemos inclusive usar esses processos, chamados sinapses, para atribuir valores e funções sobre as mesmas. Uma sinapse normalmente possui uma sub estrutura lógica chamada Bias, onde podem ser realizadas alterações intermediárias aos neurônios num processo que posteriormente será parte essencial do processo de aprendizado de máquina já que é nessa “camada” que ocorrem atualizações de valores para aprendizado da rede.

Em meio a neurônios artificiais e suas conexões estaremos os separando virtualmente por camadas de processamento, dependendo da aplicação da rede essa pode ter apenas uma camada, mais de uma camada ou até mesmo camadas sobre camadas (deep learning). Em suma um modelo básico de rede

neural conterá uma camada de entrada, que pode ser alimentada manualmente pelo usuário ou a partir de uma base de dados.

Em alguns modelos haverão o que são chamadas camadas ocultas, que são intermediárias, aplicando funções de ativação ou funcionando como uma espécie de filtros sobre os dados processados. Por fim desde os moldes mais básicos até os mais avançados sempre haverá uma ou mais camadas de saída, que representam o fim do processamento dos dados dentro da rede após aplicação de todas suas funções.

A interação entre camadas de neurônios artificiais normalmente é definida na literatura como o uso de pesos e função soma. Raciocine que a estrutura mais básica terá sempre esta característica, um neurônio de entrada tem um valor atribuído a si mesmo, em sua sinapse, em sua conexão com o neurônio subsequente haverá um peso de valor gerado aleatoriamente e aplicado sobre o valor inicial desse neurônio em forma de multiplicação. O resultado dessa multiplicação, do valor inicial do neurônio pelo seu peso é o valor que irá definir o valor do próximo neurônio, ou em outras palavras, do(s) que estiver(em) na próxima camada. Seguindo esse raciocínio lógico, quando houverem múltiplos neurônios por camada além dessa fase mencionada anteriormente haverá a soma dessas multiplicações.

1 neurônio Z = entradas x pesos

n neurônios Z = (entrada1 x peso1) + (entrada2 + peso2) + etc…

Por fim um dos conceitos introdutórios que é interessante abordarmos agora é o de que mesmo nos modelos mais básicos de redes neurais haverão funções de ativação. Ao final do processamento das camadas é executada uma última função, chamada de função de ativação, onde o valor final poderá definir uma tomada de decisão, uma ativação (ou desativação) ou classificação. Esta etapa pode estar ligada a decisão de problemas lineares (0 ou 1, True ou False, Ligado ou Desligado) ou não lineares (probabilidade de 0 ou probabilidade de 1). Posteriormente ainda trabalharemos com conceitos mais complexos de funções de ativação.

Linear - ReLU (Rectified Linear Units) - Onde se X for maior que 0 é feita a

ativação do neurônio, caso contrário, não. Também é bastante comum se usar a Step Function (Função Degrau) onde é definido um valor como parâmetro e se o valor de X for igual ou maior a esse valor é feita a ativação do neurônio, caso contrário, não.

Não Linear - Sigmoid - Probabilidade de ativação ou de classificação de acordo com a proximidade do valor de X a 0 ou a 1. Apresenta e considera os valores intermediários entre 0 e 1 de forma probabilística.

Também estaremos trabalhando com outras formas de processamento quando estivermos realizando o treinamento supervisionado de nossa rede neural artificial. Raciocine que os tópicos apresentados anteriormente apenas se trata da estrutura lógica ao qual sempre iremos trabalhar, o processo de aprendizado de máquina envolverá mais etapas onde realizaremos diferentes processos em cima de nossa rede para que ela “aprenda” os padrões os quais queremos encontrar em nossos dados e gerar saídas a partir dos mesmos.

O que é Aprendizado de Máquina

Entendendo de forma simples, e ao mesmo tempo separando o joio do trigo, precisamos de fato contextualizar o que é uma rede neural artificial e como se dá o mecanismo de aprendizado de máquina. Raciocine que, como mencionado anteriormente, temos um problema computacional a ser resolvido. Supondo que o mesmo é um problema de classificação, onde temos amostras de dois ou mais tipos de objetos e precisamos os classificar conforme seu tipo.

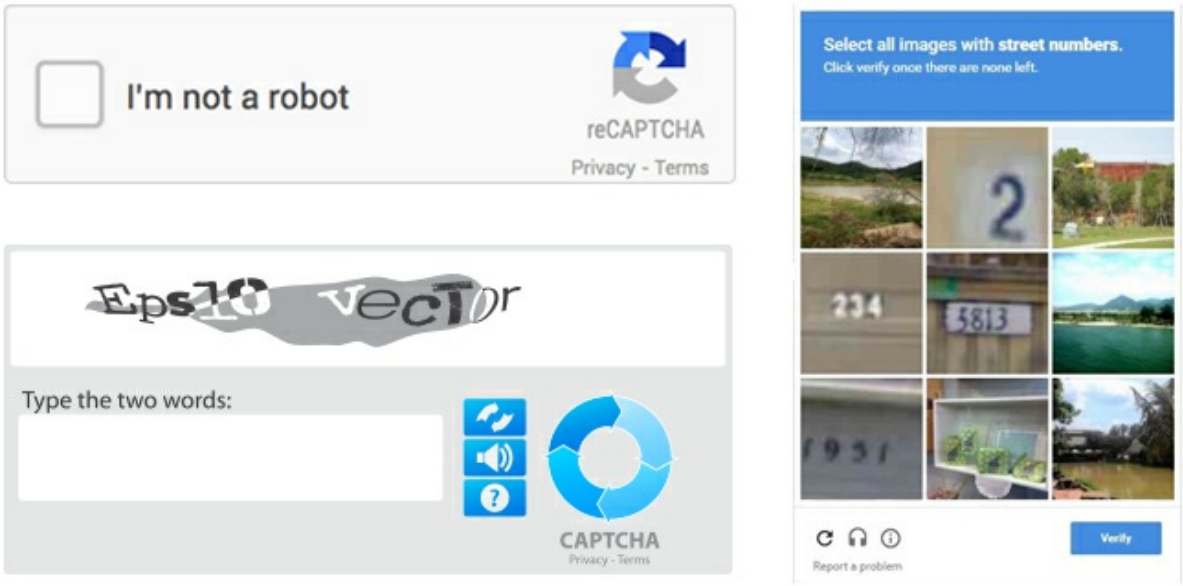
Existe literalmente uma infinidade de algoritmos que poderiam realizar essa tarefa, mas o grande pulo do gato aqui é que por meio de um modelo de rede neural, podemos treinar a mesma para que reconheça quais são os objetos, quais são suas características relevantes e assim treinar e testar a mesma para que com algumas configurações consiga realizar essa tarefa de forma rápida e precisa, além de que uma vez aprendida essa classificação, essa rede neural pode realizar novas classificações com base nos seus padrões aprendidos previamente.

Para esse processo estaremos criando toda uma estrutura de código onde a rede terá dados de entrada, oriundos das amostras a serem classificadas, processamento via camadas de neurônios artificiais encontrando padrões e aplicando funções matemáticas sobre os dados, para ao final do processo separar tais dados das amostras conforme o seu tipo.

Neste processo existe o que é chamado de aprendizado de máquina supervisionado, onde literalmente temos que mostrar o caminho para a rede, mostrar o que é o resultado final correto para que ela aprenda os padrões intermediários que levaram até aquela resolução. Assim como também teremos modelos de aprendizado não supervisionado, onde a rede com base em alguns algoritmos conseguirá sozinha identificar e reconhecer os padrões necessários.

Aprendizado de Máquina

Quando estamos falando em Ciência e Dados, mais especificamente de Aprendizado de Máquina, é interessante começarmos a de fato entender o que essas nomenclaturas significam, para desmistificarmos muita informação errônea que existe a respeito desse termo. Na computação, apesar do nome “máquina”, basicamente estaremos criando modelos lógicos onde se é possível simular uma inteligência computacional, de forma que nosso software possa ser alimentado com dados de entrada e a partir destes, reconhecer padrões, realizar diversos tipos de tarefas sobre estes, aprender padrões de experiência e extrair informações ou desencadear tomadas de decisões a partir dos mesmos. Sendo assim, ao final temos um algoritmo funcional que foi treinado e aprendeu a atingir um objetivo com base nos dados ao qual foi alimentado.



Você pode não ter percebido, mas uma das primeiras mudanças na web quando começamos a ter sites dinâmicos foi o aparecimento de ferramentas de validação como os famosos captcha. Estas aplicações por sua vez tem uma ideia genial por trás, a de que pedindo ao usuário que realize simples testes para provar que ele “não é um robô”, eles está literalmente treinando um robô a reconhecer padrões, criando dessa forma uma espécie de inteligência

artificial. Toda vez que você responde um captcha, identifica ou sinaliza um objeto ou uma pessoa em uma imagem, você está colaborando para o aprendizado de uma máquina.

Quando criamos estes tipos de modelos de algoritmos para análise e interpretação de dados (independe do tipo), inicialmente os fazemos de forma manual, codificando um modelo a ser usado e reutilizado posteriormente para problemas computacionais de mesmo tipo (identificação, classificação, regressão, previsão, etc...).

Tipos de Aprendizado de Máquina

Supervisionado

Basicamente esse é o tipo de aprendizado de máquina mais comum, onde temos um determinado problema computacional a ser resolvido (independentemente do tipo), sabemos de forma lógica como solucionaremos tal problema, logo, treinaremos uma rede neural para que identifique qual a maneira melhor, mais eficiente ou mais correta de atingir o seu objetivo.

Em outras palavras, é o tipo de aprendizado onde teremos de criar e treinar toda uma estrutura de rede neural já lhe mostrando o caminho a ser seguido, a rede neural por sua vez, apenas irá reconhecer quais foram os padrões adotados para que aquele objetivo fosse alcançado com melhor margem de acerto ou precisão. Modelo muito utilizado em problemas de classificação, regressão e previsão a partir de uma base de dados.

Como exemplo, imagine uma rede neural artificial que será treinada para avaliar o perfil de crédito de um correntista de um banco, a fim de conceder ou negar um empréstimo. A rede neural que irá classificar o novo cliente como apto ou não apto a receber crédito, fará essa escolha com base em milhares de outros clientes com as mesmas características, de forma a encontrar padrões de quais são bons ou maus pagadores. Dessa forma, o novo cliente irá fornecer dados que serão filtrados conforme os padrões da base de dados dos clientes antigos para que se possa presumir se ele por sua vez será um bom cliente (se o mesmo possui as mesmas características de todos os outros bons pagadores), obtendo seu direito ao crédito.

Não Supervisionado

Sabendo que é possível mostrar o caminho para uma rede neural, podemos presumir que não mostrar o caminho significa um aprendizado não supervisionado, correto? Não necessariamente! Uma rede neural realizando processamento de forma não supervisionada até pode encontrar padrões, porém sem saber qual o objetivo, a mesma ficará estagnada após algumas camadas de processamento ou terá baixíssima eficiência já que ela não possui

capacidade de deduzir se seus padrões estão certos ou errados para aquele fim.

Porém, esse conceito pode justamente ser utilizado para a resolução de outros tipos de problemas computacionais. Dessa forma, modelos de redes neurais artificiais para agrupamento tendem a funcionar muito bem e de forma não supervisionada. Raciocine que neste tipo de modelo em particular estamos apenas separando amostras conforme suas características, sem um objetivo em comum a não ser o de simplesmente separar tais amostras quanto ao seu tipo.

Como exemplo, imagine que você tem uma caixa com 3 tipos de frutas diferentes, a rede neural irá realizar o mapeamento de todas essas frutas, reconhecer todas suas características e principalmente quais características as fazem diferentes umas das outras para por fim realizar a separação das mesmas quanto ao seu tipo, sem a necessidade de um supervisor.

Por Reforço

Será o tipo de aprendizado onde um determinado agente busca alcançar um determinado objetivo, mas por sua vez, ele aprenderá como alcançar tal objetivo com base na leitura e interpretação do ambiente ao seu redor, interagindo com cada ponto desse ambiente tentando alcançar seu objetivo no que chamamos de tentativa e erro. Toda vez que o mesmo realizar uma ação coerente ou correta dentro daquele contexto, será recompensado de forma que grave aquela informação como algo válido a ser repetido, assim como toda vez que o mesmo realizar uma ação errada para aquele contexto, receberá uma penalidade para que aprenda que aquela forma ou meio usada na tentativa não está correta, devendo ser evitada.

Como exemplo, imagine um carro autônomo, que tem a missão de sair de um ponto A até um ponto B, estacionando o carro, porém o mesmo parte apenas do pressuposto de que pode andar para frente e para trás, assim como dobrar para esquerda ou direita. Nas primeiras tentativas o mesmo irá cometer uma série de erros até que começará a identificar quais ações foram tomadas nas tentativas que deram certo e o mesmo avançou em seu caminho. Em um processo que as vezes, dependendo da complexidade, pode levar

literalmente milhões de tentativas até que se atinja o objetivo, a inteligência artificial irá aprender quais foram os melhores padrões que obtiveram os melhores resultados para replicá-los conforme a necessidade. Uma vez aprendido o caminho nesse ambiente, a IA deste veículo autônomo fará esse processo automaticamente com uma margem de acertos quase perfeita, assim como terá memorizado como foram realizadas suas ações para que possam ser replicadas.

Aprendizagem por reforço natural vs artificial

Enquanto não colocamos em prática esses conceitos, ou ao menos não visualizamos os mesmos em forma de código, tudo pode estar parecendo um tanto quanto abstrato, porém, talvez você já tenha identificado que o que estamos tratando aqui como um tipo de aprendizagem, é justamente o tipo de aprendizagem mais básica e orgânica que indiretamente conhecemos por toda nossa vida.

Todos os mecanismos cognitivos de aprendizado que possuímos em nós é, de certa forma, um algoritmo de aprendizagem por reforço. Raciocine que a maior parte das coisas que aprendemos ao longo de nossa vida foi visualizando e identificando padrões e os imitando, ou instruções que seguimos passo-a-passo e as repetimos inúmeras vezes para criar uma memória cognitiva e muscular sobre tal processo ou com base em nossas experiências, diretamente ou indiretamente em cima de tentativa e erro.

Apenas exemplificando de forma simples, quando aprendemos um instrumento musical, temos uma abordagem teórica e uma prática. A teórica é onde associamos determinados conceitos de forma lógica em sinapses nervosas, enquanto a prática aprendemos que ações realizamos e as repetimos por diversas vezes, ao final de muito estudo e prática dominamos um determinado instrumento musical em sua teoria (como extrair diferentes sons do mesmo formando frases de uma música) e em sua prática (onde e como tocar para extrair tais sons).

Outro exemplo bastante simples de nosso cotidiano, agora focando no sistema de recompensa e penalidade, é quando temos algum animal de estimação e o treinamos para realizar determinadas tarefas simples. Dentre a simplicidade cognitiva de um cachorro podemos por exemplo, ensinar o

mesmo que a um determinado comando de voz ou gesto ele se sente, para isto, lhe damos como recompensa algum biscoitinho ou simplesmente um carinho, indiretamente o mesmo está associando que quando ele ouvir aquele comando de voz deve se sentar, já que assim ele foi recompensado. Da mesma forma, quando ele faz uma ação indesejada o chamamos a atenção para que associe que não é para repetir aquela ação naquele contexto.

Sendo assim, podemos presumir que esta mesma lógica de aprendizado pode ser contextualizada para diversos fins, inclusive, treinando uma máquina a agir e se comportar de determinadas maneiras conforme seu contexto, aplicação ou finalidade, simulando de forma artificial (simples, porém convincente) uma inteligência.

No futuro, talvez em um futuro próximo, teremos desenvolvido e evoluído nossos modelos de redes neurais artificiais intuitivas de forma a não somente simular com perfeição os mecanismos de inteligência quanto os de comportamento humano, tornando assim uma máquina indistinguível de um ser humano em sua aplicação. Vide teste de Turing.

Aprendizado por Reforço em Detalhes

Avançando um pouco em nossos conceitos, se entendidos os tópicos anteriores, podemos começar a nos focar no que será nosso objetivo ao final deste livro, os tipos de redes neurais artificiais que de uma forma mais elaborada simularão uma inteligência artificial.

Ainda falando diretamente sobre modelos de redes neurais artificiais, temos que começar a entender que, como dito anteriormente, existirão modelos específicos para cada tipo de problema computacional, e repare que até agora, os modelos citados conseguem ser modelados de forma parecida a uma rede neural biológica, para abstrair problemas reais de forma computacional, realizando processamento supervisionado ou não, para que se encontrem padrões que podem gerar diferentes tipos de saídas conforma o problema em questão. Porém, note que tudo o que foi falado até então, conota um modelo de rede neural artificial estático, no sentido de que você cria o modelo, o alimenta, treina e testa e gera previsões sobre si mas ao final do processo ele é encerrado assim como qualquer outro tipo de programa.

Já no âmbito da inteligência artificial de verdade, estaremos vendo modelos de redes neurais que irão além disso, uma vez que além de todas características herdadas dos outros modelos, as mesmas terão execução contínua inclusive com tomadas de decisões.

Os modelos de redes neurais artificiais intuitivas serão aqueles que, em sua fase de aprendizado de máquina, estarão simulando uma memória de curta ou longa duração (igual nossa memória como pessoa), assim como estarão em tempo real realizando a leitura do ambiente ao qual estão inseridos, usando esses dados como dados de entrada que retroalimentam a rede neural continuamente, e por fim, com base no chamado aprendizado por reforço, a mesma será capaz de tomar decisões arbitrárias simulando de fato uma inteligência artificial.

Diferentemente de um modelo de rede neural artificial “tradicional” onde a fase supervisionada se dá por literalmente programar do início ao fim de um processo, uma rede neural artificial “intuitiva” funciona de forma autosupervisionada, onde ela realizará determinadas ações e, quando a

experiência for positiva, a mesma será recompensada, assim como quando a experiência for negativa, penalizada.

Existirão, é claro, uma série de algoritmos que veremos na sequência onde estaremos criando os mecanismos de recompensa e penalização, e a forma como a rede neural aprende diante dessas circunstâncias. Por hora, raciocine que nossa rede neural, em sua inteligência simulada, será assim como um bebê descobrindo o mundo, com muita tentativa, erro e observações, naturalmente a mesma aprenderá de forma contínua e com base em sua experiência será capaz de tomar decisões cada vez mais corretas para o seu propósito final.

Apenas realizando um adendo, não confundir esse modelo de rede neural artificial com os chamados algoritmos genéticos, estes por sua vez, adotam o conceito de evolução baseada em gerações de processamento, mas não trabalham de forma autônoma e contínua. Veremos o referencial teórico para esse modelo com mais detalhes nos capítulos subsequentes.

Características Específicas do Aprendizado por Reforço

À medida que vamos construindo nossa bagagem de conhecimento sobre o assunto, vamos cada vez mais contextualizando ou direcionando nosso foco para as redes neurais artificiais intuitivas. Nesse processo será fundamental começarmos a entender outros conceitos das mesmas, entre eles, o de aprendizado por reforço utilizado neste tipo de rede neural.

Como mencionado anteriormente de forma rápida, o chamado aprendizado por reforço implicitamente é a abstração da forma cognitiva de como nós seres vivos aprendemos de acordo com a forma com que experenciamos esse mundo. Por meio de nossos sentidos enxergamos, ouvimos, saboreamos e tocamos deferentes objetos e criando memórias de como interagimos e que experiência tivemos com os mesmos.

Da mesma forma, porém de forma artificial, teremos um agente que simulará um ser vivo ou ao menos um objeto inteligente que por sua vez, via sensores (equivalendo aos seus sentidos), ele fará a leitura do ambiente e com o chamado aprendizado por reforço, experenciará o mesmo realizando ações com os objetos e aprendendo com base em tentativa erro ou acerto.

Para isso, existirão uma série de algoritmos que veremos nos capítulos subsequentes em maiores detalhes que literalmente simularão como esse agente usará de seus recursos de forma inteligente. Desde o mapeamento do ambiente, dos estados e das possíveis ações até a realização das mesmas em busca de objetivos a serem cumpridos gerando recompensas ou penalidades, por fim guardando os melhores estados em uma memória.

Então, apenas definindo algumas características deste tipo de algoritmo, podemos destacar:

- A partir do seu start, toda execução posterior é autônoma, podendo ser em tempo real via backpropagation ou em alguns casos via aprendizado por gerações quando numa tarefa repetitiva.

- Pode ter ou não um objetivo definido, como ir de um ponto A até o B ou aprender a se movimentar dentro de um ambiente fechado, respectivamente.

- Realiza leitura de sensores que simulam sentidos, mapeando o ambiente ao seu redor e o seu estado atual (que tipo de ação está realizando).

- Busca alcançar objetivos da forma mais direta, de menor tempo de execução ou mais eficiente, trabalhando com base em sua memória de experiência, estado e recompensas.

- Usa de redes neurais realizando processamento retroalimentado em tempo real, convertendo as saídas encontradas em tomadas de decisão.

- Possui um tempo considerável de processamento até gerar suas memórias de curta e longa duração, dependendo da complexidade de sua aplicação, precisando que uma determinada ação seja repetida milhares ou milhões de vezes repetidamente até que se encontre e aprenda o padrão correto.

Principais Algoritmos para Inteligência Artificial

Quando estamos falando em ciência de dados, aprendizado de máquina, rede neurais artificias, etc... existe uma grande gama de nomenclatura que em suma faz parte da mesma área/nicho da computação, porém com importantes características que as diferenciam entre si que precisamos entender para acabar separando as coisas.

Machine learning é a nomenclatura usual da área da computação onde se criam algoritmos de redes neurais artificiais abstraindo problemas reais em computacionais, de forma que a "máquina" consiga encontrar padrões e aprender a partir destes. De forma bastante reduzia, podemos entender que as redes neurais artificiais são estruturas de processamento de dados onde processamos amostras como dados de entrada, a rede neural reconhece padrões e aplica funções sobre esses dados, por fim gerando saídas em forma de classificação, previsão, etc..

Para alguns autores e cientistas da área, a chamada data science (ciência de dados em tradução livre) é o nicho onde, com ajuda de redes neurais artificiais, conseguimos processar enormes volumes de dados estatísticos quantitativos ou qualitativos de forma a se extrair informação dos mesmos para se realizem estudos retroativos ou prospectivos. Porém é importante entender que independentemente do tipo de problema computacional, teremos diferentes modelos ou aplicações de redes neurais para que façam o trabalho pesado por nós.

Sendo assim, não é incorreto dizer que toda aplicação que use deste tipo de arquitetura de dados como ferramenta de processamento, é de certa forma inteligência artificial simulada.

Partindo do pressuposto que temos diferentes tipos de redes neurais, moldáveis a praticamente qualquer situação, podemos entender rapidamente os exemplos mais comumente usados.

- Redes neurais artificiais – Modelo de processamento de dados via perceptrons (estrutura de neurônios interconectados que processam dados de

entrada gerando saídas) estruturadas para problemas de classificação, regressão e previsão.

- Redes neurais artificiais profundas (deep learning) – Estruturas de rede neural com as mesmas características do exemplo anterior, porém mais robustas, com maior quantidade de neurônios assim como maior número de camadas de neurônios interconectados entre si ativos realizando processamento de dados.

- Redes neurais artificiais convolucionais – Modelo de rede neural com capacidade de usar de imagens como dados de entrada, convertendo as informações de cada pixel da matriz da imagem para arrays (matrizes numéricas), realizando processamento sobre este tipo de dado.

- Redes neurais artificiais recorrentes – Modelos de rede neurais adaptados a realizar o processamento de dados correlacionados com uma série temporal, para que a partir dos mesmos se realizem previsões.

- Redes neurais artificiais para mapas auto-organizáveis – Modelo de rede neural com funcionamento não supervisionado e com processamento via algoritmos de aglomeração, interpretando amostras e classificando as mesmas quanto sua similaridade.

- Redes neurais artificiais em Boltzmann Machines – Modelo não sequencial (sem entradas e saídas definidas) que processa leitura dos perceptrons reforçando conexões entre os mesmos de acordo com sua similaridade, porém podendo também ser adaptadas para sistemas de recomendação

- Redes neurais artificiais com algoritmos genéticos – Modelo baseado em estudo de gerações de processamento sequencial, simulando o mecanismo de evolução reforçando os melhores dados e descartando os piores geração após

geração de processamento.

- Redes neurais artificiais adversariais generativas – Modelo capaz de identificar características de similaridade a partir de dados de entrada de imagens, sendo capaz de a partir desses dados gerar novas imagens a partir do zero com as mesmas características.

- Redes neurais artificiais para processamento de linguagem natural – Modelo de nome autoexplicativo, onde a rede tem capacidade de encontrar padrões de linguagem seja ela falada ou escrita, aprendendo a linguagem.

- Redes neurais artificiais intuitivas – Modelo autônomo que via sensores é capaz de realizar a leitura do ambiente ao seu entorno, interagir com o mesmo usando os resultados de tentativa e erro como aprendizado e a partir disso tomar decisões por conta própria.

- Redes neurais artificiais baseadas em Augumented Random Search (ARS) – Modelo que pode ser construído em paralelo a uma rede neural artificial intuitiva, seu foco se dá em criar mecanismos de aprendizado por reforço para um Agente que possui um corpo, dessa forma uma rede neural realizará aprendizado por reforço para o controle de seu corpo, enquanto outra realizará aprendizado por reforço para seu aprendizado lógico convencional.

Estes modelos citados acima são apenas uma pequena amostra do que se existe de redes neurais artificiais quando as classificamos quanto ao seu propósito. Raciocine que uma rede neural pode ser adaptada para resolução de praticamente qualquer problema computacional, e nesse sentido são infinitas as possibilidades. Outro ponto a destacar também é que é perfeitamente normal alguns tipos/modelos de redes neurais artificiais herdarem características de outros modelos, desde que o propósito final seja atendido.

Adendo – Augumented Random Search

O algoritmo Augmented Random Search, idealizado por Horia Mania e Aurelia Guy, ambos da universidade de Berkeley, é um modelo de estrutura de código voltada para inteligência artificial, principalmente no âmbito de abstrair certas estruturas de nosso Agente tornando-o um simulacro de ser humano ou de algum animal.

Diferente de outros modelos onde nosso agente é uma forma genérica com alguns sensores, por meio do ARS temos, entre outros modelos, o chamado MiJoCo (Multi-joint dynamics with contact), que por sua vez tenta simular para nosso agente uma anatomia com todas suas características fundamentais, como tamanho e proporção, articulações, integração com a física simulada para o ambiente, etc...

Se você já está familiarizado com os conceitos de Agente vistos nos capítulos anteriores, deve lembrar que o mesmo, via rede neural artificial intuitiva, trabalha com estados e ações, partindo do princípio de sua interação com o ambiente. Pois bem, quando estamos trabalhando com ARS estamos dando um passo além pois aqui o próprio corpo de nosso agente passa a ser um ambiente. Raciocine que o mesmo deverá mapear, entender e aprender sobre o seu corpo antes mesmo de explorar o ambiente ao seu entorno.

O conceito de nosso Agente se manterá da mesma forma como conhecemos, com sua tarefa de ir de um ponto A até um ponto B ou realizar uma determinada tarefa, realizando leituras do ambiente, testando possibilidades, aprendendo com recompensas e penalidades, etc... nos moldes de aprendizado por reforço. Porém dessa vez, de uma forma mais complexa, terá de realizar suas determinadas tarefas controlando um corpo.

Raciocine, apenas como exemplo, um Agente que deve explorar um ambiente e em um determinado ponto o mesmo deve subir um lance de escadas, raciocine que o processo que envolve o mesmo aprender o que é um lance de escadas e como progredir sobre esse “obstáculo” é feito considerando o que são suas pernas, como deverá mover suas pernas, qual

movimento, em qual direção, em que período de tempo, com contração de quais grupos musculares, etc...

Se você já jogou algum jogo recente da série GTA, deve lembrar que os NPCs são dotados de uma tecnologia chamada Ragdoll que simula seu corpo e a física sobre o mesmo de forma realista, reagindo a qualquer estímulo do ambiente sobre si, aqui o conceito será semelhante, porém nosso Agente terá um corpo e uma inteligência, ao contrário do jogo onde há um corpo e um simples script de que ações o NPC fica realizando aleatoriamente para naquele cenário simplesmente parecer uma pessoa vivendo sua vida normalmente.

Note que novamente estaremos abstraindo algo de nossa própria natureza, todo animal em seus primeiros dias/meses de vida, à medida que descobre o seu próprio corpo e explora o ambiente, aprende sobre o mesmo. Ao tentar desenvolver uma habilidade nova, como por exemplo tocar um instrumento musical, parte do processo é entender a lógica envolvida e parte do processo é saber o que e como fazer com seu corpo.

Se você um dia, apenas como exemplo, fez aulas de violão, certamente se lembrará que para extrair diferentes sons do instrumento existe uma enorme variedade de posições dos seus dedos nas casas do violão para que se contraiam as cordas do mesmo para ressoar uma determinada frequência. Entender o que é um ré menor é lógica, a posição dos dedos nas cordas é física, a prática de repetição feita nesse caso é para desenvolver uma memória muscular, onde sempre que você for tocar um ré menor, já saiba como e onde posicionar seus dedos para essa nota.

Ao programar este tipo de característica para um Agente, não teremos como de costume um algoritmo conceituando cada ação a ser realizada e em que ordem a mesma deve ser executada, mas fazendo o uso de q-learning, aprenderá o que deve ser feito em tentativa e erro fazendo uso de aprendizado por reforço.

Apenas concluindo esta linha de raciocínio, lembre-se que em nossos modelos de inteligência artificial tínhamos o conceito de continuidade, onde nosso Agente sempre estava em busca de algo a se fazer, uma vez que o “cérebro” do mesmo não para nunca seu processamento. Agora iremos somar a este modelo muito de física vetorial, pois estaremos ensinando o controle e movimento do corpo de nosso Agente em um

ambiente simulado de 2 ou 3 dimensões, dependendo o caso.

Método de Diferenças Finitas

Um ponto que deve ser entendido neste momento é o chamado Método de Diferenças Finitas, pois esse trata algumas particularidades que diferenciará o ARS de outros modelos lógicos de inteligência artificial.

Como de costume, para nossos modelos voltados a inteligência artificial, temos aquele conceito básico onde nosso Agente realizará uma determinada ação por tentativa e erro, recebendo recompensas quando as ações foram dadas como corretas e penalidades quando executadas determinadas ações que diretamente ou indiretamente acabaram o distanciando de alcançar seu objetivo. Lembre-se que nesse processo, conforme nosso agente, via aprendizado por reforço, aprendia o jeito certo de realizar determinada tarefa, a medida que os padrões dos pesos eram reajustados para guardar os melhores padrões, a arquitetura do código era desenvolvida de forma a aos poucos descartar totalmente os processos das tentativas que deram errado.

O grande problema nisso é que, ao contextualizar este tipo de conceito para algoritmos em ARS, os mesmos irão contra a própria lógica e proposta de ARS. Vamos entender melhor, em uma rede neural artificial intuitiva comum, onde nosso Agente é genérico e possui apenas sensores para ler, mapear e explorar o ambiente, e um objetivo de, por exemplo, se deslocar de um ponto A até um ponto B, o mesmo simplesmente iria considerando qual ação de deslocamento o deixou mais afastado do ponto A e mais próximo do ponto B, sem a necessidade de considerar nada mais além disso. Agora raciocine que ao considerar um corpo para nosso Agente, o mesmo inicialmente iria entender o que são suas pernas, qual tipo de movimento é possível realizar com elas, e começar a fazer o uso das mesmas em seu deslocamento, porém, a cada tentativa dada como errada, a nível dos ajustes dos pesos em busca dos padrões corretos o mesmo iria “desaprender” um pouco a caminhar a cada erro, tendo que reaprender a andar de forma compensatória em busca do acerto.

Sendo assim, nossa estrutura de código deverá levar em consideração esse problema e ser adaptada de forma que nosso Agente não tenha de aprender a caminhar novamente a cada processo. Teremos de separar as coisas no sentido de que, aprender a caminhar e caminhar de forma correta deve ser aprendida e não mais esquecida, e por parte de lógica, aprender quais ações foram tomada de forma errada deverão ser compensadas por ações tomadas de forma correta, sem interferir no aprendizado de caminhar.

Novamente estamos abstraindo coisas de nossa própria natureza, tentando simular nossos mecanismos internos que nos fazem animais racionais de forma simples e dentro de uma série de diversas limitações computacionais. Apenas tenha em mente que, por exemplo, você aprendeu a digitar textos em algum momento de sua vida, todas as funções físicas e cognitivas que você usa ao digitar não precisam ser reaprendidas caso você digite algo errado, mas simplesmente deve continuar usando a mesma habilidade para corrigir o erro lógico encontrado.

Dessa forma, naquele processo de retroalimentação das redes neurais utilizadas no processo, teremos algumas particularidades para que não se percam informações relativas a estrutura de nosso Agente e ao mesmo tempo sejam corrigidas apenas as variáveis que estão diretamente ligadas a sua lógica. É como se houvesse aprendizado simultâneo para “corpo e mente” de nosso Agente, porém nenhum aprendizado referente ao seu corpo pode ser perdido enquanto todo e qualquer aprendizado para sua mente pode sofrer alterações.

Em uma rede neural artificial intuitiva convencional o aprendizado é/pode ser feito ao final de cada ciclo, ao final de cada ação, agora, para conseguirmos separar os aprendizados de corpo e de mente de nosso Agente, esse processo é realizado ao final de todo o processo, tendo informações de todos os ciclos envolvidos desde o gatilho inicial até a conclusão da tarefa, de modo que possamos separar o que deu certo e errado no processo de aprendizado apenas por parte da mente de nosso Agente.

Se você entendeu a lógica até aqui, deve ter percebido que internamente não teremos mais aquele conceito clássico de descida do gradiente pois diferente de uma rede neural convencional, aqui teremos estados específicos para o corpo de nosso Agente e dinâmicos para sua mente. Ainda no exemplo de caminhar de um ponto A até um ponto B, o processo de caminhar sempre

será o mesmo, para onde, de que forma, em que sentido, em qual velocidade será reaprendido a cada ciclo. Ao final dessa tarefa inicial, caso delegamos que nosso Agente agora caminhe de um ponto B até um ponto C, o mesmo não terá de aprender a caminhar novamente, mas se focará apenas em caminhar de encontro ao ponto C.

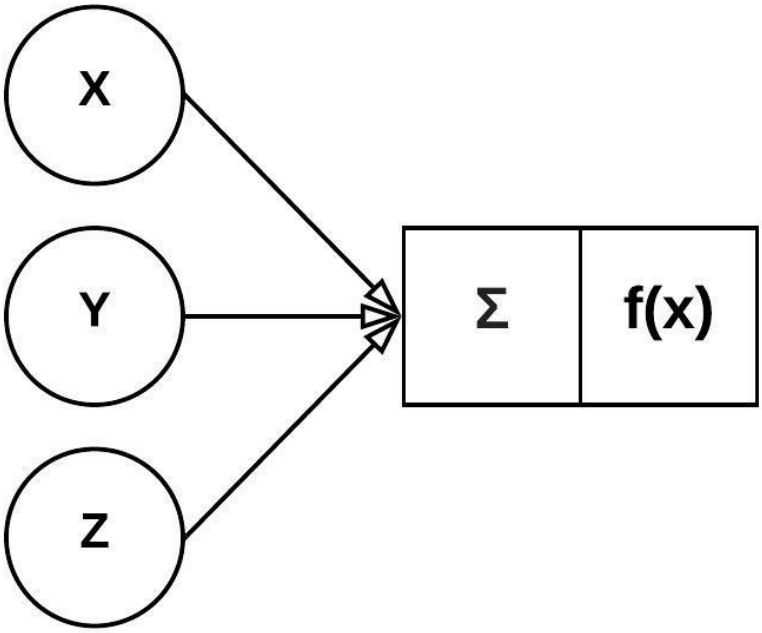
Finalizando nossa linha de raciocínio, agora o nome Augumented Random Search deve fazer mais sentido, uma vez que os estados e ações de nosso Agente serão realizados por uma pesquisa aleatória da interação de nosso Agente com o ambiente ao seu entorno, de forma extracorpórea, ou seja, a simulação de sua inteligência artificial em sua realidade aumentada leva a consideração de um corpo e o constante aprendizado de sua mente.

Perceptrons

Para finalmente darmos início a codificação de tais conceitos, começaremos com a devida codificação de um perceptron de uma camada, estrutura mais básica que teremos e que ao mesmo tempo será nosso ponto inicial sempre que começarmos a programar uma rede neural artificial para problemas de menor complexidade, chamados linearmente separáveis.

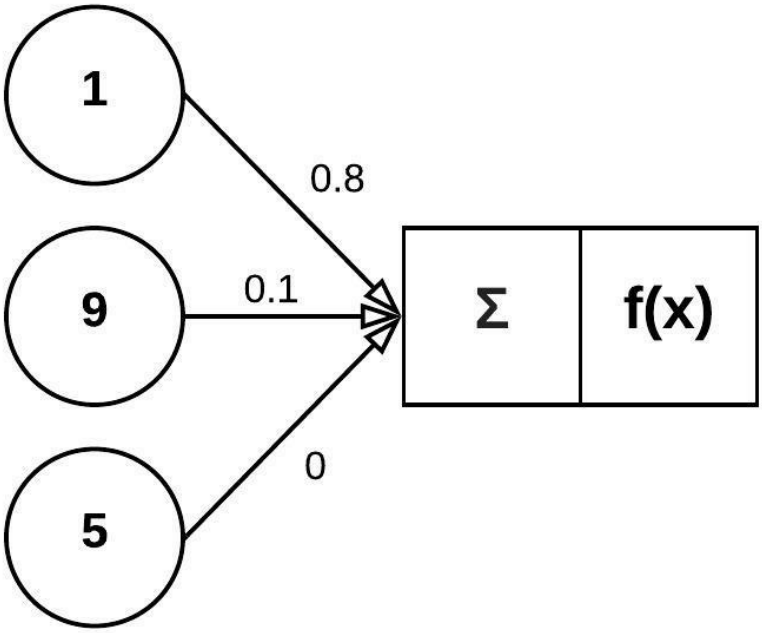
O modelo mais básico usado na literatura para fins didáticos é o modelo abaixo, onde temos a representação de 3 espaços alocados para entradas (X, Y e Z), note que estas 3 figuras se conectam com uma estrutura central com o símbolo Sigma Σ, normalmente atribuído a uma Função Soma, operação aritmética bastante comum e por fim, esta se comunica com uma última estrutura lógica onde há o símbolo de uma Função de Ativação f(x).

Em suma, estaremos sempre trabalhando com no mínimo essas três estruturas lógicas, haverão modelos muito mais complexos ou realmente diferentes em sua estrutura, mas por hora o que deve ficar bem entendido é que todo perceptron terá entradas, operações realizadas sobre estas e uma função que resultará na ativação ou não deste neurônio em seu contexto.



A partir deste modelo podemos resolver de forma computacional os chamados problemas linearmente separáveis. De forma bastante geral podemos entender tal conceito como o tipo de problema computacional onde se resulta apenas um valor a ser usado para uma tomada de decisão. Imagine que a partir desse modelo podemos criar uma pequena rede neural que pode processar dados para que se ative ou não um neurônio, podemos abstrair essa condição também como fazíamos com operadores lógicos, onde uma tomada de decisão resultava em 0 ou 1, True ou False, return x ou return y, etc… tomadas de decisão onde o que importa é uma opção ou outra.

Partindo para prática, agora atribuindo valores e funções para essa estrutura, podemos finalmente começar a entender de forma objetiva o processamento da mesma:



Sendo assim, usando o modelo acima, aplicamos valores aos nós da camada de entrada, pesos atribuídos a elas, uma função de soma e uma função de ativação. Aqui, por hora, estamos atribuindo manualmente esses valores de pesos apenas para exemplo, em uma aplicação real eles podem ser iniciados zerados, com valores aleatórios gerados automaticamente ou com valores temporários a serem modificados no processo de aprendizagem.

A maneira que faremos a interpretação inicial desse perceptron, passo-a-passo, é da seguinte forma:

Temos três entradas com valores 1, 9 e 5, e seus respectivos pesos 0.8, 0.1 e 0.

Inicialmente podemos considerar que o primeiro neurônio tem um valor baixo mas um impacto relativo devido ao seu peso, da mesma forma o segundo neurônio de entrada possui um valor alto, 9, porém de acordo com seu peso ele gera menor impacto sobre a função, por fim o terceiro neurônio de entrada possuir valor 5, porém devido ao seu peso ser 0 significa que ele não causará nenhum efeito sobre a função.

Em seguida teremos de executar uma simples função de soma entre cada entrada e seu respectivo peso e posteriormente a soma dos valores obtidos a partir deles.

Neste exemplo essa função se dará da seguinte forma:

Soma = (1 * 0.8) + (9 * 0.1) + (5 * 0)

Soma = 0.8 + 0.9 + 0

Soma = 1.7

Por fim a função de ativação neste caso, chamada de Step Function (em tradução livre Função Degrau) possui uma regra bastante simples, se o valor resultado da função de soma for 1 ou maior que 1, o neurônio será ativado, se for uma valor abaixo de 1 (mesmo aproximado, mas menor que 1) este neurônio em questão não será ativado.

Step Function = Soma => 1 (Ativação)

Step Function = Soma < 1 (Não Ativação)

Note que esta é uma fase manual, onde inicialmente você terá de realmente fazer a alimentação dos valores de entrada e o cálculo destes valores manualmente, posteriormente entraremos em exemplos onde para treinar uma rede neural teremos inclusive que manualmente corrigir erros quando houverem para que a rede possa “aprender” com essas modificações.

Outro ponto importante é que aqui estamos apenas trabalhando com funções básicas que resultarão na ativação ou não de um neurônio, haverão situações que teremos múltiplas entradas, múltiplas camadas de funções e de ativação e até mesmo múltiplas saídas…

Última observação que vale citar aqui é que o valor de nossa Step Function também pode ser alterado manualmente de acordo com a necessidade.

Com base nesse processamento podemos concluir que de acordo

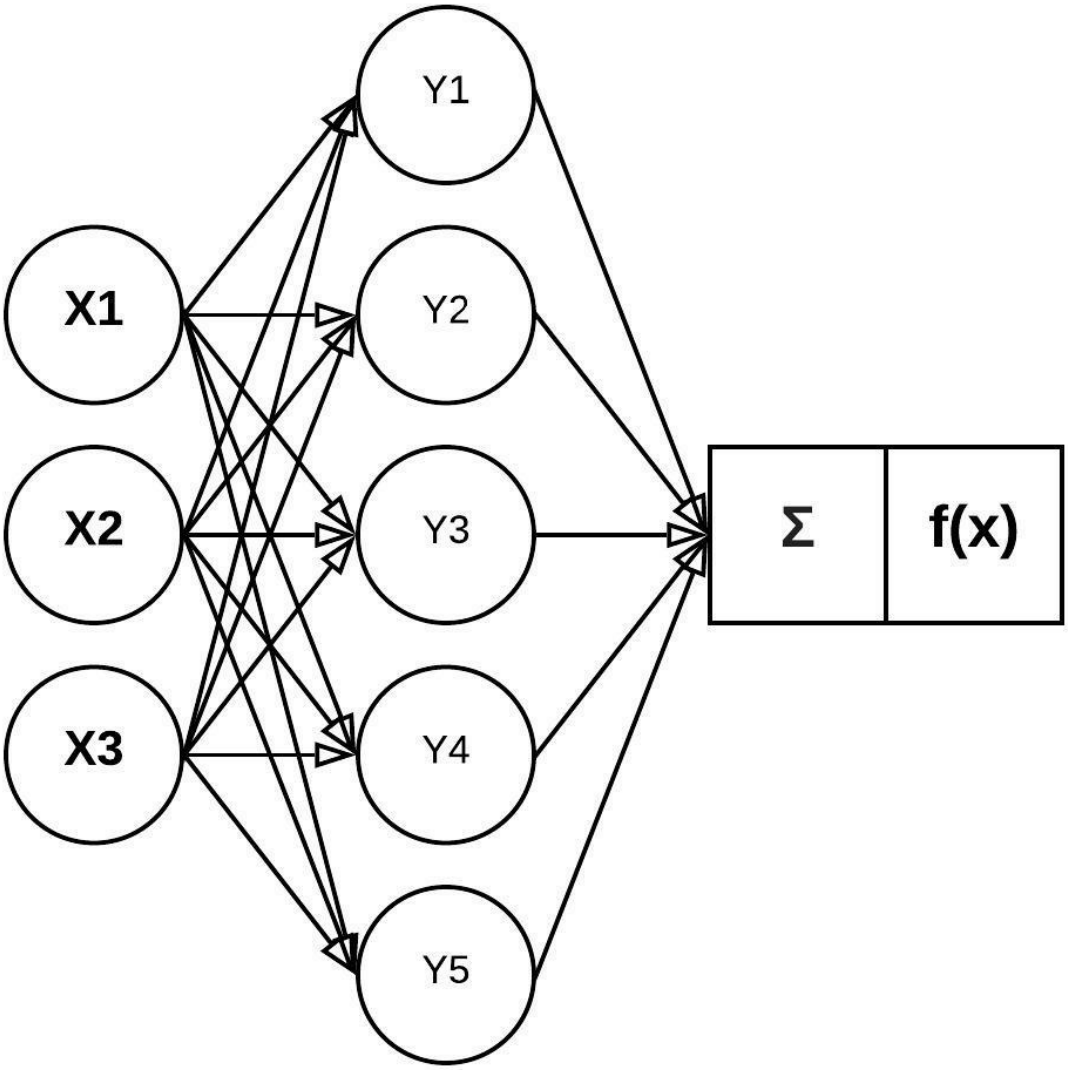
com o parâmetro setado em nossa Step Function, este perceptron em sua aplicação seria ativado, desencadeando uma tomada de decisão ou classificação de acordo com o seu propósito.

Perceptron Multicamada

Uma vez entendida a lógica de um perceptron, e exemplificado em cima de um modelo de uma camada, hora de aumentarmos um pouco a complexidade, partindo para o conceito de perceptron multicamada.

O nome perceptron multicamada por si só é bastante sugestivo, porém é importante que fique bem claro o que seriam essa(s) camada(s) adicionais. Como entendido anteriormente, problemas de menor complexidade que podem ser abstraídos sobre o modelo de um perceptron tendem a seguir sempre a mesma lógica, processar entradas e seus respectivos pesos, executar uma função (em perceptron de uma camada normalmente uma função de soma) e por fim a aplicação de uma função de ativação (em perceptron de uma camada normalmente uma função degrau, multicamada podemos fazer uso de outras como função sigmoide por exemplo).

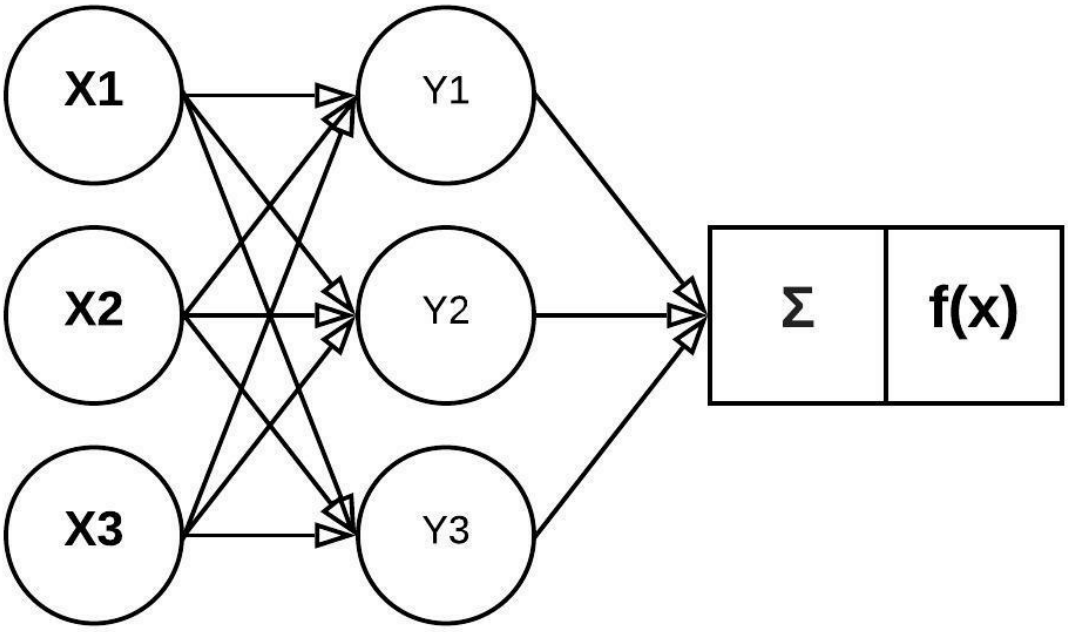
Agora, podemos adicionar o que por convenção é chamado de camada oculta em nosso perceptron, de forma que o processamento dos dados de entrada com seus pesos, passarão por uma segunda camada para ajuste fino dos mesmos, para posteriormente passar por função e ativação.



Repare no modelo acima, temos entradas representadas por X1, X2 e X3 e suas conexões com todos neurônios de uma segunda camada, chamada camada oculta, que se conecta com o neurônio onde é aplicada a função de soma e por fim com a fase final, a fase de ativação.

Além desse diferencial na sua representatividade, é importante ter em mente que assim como o perceptron de uma camada, cada neurônio de entrada terá seus respectivos pesos, e posteriormente, cada nó de cada neurônio da camada oculta também terá um peso associado. Dessa forma, a camada oculta funciona basicamente como um filtro onde serão feitos mais

processamentos para o ajuste dos pesos dentro desta rede.



Apenas para fins de exemplo, note que a entrada X1 tem 3 pesos associados que se conectam aos neurônios Y1, Y2 e Y3, que por sua vez possuem pesos associados à sua conexão com a função final a ser aplicada.

Outro ponto importante de se salientar é que, aqui estamos trabalhando de forma procedural, ou seja, começando dos meios e modelos mais simples para os mais avançados. Por hora, apenas para fins de explicação, estamos trabalhando com poucos neurônios, uma camada oculta, um tipo de função e um tipo de ativação. Porém com uma rápida pesquisada você verá que existem diversos tipos de função de ativação que posteriormente se adequarão melhor a outros modelos conforme nossa necessidade. Nos capítulos subsequentes estaremos trabalhando com modelos diferentes de redes neurais que de acordo com suas particularidades possuem estrutura e número de neurônios e seus respectivos nós diferentes dos exemplos tradicionais mencionados anteriormente.

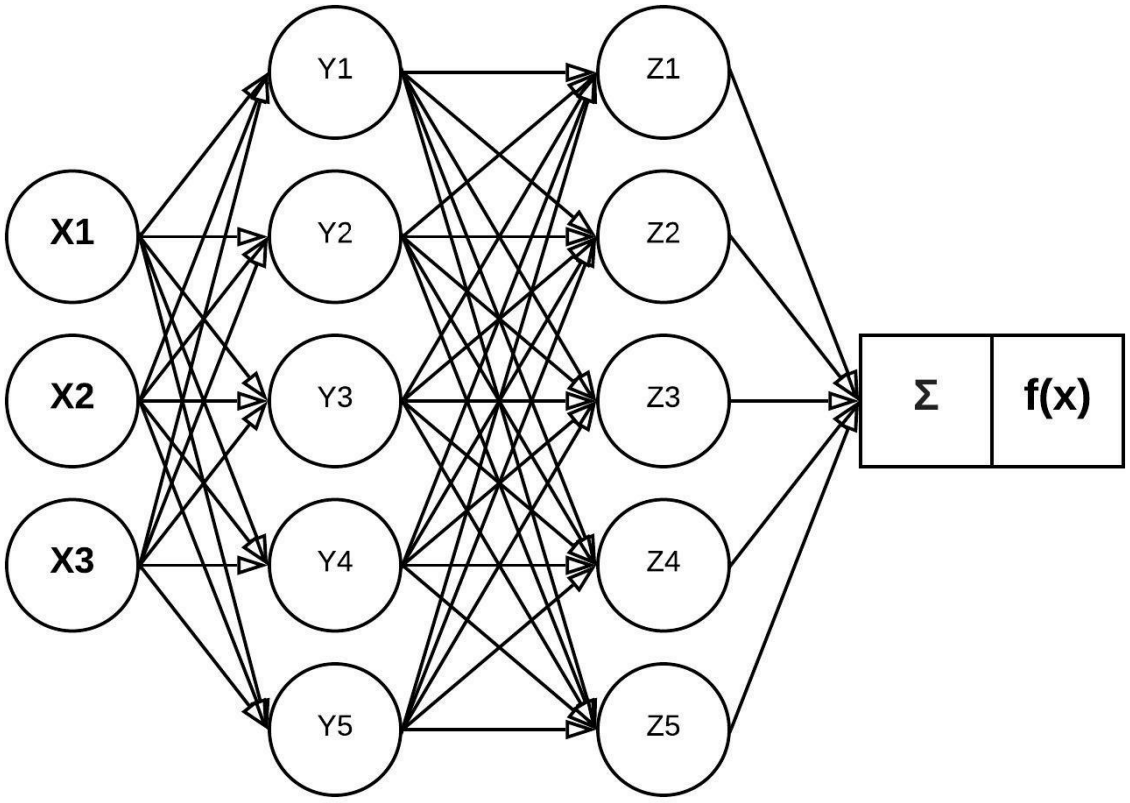
Porém tudo ao seu tempo, inicialmente trabalhando com problemas

computacionais de menor complexidade os modelos acima são suficientes, posteriormente para problemas mais robustos estaremos entendendo os diferentes modelos e formatos de estrutura de uma rede neural artificial.

Deep Learning

Um dos conceitos mais distorcidos dentro dessa área de redes neurais é o de Deep Learning (em tradução livre Aprendizado Profundo). Como vimos anteriormente, uma rede neural pode ser algo básico ou complexo de acordo com a necessidade. Para problemas de lógica simples, tomadas de decisão simples ou classificações linearmente separáveis, modelos de perceptron com uma camada oculta junto a uma função de soma e de ativação correta normalmente é o suficiente para solucionar tal problema computacional.

Porém haverão situações onde, para resolução de problemas de maior complexidade teremos de fazer diferentes tratamentos para nossos dados de entrada, uso de múltiplas camadas ocultas, além de funções de retroalimentação e ativação específicas para determinados fins bastante específicos de acordo com o problema abstraído. O que acarreta em um volume de processamento de dados muito maior por parte da rede neural artificial, o que finalmente é comumente caracterizado como deep learning.



A “aprendizagem” profunda se dá quando criamos modelos de rede neural onde múltiplas entradas com seus respectivos pesos passam por múltiplas camadas ocultas, fases supervisionadas ou não, feedforward e backpropagation e por fim literalmente algumas horas de processamento. Na literatura é bastante comum também se encontrar este ponto referenciado como redes neurais densas ou redes neurais profundas.

Raciocine que para criarmos uma arquitetura de rede neural que aprende por exemplo, a lógica de uma tabela verdade AND (que inclusive criaremos logo a seguir), estaremos programando poucas linhas de código e o processamento do código se dará em poucos segundos. Agora imagine uma rede neural que identifica e classifica um padrão de imagem de uma mamografia digital em uma base de dados gigante, além de diversos testes para análise pixel a pixel da imagem para no fim determinar se uma determinada característica na imagem é ou não câncer, e se é ou não benigno ou maligno (criaremos tal modelo posteriormente). Isto sim resultará em uma rede mais robusta onde criaremos mecanismos lógicos que farão a leitura,

análise e processamento desses dados, requerendo um nível de processamento computacional alto. Ao final do livro estaremos trabalhando com redes dedicadas a processamento puro de imagens, assim como a geração das mesmas, por hora imagine que este processo envolve múltiplos neurônios, múltiplos nós de comunicação, com múltiplas camadas, etc…, sempre proporcional a complexidade do problema computacional a ser resolvido.

Então a fim de desmistificar, quando falamos em deep learning, a literatura nos aponta diferentes nortes e ao leigo isso acaba sendo bastante confuso. De forma geral, deep learning será uma arquitetura de rede neural onde teremos múltiplos parâmetros assim como um grande volume de processamento sobre esses dados por meio de múltiplas camadas de processamento.

Q-Learning e Deep Q-Learning

Uma vez entendidos os princípios básicos de um modelo de rede neural artificial, assim como as principais características comuns aos modelos de redes neurais artificiais, podemos avançar nossos estudos entendendo os mecanismos internos de um algoritmo intuitivo e o que faz dele um modelo a ser usado para simular uma inteligência artificial.

Raciocine que o que visto até então nos capítulos anteriores é como o núcleo do que usaremos em redes neurais artificiais, independentemente do seu tipo ou propósito. Ao longo dos exemplos estaremos entendendo de forma prática desde o básico (perceptrons) até o avançado (inteligência artificial), e para isso temos que identificar cada uma dessas particularidades em seu contexto.

Por hora, também é interessante introduzir os conceitos de inteligência artificial, porém os mesmos serão estudados com maior profundidade em seus respectivos capítulos.

Para separarmos, sem que haja confusão, conceitos teóricos de machine learning, deep learning e q-learning, estaremos sempre que possível realizando a comparação entre esses modelos.

Dando sequência, vamos entender as particularidades em Q-Learning.

Equação de Bellman

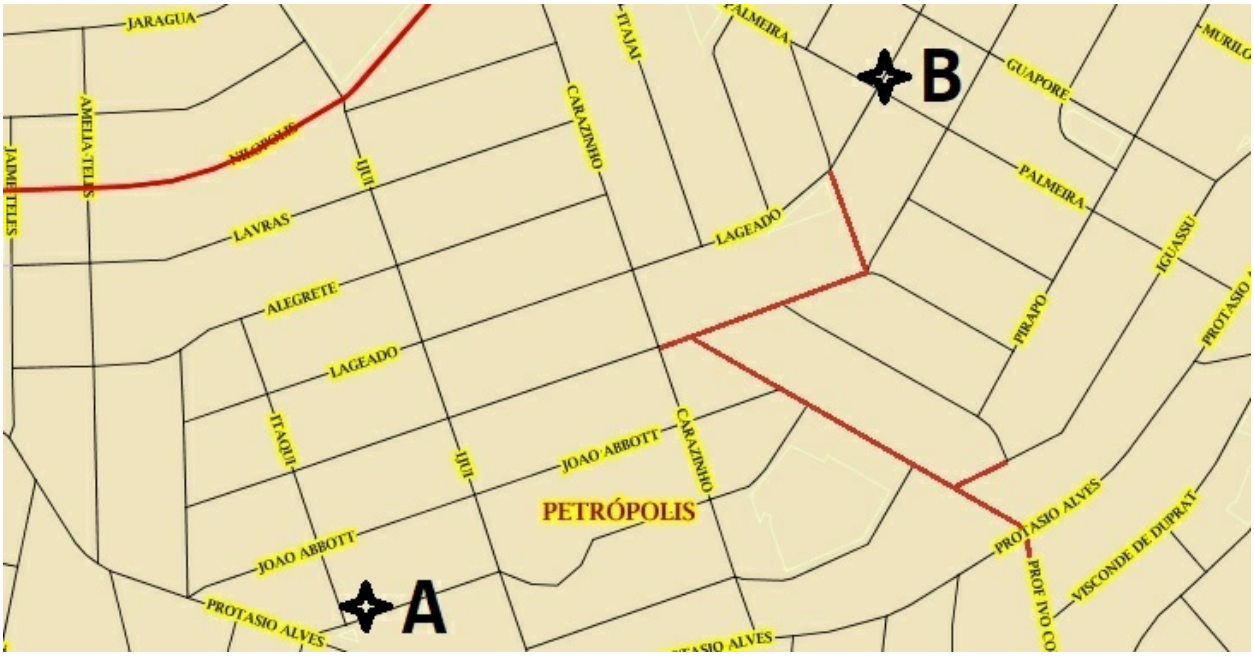
Richard Bellman foi um matemático estadunidense referenciado no meio da ciência da computação pela “invenção” da programação dinâmica em 1953, modelo esse de equações e algoritmos aplicados para estados de programação em tempo real. Em outras palavras, por meio de suas fórmulas é possível criar arquiteturas de código onde camadas de processamento podem ser sobrepostas e reprocessadas em tempo real. Como mencionado anteriormente, um dos grandes diferenciais do modelo de rede neural

intuitiva é a capacidade da mesma de trabalhar continuamente, realizando processamento de seus estados e memória em tempo real.

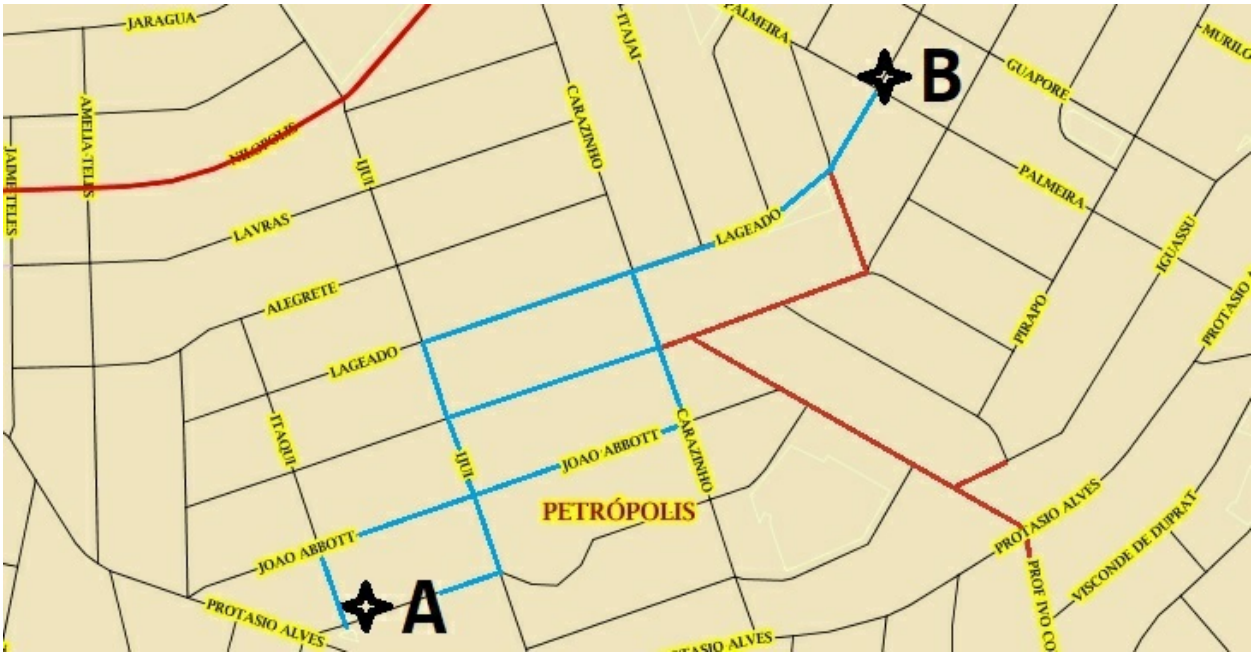
Por parte da lógica desse modelo, raciocine que teremos um agente, um ambiente e um objetivo a ser alcançado. O agente por si só terá uma programação básica de quais ações podem ser tomadas (em alguns casos uma programação básica adicional de seu tamanho e formado) e com base nisso o mesmo realizará uma série de testes fazendo o mapeamento do ambiente até que atinja seu objetivo.

Quando atingido o objetivo esperado, o agente irá verificar qual caminho percorreu, quais sequências de ações foram tomadas do início ao final do processo e irá salvar essa configuração em uma espécie de memória. A experiência desse agente com o ambiente se dará de forma sequencial, uma ação após a outra, de forma que caso ele atinja algum obstáculo receberá uma penalidade, assim como quando ele alcançar seu objetivo receberá uma recompensa.

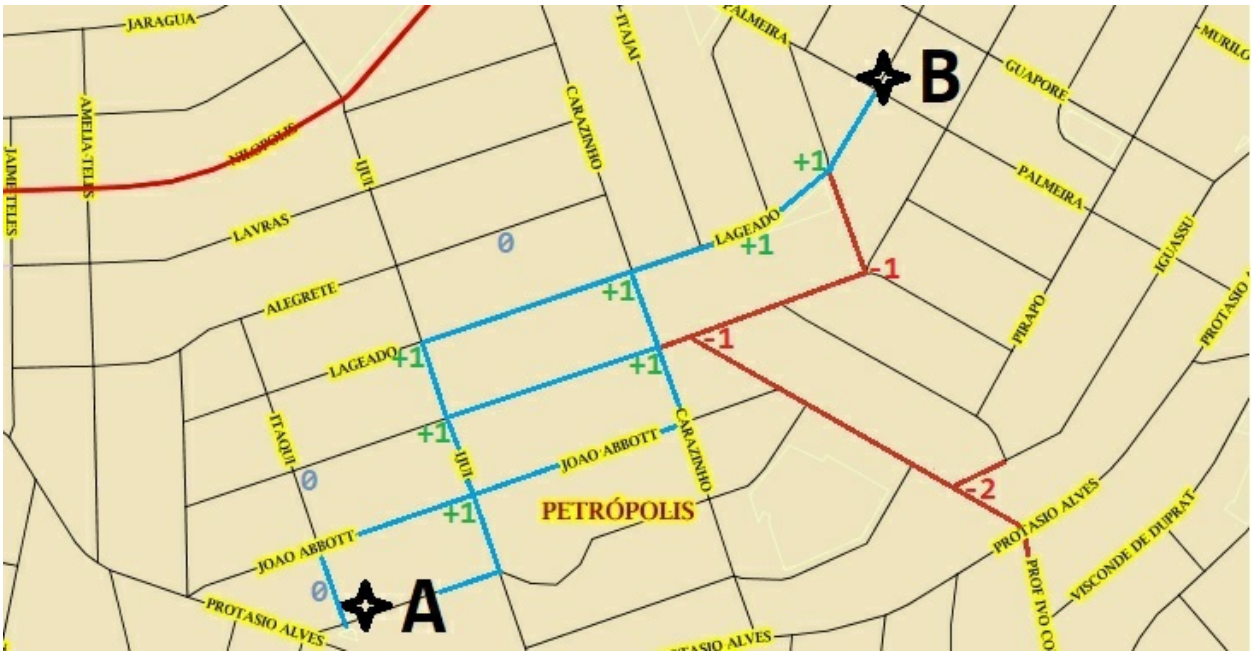
Toda trajetória será remapeada, passo a passo realizado, atribuindo valores de experiência positiva quando correto, de forma que numa situação de repetir o mesmo percurso, o agente irá buscar e replicar as ações que ele já conhece e sabe que deram certo ao invés de tentar novas experiências.



Apenas como exemplo, imagine que você possui um agente (com suas características de possíveis ações a serem tomadas bem definidas), um ambiente que nesse caso trata-se de um mapa de uma determinada cidade e um objetivo de ir do ponto A até o B. Realizando uma leitura rápida desse mapa podemos ver duas características importantes, primeira delas que existem diversas possíveis rotas que fazem o trajeto do ponto A até o ponto B. Note que destacado em linhas vermelhas temos alguns segmentos dessas trajetórias que por algum motivo estão fechadas, sendo assim, o mapeamento de todas possíveis rotas é feito.



Agora delimitados como linhas azuis estão traçadas algumas rotas diretas entre o ponto A e o ponto B. O que nosso agente irá fazer é percorrer cada uma dessas rotas identificando sempre o seu nível de avanço conforme a distância de seu objetivo diminui, assim como demarcar cada etapa concluída com uma pontuação positiva para cada possibilidade válida executada com sucesso, assim como uma pontuação negativa quando atingido qualquer estado onde seu progresso for interrompido. Lembrando que cada passo em busca de seu objetivo é feito uma nova leitura do seu estado atual e sua posição no ambiente.



Apenas exemplificando, note que pontuamos alguns trajetos com uma numeração entre -2, -1, 0 e +1 (podendo usar qualquer métrica ao qual se sinta mais confortável adotar). Possíveis trajetos classificados como 0 são aqueles que não necessariamente estão errados, porém não são os mais eficientes, no sentido de ter uma distância maior do objetivo sem necessidade. Em vermelho, nos entroncamentos das ruas representadas por linhas vermelhas temos pontuação -2 por exemplo, onde seria um trajeto que nosso agente testou e considerou muito fora da rota esperada.

Da mesma forma os entroncamentos pontuados em -1 por sua vez foram testados pelo agente e considerados ineficientes. Por fim, sobre as linhas azuis que representam os melhores caminhos, existem marcações +1 sinalizando que tais caminhos são válidos e eficientes em seu propósito de traçar a menor distância entre o ponto A e o ponto B.

Como mencionado anteriormente, o agente arbitrariamente irá testar todos os possíveis trajetos deste ambiente, gerando com base em sua experiência um mapa dessas rotas corretas. Desse modo, caso essa função tenha de ser repetida o agente irá se basear puramente nos resultados corretos que ele conhece de sua experiência passada.

$$V(s) = \max_{a}(R(s,a) + \gamma V(s'))$$

Traduzindo este conceito lógico em forma de equação chegamos na chamada equação de Bellman, equação esta que integrará o algoritmo de nossa rede neural de forma a realizar cálculos para cada estado do agente, podendo assim prever qual a próxima ação a ser tomada de acordo com os melhores resultados, já que sempre estamos buscando melhor eficiência.

Sem nos aprofundarmos muito no quesito de cálculo, podemos entender a lógica dessa equação para entendermos como a mesma é alimentada de dados e realiza seu processamento.

Na equação escrita acima, temos alguns pontos a serem entendidos, como:

V – Possíveis valores de cada estado a cada etapa.

s – Estado atual de nosso agente.

s’ – Próximo estado de nosso agente.

A – Ação a ser tomada.

R – Recompensa gerada para o resultado da ação tomada por nosso agente.

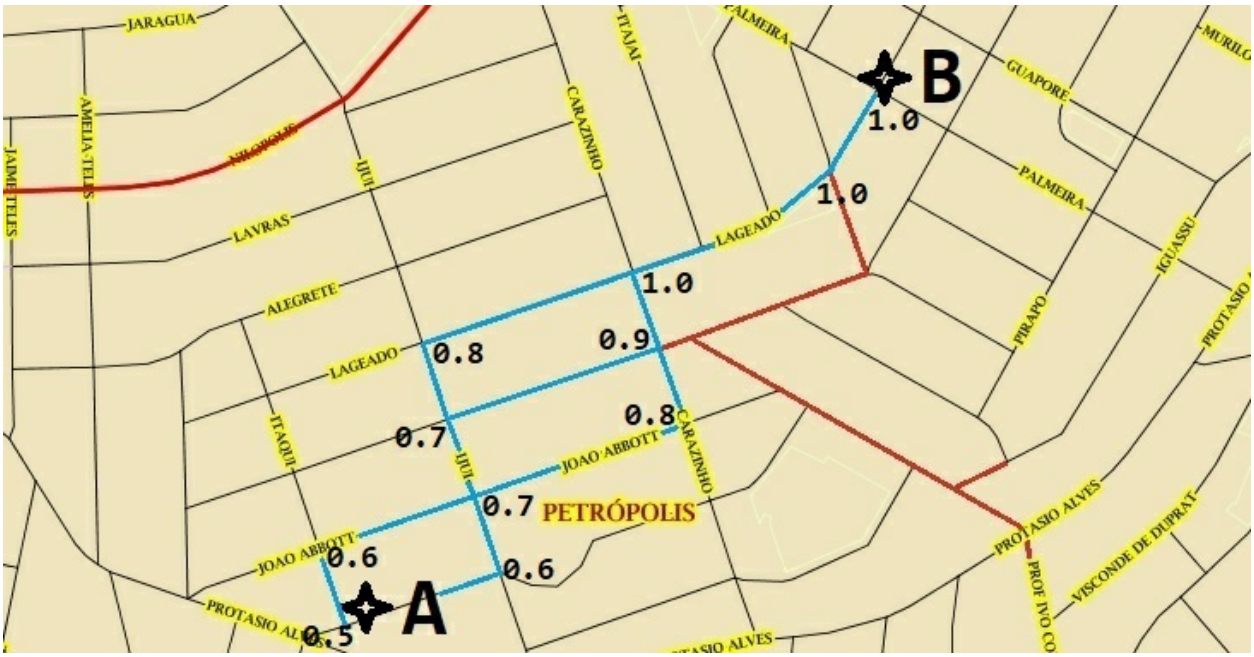
y – Fator de desconto.

V (s) – Qual é o valor do último estado de nosso agente.

R (s, a) – Lendo inversamente, de acordo com a última ação o agente recebe um valor de estado para que seja recompensado ou penalizado.

* A equação de Bellman é calculada a cada estado de nosso agente, atualizando assim os seus dados de estado.

* Mesmo que numericamente os estados se repitam quando calculados manualmente, vale lembrar que a cada ação executada um novo estado é alcançado.



Com base na aplicação da equação de Bellman para cada estado é retroativamente pontuados os valores dos melhores caminhos e tais dados passam a compor o equivalente a memória de longa duração de nosso agente.

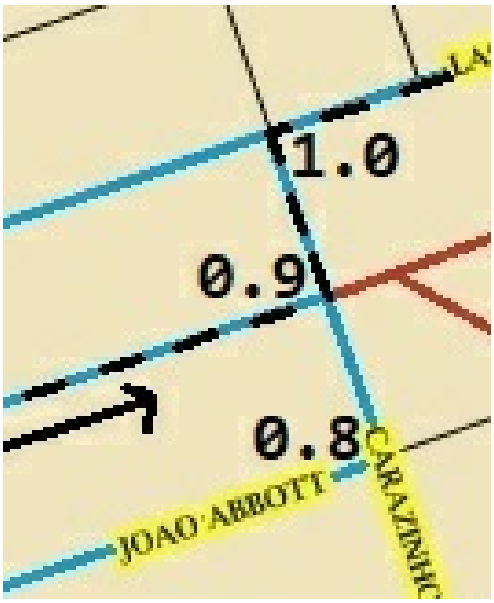
Se o mesmo objetivo com os mesmos parâmetros for necessário ser realizado novamente, nosso agente já possui todos os dados de experiência, apenas necessitando replicá-los.

Plano ou Política de Ação

Note que em nosso exemplo a pontuação aplicada para cada estado segue a simples lógica de que quanto maior for o número mais próximo do objetivo estará o nosso agente. Como normalmente acontece na prática,

existirão diversos caminhos corretos, o que nosso agente fará para descobrir a melhor ou mais eficiente rota simplesmente será fazer a sua leitura do seu estado atual, assim como as pontuações dos possíveis estados e ações a serem tomadas, optando automaticamente para avançar para o próximo estado mapeado de maior pontuação.

Essa ação tomada de forma mais direta normalmente é denominada como plano de ação, para alguns autores, este tipo de lógica usada para tomada de decisão é como se fosse uma modelo de aprendizado de máquina auto supervisionado, onde o agente sabe com margem de precisão muito alta o que deve ser feito, porém em situações de maior complexidade teremos inúmeros fatores que poderão prejudicar essa precisão, fazendo com que o agente tenha que considerar esses fatores impactando diretamente seu desempenho, ou a probabilidade de sucesso de suas tomadas de decisão.



Apenas como exemplo, seguindo a linha tracejada indicada pela seta, ao chegar no entroncamento nosso agente terá de fazer a leitura de seu estado atual, do ambiente (bloqueado à frente, com caminhos possíveis para a direita e para a esquerda), e fará sua tomada de decisão, nesse caso aplicando seu plano de ação para 1 de apenas 2 direções, e com base no valor mais alto do próximo estado, nesse caso, seguindo o caminho à esquerda.

Haverá situações onde o agente terá muitos fatores que prejudicam sua leitura de estado ou de ambiente, assim como pontos de estado alcançado onde será necessário realizar múltiplas escolhas. Para essas situações devemos considerar todos os fatores inclusive dados de ações imediatamente realizadas nos passos anteriores ou a leitura da possível modificação do ambiente entre um estado e outro para que se determine direções com melhor probabilidade de acertos.

Como exemplo, imagine a complexidade do sistema de tomada de decisão de um carro autônomo, sua IA faz a leitura do ambiente dezenas de vezes por segundo, também levando em consideração que dependendo da velocidade do carro percorrendo um determinado trajeto, em uma fração de segundo o ambiente pode se modificar à sua frente caso algum animal atravesse a pista ou um objeto seja jogado nela como exemplo. Raciocine que existe uma série de ações a serem tomadas seja tentando frear o carro ou desviar do obstáculo (ou as duas coisas) ainda tendo que levar em consideração qual tipo de ação terá maior probabilidade de sucesso, tudo em uma fração de segundo.

Sistema de Recompensas e Penalidades

Como vimos nos tópicos anteriores, nosso agente de acordo com suas ações recebe recompensas ou penalidades como consequência de seus atos, porém é necessário entender um pouco mais a fundo como esse mecanismo de fato funciona em nosso algoritmo.

Anteriormente também comentamos que uma das principais características de nosso modelo de rede neural artificial intuitiva é a de que nosso agente a partir de um gatilho inicial se torna “vivo”, agindo de forma constante e independente. Isso se dá porque devemos considerar que todo e qualquer tipo de processamento constante funciona de forma cíclica, podendo ter uma série de particularidades em seu ciclo, mas basicamente um ciclo de processamento tem um estágio inicial, camadas de processamento de entradas gerando saídas que retroalimentam as entradas gerando um loop de processamento.

Contextualizando para nosso agente, todo e qualquer agente sempre fará o ciclo de leitura de seu estado atual (inicial), tomará uma série de ações para com o ambiente ao seu redor, recebendo recompensas ou penalidades de acordo com as consequências de suas ações, retroalimentando com estes dados o seu novo estado que por sua vez é o seu “novo” estado atual (inicial).

Para alguns autores, pontuar essas recompensas e penalidades facilita a interpretação dos estados de nosso agente, porém devemos tomar muito cuidado com esse tipo de interpretação, uma vez que se você raciocinar a lógica deste modelo, na realidade teremos uma leitura diferencial, com muito mais penalidades do que recompensas uma vez que independentemente da tarefa a ser realizada, boa parte das vezes existem poucos meios de realizar tal função corretamente, em contraponto a inúmeras maneiras erradas de se tentar realizar a mesma.

Logo, devemos ter esse tipo de discernimento para que possamos de fato ler o estado de nosso agente e entender suas tomadas de decisão. Internamente, por parte estruturada do algoritmo, lembre-se que a aprendizagem por reforço justamente visa dar mais ênfase aos acertos, dando mais importância a eles dentro do código, enquanto alguns modelos até mesmo descartam totalmente grandes amostragens de erros, uma vez que eles são maioria e podem ocupar muito da capacidade de memória de nosso agente.

Na prática, nos capítulos onde estaremos implementando tais conceitos em código, você notará que temos controle da taxa de aprendizado de nosso agente, assim como temos controle dos pesos das penalidades no processo de aprendizado de máquina. Teremos meios para dar mais pesos aos erros enfatizando que os mesmos tenham seus padrões facilmente identificados pela rede neural de forma a reforçar o aprendizado da rede pelo viés correto.

Aplicação sob Diferença Temporal

A essa altura, ao menos por parte teórica, você já deve estar entendendo o funcionamento lógico de uma rede neural artificial intuitiva, assim como o papel de nosso agente, a maneira como o mesmo se comporta em relação ao ambiente e as suas ações.

Hora de aprofundar um pouco o conceito de diferença temporal, haja visto que já temos uma boa noção sobre o mesmo.

Em suma, quando estamos falando que neste tipo específico de rede neural artificial temos processamento em tempo real, aprendizado autosupervisionada, tempo de ação, tomada de ação autônoma, etc... estamos falando de estruturas lógicas que irão simular todo processamento necessário dessa inteligência artificial em decorrência de um período de tempo real. Caso você procure por artigos científicos que tratam sobre esse assunto verá inúmeros exemplos de como moldar uma estrutura lógica e de código de forma que são aproveitados resultados de amostras de processamento passados assim como pré-programados já estarão uma série de novas ações a serem tomadas, repetindo esses passos em ciclo.

Todo e qualquer código tem o que chamamos de interpretação léxica, uma forma sequencial de ler linhas e blocos de código. A partir dos mesmos, temos de criar estruturas que fiquem carregadas na memória realizando processamento de dados de entrara e saída de forma sequencial e cíclica. Um computador não tem (ainda) discernimento sobre coisas tão abstratas como fisicamente se dá a passagem do tempo, o que temos são dados de tempo de processamento contínuo, que será usado para simular tempo real.

Tenha em mente que diferente dos modelos de redes neurais artificiais convencionais, onde o processamento tem um começo, meio e fim bem definidos, nosso agente terá para si um modelo de rede neural artificial baseado em um começo (um gatilho inicial), meio e um recomeço. Dessa maneira simulamos continuidade em decorrência do tempo.

Outro ponto importante a ser observado é que este modelo, quando comparado aos modelos de redes neurais artificiais comuns, não possui explicitamente dados base, dados para treino e teste, no lugar destas estruturas estaremos criando o que por alguns autores é a chamada memória de replay, onde a base se constrói gradualmente por dados de experiência se auto retroalimentando.

Apenas finalizando essa linha de raciocínio, você terá de moldar seu agente (e isso será visto na prática nos capítulos de implementação) de forma que o mesmo interpretará estados passados (últimas ações executadas), estados presentes (estado atual + mapeamento do ambiente + recompensa

obtida) e estados futuros (próximas ações a serem tomadas) de forma que essa sequência se repetirá de forma intermitente.

$$V(s) = \max_{a}(R(s,a) + \gamma V(s'))$$

Anteriormente, vimos que de acordo com a equação de Bellman, havia uma variável V que por sua vez representava os possíveis valores de cada estado a cada etapa.

$$Q(s,a) = R(s,a) + \gamma \max_{a'} Q(s',a')$$

Agora temos uma variável Q (s, a) que representa valores que já foram computados em uma fase de tomada de ação anterior e que está armazenado em memória, inclusive levando em consideração o valor de recompensa obtido anteriormente.

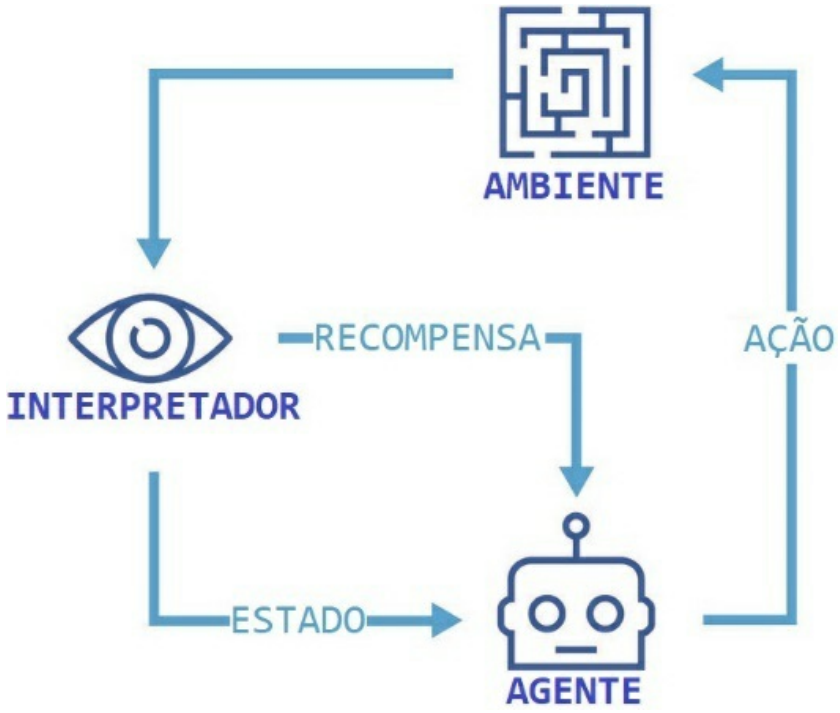
Repare que dessa maneira, estamos sempre trabalhando levando em consideração o estado anterior para que seja previsto o próximo estado.

$$TD(a,s) = R(s,a) + \gamma \max_{a'} Q(s',a') - Q(s,a)$$

Assim chegamos na fórmula da equação final, onde temos uma variável TD representando a diferença temporal, que por sua vez realizará a atualização dos valores de Q. Novamente, a cada etapa de execução, a aplicação dessa fórmula será realizada, com o diferencial que agora ela não se inicia aleatoriamente, mas a partir do último valor de estado obtido, gerando um ciclo de processamento retroalimentado e capaz de simular continuidade, já que sempre será retroalimentado e sempre será tomada uma nova decisão.

Modelo de Rede Neural Artificial Intuitiva

Apenas encerrando essa etapa, com o conhecimento acumulado até então, podemos finalmente estruturar um modelo lógico a ser transformado para código.



Em suma, teremos um agente, que por sua vez pode ser qualquer tipo de objeto ou ser vivo; Da mesma forma teremos um ambiente a ser explorado, com suas características particulares inicialmente desconhecidas por nosso agente; Sendo assim, nosso agente após ser ativado uma vez irá reconhecer seu formato e possíveis funções e a partir disto começará a explorar o ambiente, literalmente por tentativa e erro assim como milhares (ou até mesmo milhões) de execuções dependendo da complexidade de seu objetivo. À medida que o mesmo realiza determinadas ações, é recompensado ou penalizado para que dessa forma encontre o padrão correto de alcançar seu objetivo.

Rotinas de uma Rede Neural Artificial

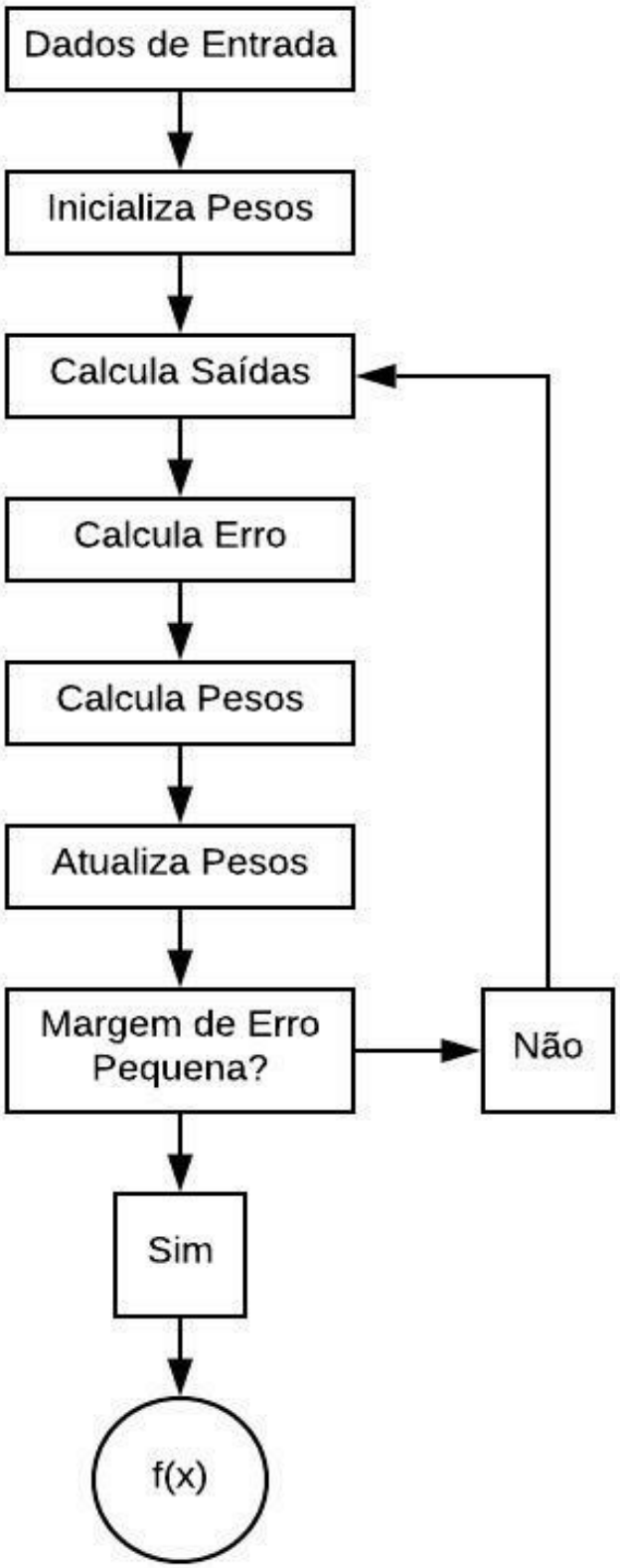
Independentemente da área que estivermos falando sempre costumamos dividir nossas atribuições em rotinas que executaremos ao longo de um período de tempo, é assim em nossa rotina de trabalho, de estudos, até mesmo de tentar estabelecer padrões de como aproveitar melhor o pouco tempo que temos longe destes… Sendo assim, uma rotina basicamente será aquela sequência de processos que estaremos executando sempre dentro de uma determinada atividade. Em machine learning, no âmbito profissional, teremos diversos tipos de problemas a serem contextualizados para nossas redes neurais e sendo assim teremos uma rotina a ser aplicada sobre esses dados, independente de qual seja o resultado esperado.

No primeiro dataset que usaremos de exemplo para apresentar uma rede neural mais robusta, estaremos executando uma rotina bastante completa que visa extrair o máximo de informações a partir de um banco de dados pré-estabelecido, assim como estaremos criando nossa rede neural em um modelo onde determinados processos serão divididos em blocos que poderão ser reutilizáveis se adaptados para outros contextos posteriormente. Tenha em mente que uma vez criadas nossas ferramentas não há necessidade de cada vez criá-las a partir do zero, estaremos elucidando quais rotinas de processos melhor se adaptam de acordo com os tipos de problema computacional apresentado e dessa forma, uma vez entendido os conceitos e codificadas suas ferramentas, posteriormente você poderá simplesmente as reajustar de acordo com a necessidade.

Trabalhando sobre a Breast Cancer Dataset, primeira rede neural mais robusta que estaremos criando e entendendo seus conceitos passo a passo a seguir, estaremos executando uma determinada rotina que não necessariamente se aplica a todos os demais problemas computacionais, mas já servirá de exemplo para criar uma boa bagagem de conhecimento sobre o processo de importação e tratamento dos dados, criação e aplicação de uma rede neural artificial assim como a configuração da mesma a fim de obter melhores resultados. Raciocine que posteriormente em outros datasets estaremos trabalhando e usando outros modelos e ferramentas que não se aplicariam no contexto da Breast Dataset, porém começar por ela promoverá

um entendimento bom o suficiente para que você entenda também de acordo com o problema, a rotina de processamento a ser aplicada.

Toda rede neural pode começar a ser contextualizada por um algoritmo bastante simples, onde entre nossas entradas e saídas temos:


Dados de Entrada
Inicializa Pesos
Calcula Saídas
Calcula Erro
Calcula Pesos
Atualiza Pesos
Margem de Erro Pequena?
Não
Sim
f(x)

Em algumas literaturas ainda podemos encontrar como rotina: Fase de criação e modelagem da rede > Fase supervisionada de reajustes da mesma > Fase de aprendizado de máquina > Fase de testes de performance da rede e dos resultados > Saídas.

Como mencionado nos capítulos anteriores, aqui usaremos de uma abordagem progressiva e bastante didática. Estaremos sim executando esses modelos de rotinas mencionados acima, porém de uma forma mais natural, seguindo a lógica a qual estaremos criando e aplicando linha a linha de código em nossa rede neural.

Sendo assim, partindo para prática estaremos com base inicial no Breast Cancer Dataset entendendo e aplicando cada um dos passos abaixo:

Rotina Breast Cancer Dataset

Rede neural simples:

Importação da base de dados de exemplo

Criação de uma função sigmoide de forma manual

Criação de uma função Sigmoide Derivada de forma manual

Tratamento dos dados, separando em atributos previsores e suas saídas

Criação de uma rede neural simples, camada a camada manualmente com aplicação de pesos aleatórios, seus reajustes em processo de aprendizado de máquina para treino do algoritmo.

Rede neural densa:

Importação da base de dados a partir de arquivo .csv

Tratamento dos dados atribuindo os mesmos em variáveis a serem usadas para rede de forma geral, assim como partes para treino e teste da rede neural.

Criação de uma rede neural densa multicamada que irá inicialmente classificar os dados a partir de uma série de camadas que aplicarão funções pré-definidas e parametrizadas por ferramentas de bibliotecas externas e

posteriormente gerar previsões a partir dos mesmos.

Teste de precisão nos resultados e sobre a eficiência do algoritmo de rede neural.

Parametrização manual das ferramentas a fim de obter melhores resultados

Realização de técnicas de validação cruzada, e tuning assim como seus respectivos testes de eficiência.

Teste de viés de confirmação a partir de uma amostra.

Salvar o modelo para reutilização.

Posteriormente iremos trabalhar de forma mais aprofundada em outros modelos técnicas de tratamento e processamento de dados que não se aplicariam corretamente na Breast Cancer Dataset mas a outros tipos de bancos de dados.

Tenha em mente que posteriormente estaremos passo-a-passo trabalhando em modelos onde serão necessários um polimento dos dados no sentido de a partir de um banco de dados bruto remover todos dados faltantes, errôneos ou de alguma forma irrelevantes para o processo.

Estaremos trabalhando com um modelo de abstração chamado de variáveis do tipo Dummy onde teremos um modelo de classificação com devidas particularidades para classificação de muitas saídas.

Também estaremos entendendo como é feito o processamento de imagens por uma rede neural, neste processo, iremos realizar uma série de processos para conversão das informações de pixel a pixel da imagem para dados de uma matriz a ser processado pela rede.

Enfim, sob a sintaxe da linguagem Python e com o auxílio de algumas funções, módulos e bibliotecas estaremos abstraindo e contextualizando problemas da vida real de forma a serem solucionados de forma computacional por uma rede neural artificial.

APRENDIZADO DE MÁQUINA

Perceptron de Uma Camada – Modelo Simples

Anteriormente entendemos a lógica de um Perceptron, assim como as particularidades que estes possuem quanto a seu número de camadas, deixando um pouco de lado o referencial teórico e finalmente partindo para prática, a seguir vamos codificar alguns modelos de perceptron, já no capítulo seguinte entraremos de cabeça na codificação de redes neurais artificiais.

Partindo para o Código:

Inicialmente vamos fazer a codificação do algoritmo acima, pondo em prática o mesmo, através do Google Colaboratory (sinta-se livre para usar o Colaboratory ou o Jupyter da suíte Anaconda), ferramentas que nos permitirão tanto criar este perceptron quanto o executar em simultâneo. Posteriormente em problemas que denotarão maior complexidade estaremos usando outras ferramentas.



Abrindo nosso Colaboratory inicialmente renomeamos nosso notebook para Perceptron 1 Camada.ipynb apenas por convenção. *Ao criar um notebook Python 3 em seu Colaboratory automaticamente será criada uma pasta em seu Drive de nome Colab Notebooks, onde será salva uma cópia deste arquivo para posterior utilização.

Por parte de código apenas foi criado na primeira célula um

comentário, na segunda célula foram declaradas duas variáveis de nomes entradas e pesos, respectivamente. Como estamos atribuindo vários valores para cada uma passamos esses valores em forma de lista, pela sintaxe Python, entre chaves e separados por vírgula. Aqui estamos usando os mesmos valores usados no modelo de perceptron anteriormente descrito, logo, entradas recebe os valores 1, 9 e 5 referentes aos nós de entrada X, Y e Z de nosso modelo assim como pesos recebe os valores 0.8, 0.1 e 0 como no modelo.

```
[3]
def soma(e,p):
  s = 0
  for i in range(3):
    s += e[i] * p[i]
  return s
```

Em seguida criamos uma função de nome soma que recebe como parâmetro as variáveis temporárias e e p. Dentro dessa função inicialmente criamos uma nova variável de nome s que inicializa com valor 0. Em seguida criamos um laço de repetição que irá percorrer todos valores de e e p, realizar a sua multiplicação e atribuir esse valor a variável s. Por fim apenas deixamos um comando para retornar s, agora com valor atualizado.

```
[4]
s = soma(entradas,pesos)
```

```
[5]
print(s)
```

```
1.7000000000000002
```

Criamos uma variável de nome s que recebe como atributo a função soma, passando como parâmetro as variáveis entradas e pesos. Executando uma função print( ) passando como parâmetro a variável s finalmente podemos ver que foram feitas as devidas operações sobre as variáveis, retornando o valor de 1.7, confirmando o modelo anterior.

```
[6]
def stepFunction(s):
  if (s >= 1):
    return 1
  return 0
```

Assim como criamos a função de soma entre nossas entradas e seus respectivos pesos, o processo final desse perceptron é a criação de nossa função degrau. Função essa que simplesmente irá pegar o valor retornado de soma e estimar com base nesse valor se esse neurônio será ativado ou não.

Para isso simplesmente criamos outra função, agora de nome stepFunction( ) que recebe como parâmetro s. Dentro da função simplesmente criamos uma estrutura condicional onde, se o valor de s for igual ou maior que 1, retornará 1 (referência para ativação), caso contrário, retornará 0 (referência para não ativação).

```
[7]    1 saida = stepFunction(s)

[8]    1 print(saida)

       1
```

Em seguida criamos uma variável de nome saida que recebe como atributo a função stepFunction( ) que por sua vez tem como parâmetro s. Por fim executamos uma função print( ) agora com saida como parâmetro, retornando finalmente o valor 1, resultando como previsto, na ativação deste perceptron.

Código Completo:

```python
#Perceptron de Uma Camada

entradas = [1, 9, 5]
pesos = [0.8, 0.1, 0]

def soma(e,p):
  s = 0
  for i in range(3):
    s += e[i] * p[i]
  return s

s = soma(entradas,pesos)

def stepFunction(s):
  if (s >= 1):
    return 1
  return 0

saida = stepFunction(s)

print(s)
print(saida)
```

```
1.7000000000000002
1
```

Aprimorando o Código:

Como mencionado nos capítulos iniciais, uma das particularidades por qual escolhemos desenvolver nossos códigos em Python é a questão de termos diversas bibliotecas, módulos e extensões que irão facilitar nossa vida oferecendo ferramentas que não só automatizam, mas melhoram processos no que diz respeito a performance. Não existe problema algum em mantermos o código acima usando apenas os recursos do interpretador do Python, porém se podemos realizar a mesma tarefa de forma mais reduzida e com maior performance por quê não o fazer.

Sendo assim, usaremos aplicado ao exemplo atual alguns recursos da biblioteca Numpy, de forma a usar seus recursos internos para tornar nosso código mais enxuto e eficiente. Aqui trabalharemos pressupondo que você já instalou tal biblioteca como foi orientado em um capítulo anterior.

```python
[1]  import numpy as np
```

Sempre que formos trabalhar com bibliotecas que nativamente não são carregadas e pré-alocadas por nossa IDE precisamos fazer a importação das mesmas. No caso da Numpy, uma vez que essa já está instalada no sistema, basta executarmos o código import numpy para que a mesma seja carregada e possamos usufruir de seus recursos. Por convenção quando importamos alguns tipos de biblioteca ou módulos podemos as referenciar por alguma abreviação, simplesmente para facilitar sua chamada durante o código. Logo, o comando import numpy as np importa a biblioteca Numpy e sempre que a formos usar basta iniciar o código com np.

```
[2]    1 entradas = np.array([1, 9, 5])
       2 pesos = np.array([0.8, 0.1, 0])
```

Da mesma forma que fizemos anteriormente, criamos duas variáveis que recebem os respectivos valores de entradas e pesos. Porém, note a primeira grande diferença de código, agora não passamos uma simples lista de valores, agora criamos uma array Numpy para que esses dados sejam vetorizados e trabalhados internamente pelo Numpy. Ao usarmos o comando np.array( ) estamos criando um vetor ou matriz, dependendo da situação, para que as ferramentas internas desta biblioteca possam trabalhar com os mesmos.

```
[3]    1 def Soma(e,p):
       2   return e.dot(p)
```

Segunda grande diferença agora é dada em nossa função de Soma, repare que agora ela simplesmente contém como parâmetros e e p, e ela executa uma única linha de função onde ela retornará o produto escalar de e sobre p. Em outras palavras, o comando e.dot(p) fará o mesmo processo aritmético que fizemos manualmente, de realizar as multiplicações e somas dos valores de e com p, porém de forma muito mais eficiente.

```
[4]
```

```
s = Soma(entradas, pesos)

def stepFunction(Soma):
  if (s >= 1):
    return 1
  return 0

saida = stepFunction(s)

print(s)
print(saida)
```

```
1.7000000000000002
1
```

Todo o resto do código é reaproveitado e executado da mesma forma, obtendo inclusive como retorno os mesmos resultados (o que é esperado), a diferença de trocar uma função básica, baseada em listas e condicionais, por um produto escalar realizado por módulo de uma biblioteca dedicada a isto torna o processo mais eficiente. Raciocine que à medida que formos implementando mais linhas de código com mais funções essas pequenas diferenças de performance realmente irão impactar o desempenho de nosso código final.

Código Completo:

```python
import numpy as np

entradas = np.array([1, 9, 5])
pesos = np.array([0.8, 0.1, 0])

def Soma(e,p):
  return e.dot(p)

s = Soma(entradas, pesos)

def stepFunction(Soma):
  if (s >= 1):
    return 1
  return 0

saida = stepFunction(s)

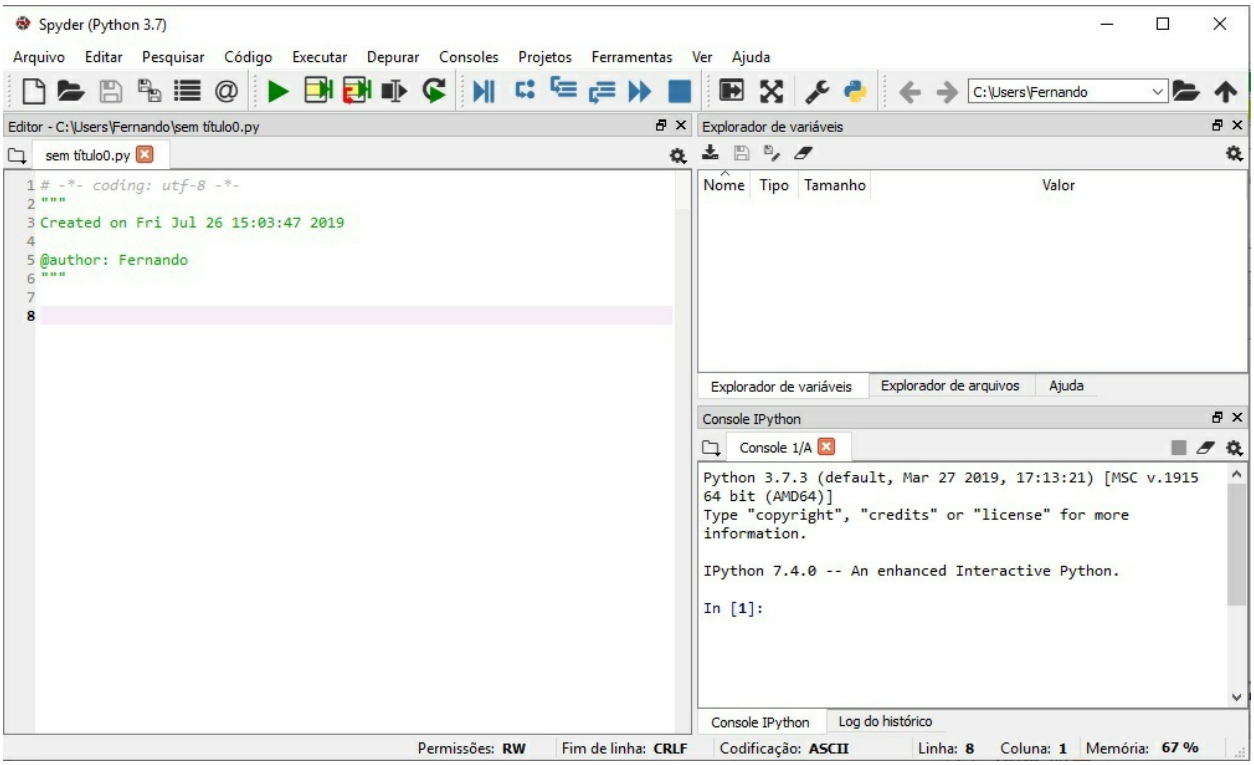
print(s)
print(saida)
```

```
1.7000000000000002
1
```

Usando o Spyder 3:

Os códigos apresentados anteriormente foram rodados diretamente no notebook Colab do Google, porém haverão situações de código mais complexo onde não iremos conseguir executar os códigos normalmente a partir de um notebook. Outra ferramenta bastante usada para manipulação de dados em geral é o Spyder. Dentro dele podemos criar e executar os mesmos códigos de uma forma um pouco diferente graças aos seus recursos. Por hora, apenas entenda que será normal você ter de se familiarizar com diferentes ferramentas uma vez que algumas situações irão requerer mais ferramentas específicas.

Tratando-se do Spyder, este será o IDE que usaremos para praticamente tudo a partir daqui, graças a versatilidade de por meio dele podermos escrever nossas linhas de código, executá-las individualmente e em tempo real e visualizar os dados de forma mais intuitiva.



Abrindo o Spyder diretamente pelo seu atalho ou por meio da suíte Anaconda nos deparamos com sua tela inicial, basicamente o Spyder já vem pré-configurado de forma que possamos simplesmente nos dedicar ao código, talvez a única configuração que você deva fazer é para sua própria comodidade modificar o caminho onde serão salvos os arquivos.

A tela inicial possui a esquerda um espaço dedicado ao código, e a direita um visualizador de variáveis assim como um console que mostra em tempo real certas execuções de blocos de código.

Apenas como exemplo, rodando este mesmo código criado anteriormente, no Spyder podemos em tempo real fazer a análise das operações sobre as variáveis, assim como os resultados das mesmas via terminal.

Perceptron 1 Camada Otimizado.py*

```python
import numpy as np

entradas = np.array([1, 9, 5])
pesos = np.array([0.8, 0.1, 0])

def Soma(e,p):
  return e.dot(p)

s = Soma(entradas, pesos)

def stepFunction(Soma):
  if (s >= 1):
    return 1
  return 0

saida = stepFunction(s)

print(s)
print(saida)
```

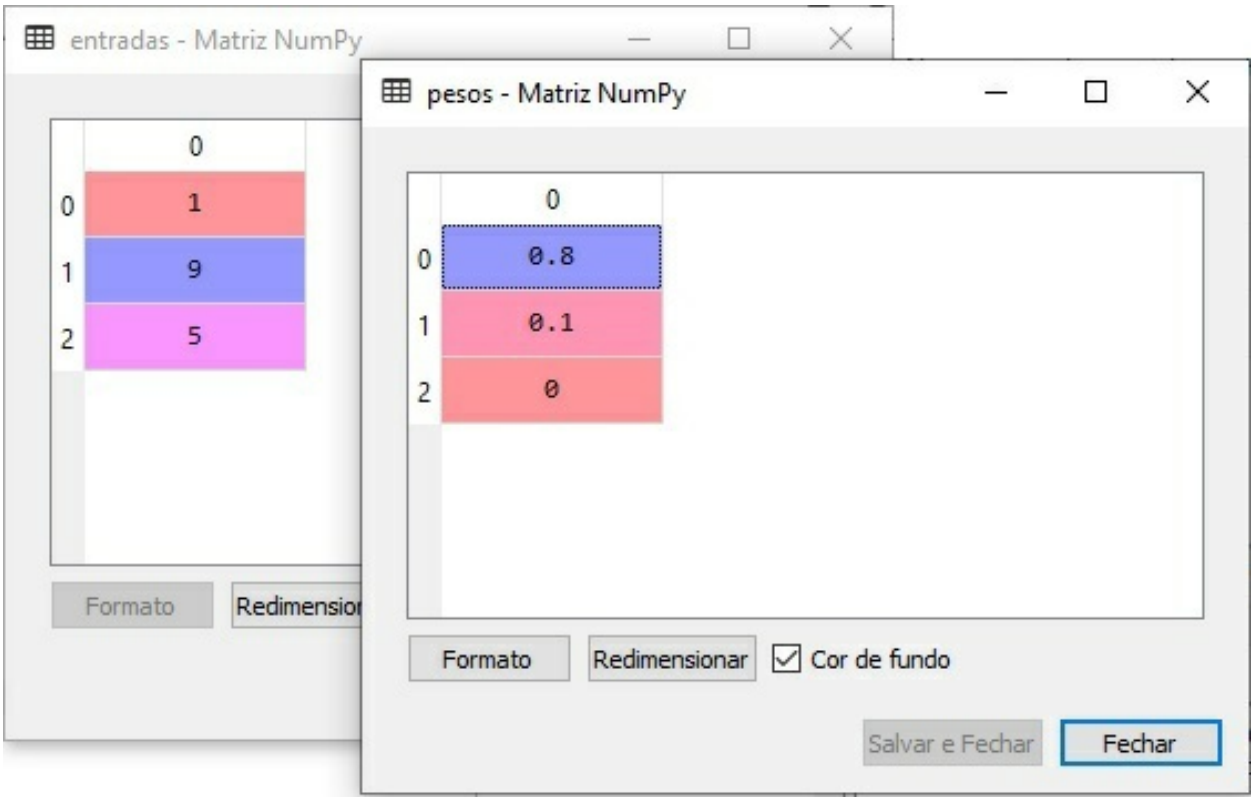
Explorador de Variáveis:

Explorador de variáveis

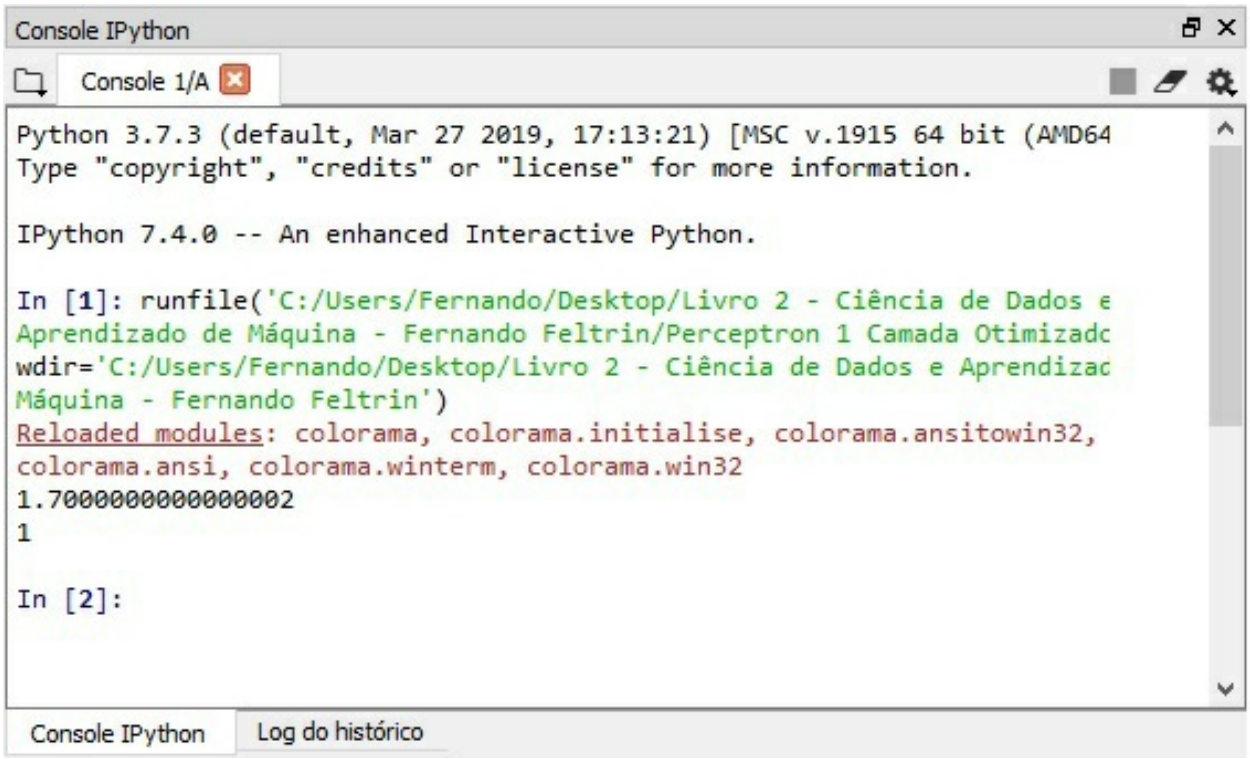
Nome	Tipo	Tamanho	Valor
entradas	int32	(3,)	[1 9 5]
pesos	float64	(3,)	[0.8 0.1 0.]
s	float64	1	1.7000000000000002
saida	int	1	1

Explorador de variáveis | Explorador de arquivos | Ajuda

Visualizando Variáveis:


entradas - Matriz NumPy
0
0
1
1
9
2
5
Formato
Redimensior
pesos - Matriz NumPy
0
0
0.8
1
0.1
2
0
Formato
Redimensionar
Cor de fundo
Salvar e Fechar
Fechar

Terminal:


Console IPython
Console 1/A
Python 3.7.3 (default, Mar 27 2019, 17:13:21) [MSC v.1915 64 bit (AMD64
Type "copyright", "credits" or "license" for more information.
IPython 7.4.0 -- An enhanced Interactive Python.
In [1]: runfile('C:/Users/Fernando/Desktop/Livro 2 - Ciência de Dados e
Aprendizado de Máquina - Fernando Feltrin/Perceptron 1 Camada Otimizadc
wdir='C:/Users/Fernando/Desktop/Livro 2 - Ciência de Dados e Aprendizac
Máquina - Fernando Feltrin')
Reloaded modules: colorama, colorama.initialise, colorama.ansitowin32,
colorama.ansi, colorama.winterm, colorama.win32
1.7000000000000002
1
In [2]:
Console IPython
Log do histórico

Perceptron de Uma Camada – Tabela AND

Entendidos os conceitos lógicos de o que é um perceptron, como os mesmos são um modelo para geração de processamento de dados para por fim ser o parâmetro de uma função de ativação. Hora de, também de forma prática, entender de fato o que é aprendizagem de máquina.

Novamente se tratando de redes neurais serem uma abstração as redes neurais biológicas, o mecanismo de aprendizado que faremos também é baseado em tal modelo. Uma das características de nosso sistema nervoso central é a de aprender criando padrões de conexões neurais para ativação de alguma função ou processamento de funções já realizadas (memória). Também é válido dizer que temos a habilidade de aprender e melhorar nosso desempenho com base em repetição de um determinado tipo de ação.

Da mesma forma, criaremos modelos computacionais de estruturas neurais onde, programaremos determinadas funções de forma manual (fase chamada supervisionada) e posteriormente iremos treinar tal rede para que identifique o que é certo, errado, e memorize este processo. Em suma, a aprendizagem de máquina ocorre quando uma rede neural atinge uma solução generalizada para uma classe de problemas.

Partindo para prática, vamos criar do zero uma rede neural que aprenderá o que é o mecanismo lógico de uma tabela AND. Operador lógico muito usado em problemas que envolvem simples tomada de decisão. Inicialmente, como é de se esperar, iremos criar uma estrutura lógica que irá processar os valores mas de forma errada, sendo assim, na chamada fase supervisionada, iremos treinar nossa rede para que ela aprenda o padrão referencial correto e de fato solucione o que é o processamento das entradas e as respectivas saídas de uma tabela AND.

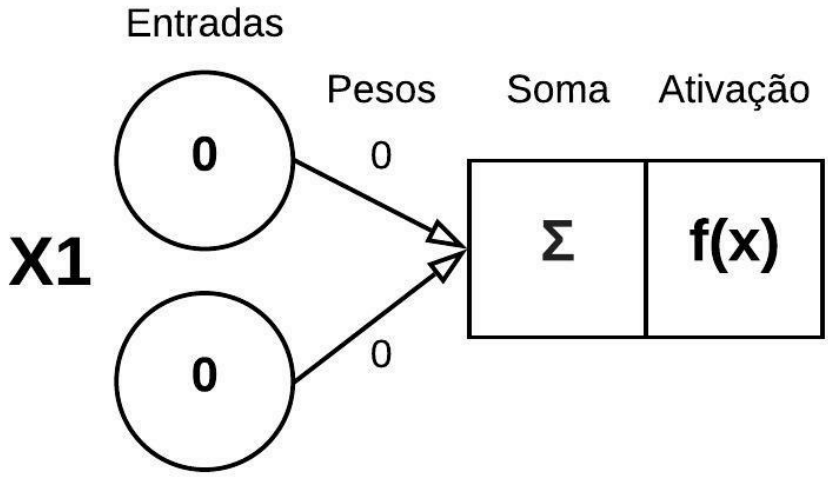
Tabela AND:

X1	X2

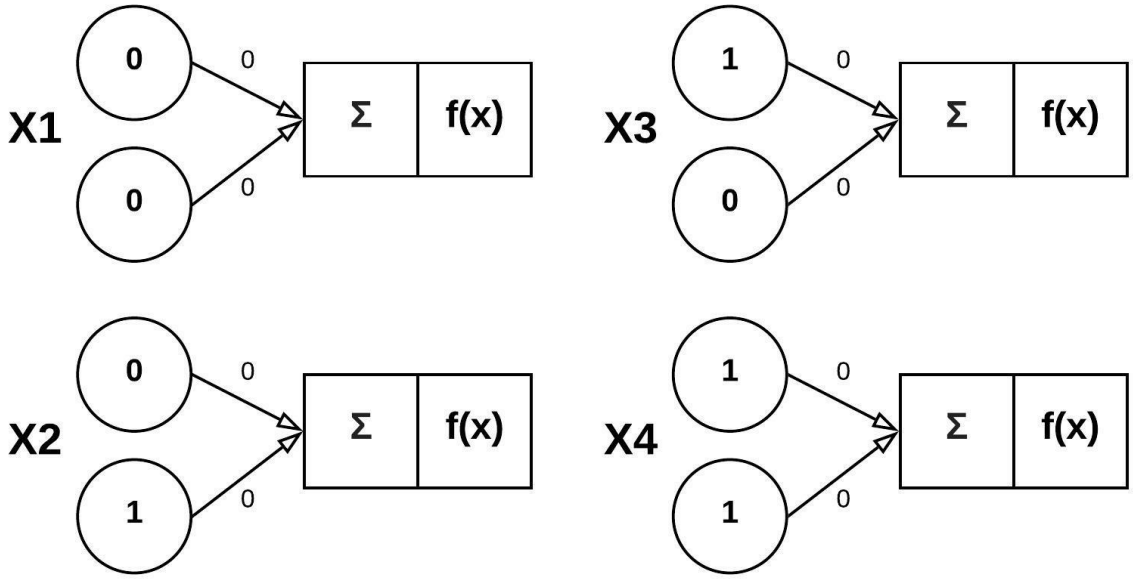
0	0	0
0	1	0
1	0	0
1	1	1

Todo mundo em algum momento de seu curso de computação teve de estudar operadores lógicos e o mais básico deles é o operador AND. Basicamente ele faz a relação entre duas proposições e apenas retorna VERDADEIRO caso os dois valores de entrada forem verdadeiros. A tabela acima nada mais é do que a tabela AND em operadores aritméticos, mas o mesmo conceito vale para True e False. Sendo 1 / True e 0 / False, apenas teremos como retorno 1 caso os dois operandos forem 1, assim como só será True se as duas proposições forem True.

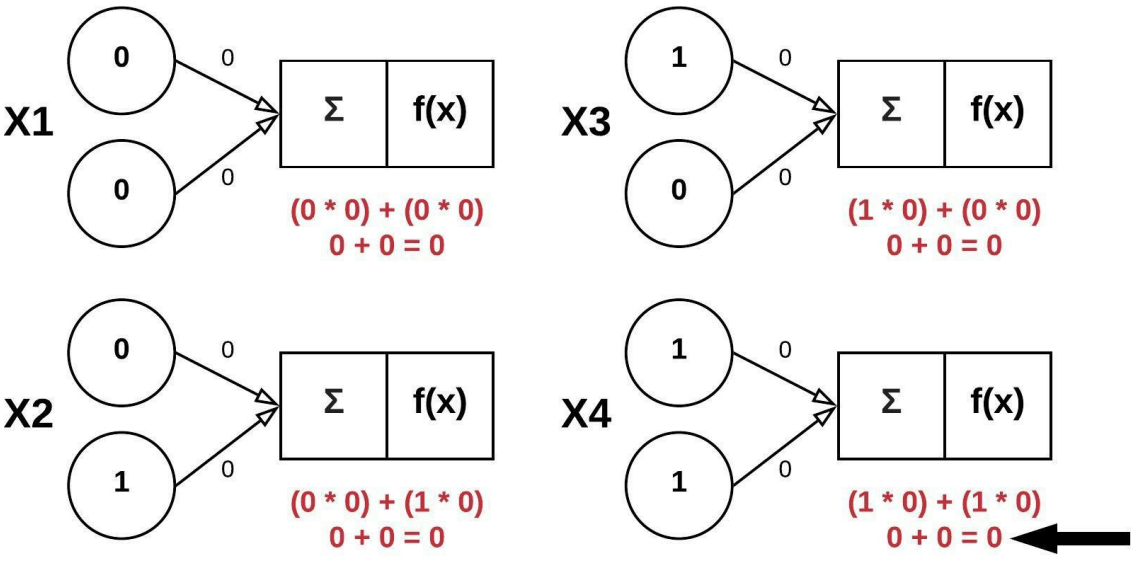
Dessa forma, temos 4 operadores a primeira coluna, mais 4 operadores na segunda camada e na última coluna os resultados das relações entre os mesmos. Logo, temos o suficiente para criar um modelo de rede neural que aprenderá essa lógica.



Basicamente o que faremos é a criação de 4 perceptrons onde cada um terá 2 neurônios de entrada, pesos que iremos atribuir manualmente, uma função soma e uma função de ativação.



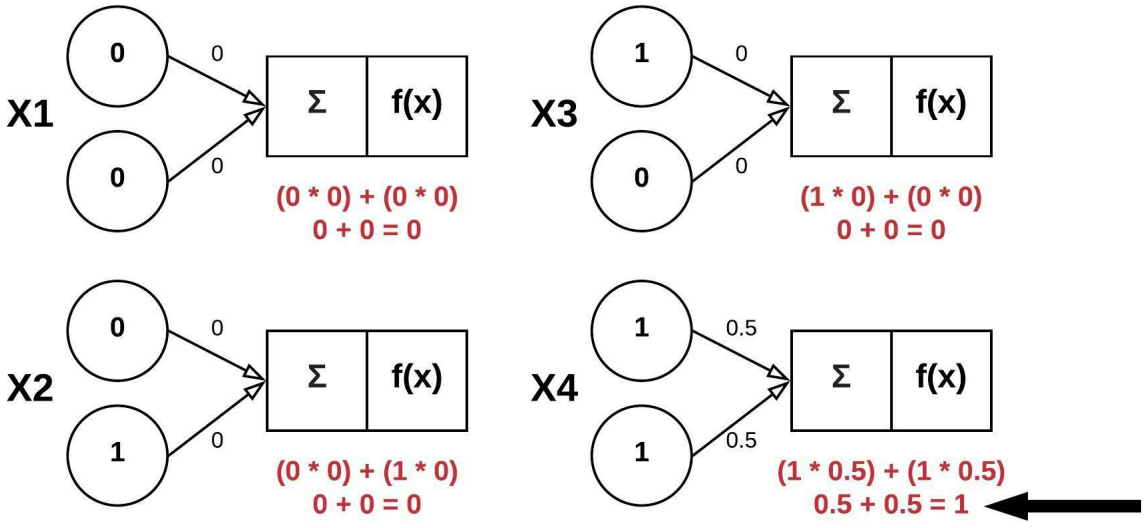
Montada a estrutura visual dos perceptrons, por hora, apenas para fins didáticos, hora de realizar as devidas operações de multiplicação entre os valores de entrada e seus respectivos pesos. Novamente, vale salientar que neste caso em particular, para fins de aprendizado, estamos iniciando essas operações com pesos zerados e definidos manualmente, esta condição não se repetirá em outros exemplos futuros.



Vamos ao modelo, criamos 4 estruturas onde X1 representa a primeira linha de nossa tabela AND (0 e 0), X2 que representa a segunda linha (0 e 1), X3 que representa a terceira linha (1 e 0) e por fim X4 que representa a quarta linha do operador (1 e 1).

Executando a função de soma dos mesmos, repare que para X1 a multiplicação das entradas pelos respectivos pesos e a soma destes resultou no valor 0, que era esperado nesse caso (Tabela AND - 0 e 0 = 0). O processo se repete exatamente igual aos perceptrons X2 e X3 onde a função soma resulta em 0. Porém, repare em X4, a multiplicação das entradas pelos respectivos pesos como manda a regra resultou em 0, mas sabemos que de acordo com a tabela verdade este resultado deveria ser 1. Isso se deu porque simplesmente os pesos 0 implicaram em uma multiplicação falha, qualquer valor multiplicado por 0 resulta em 0.

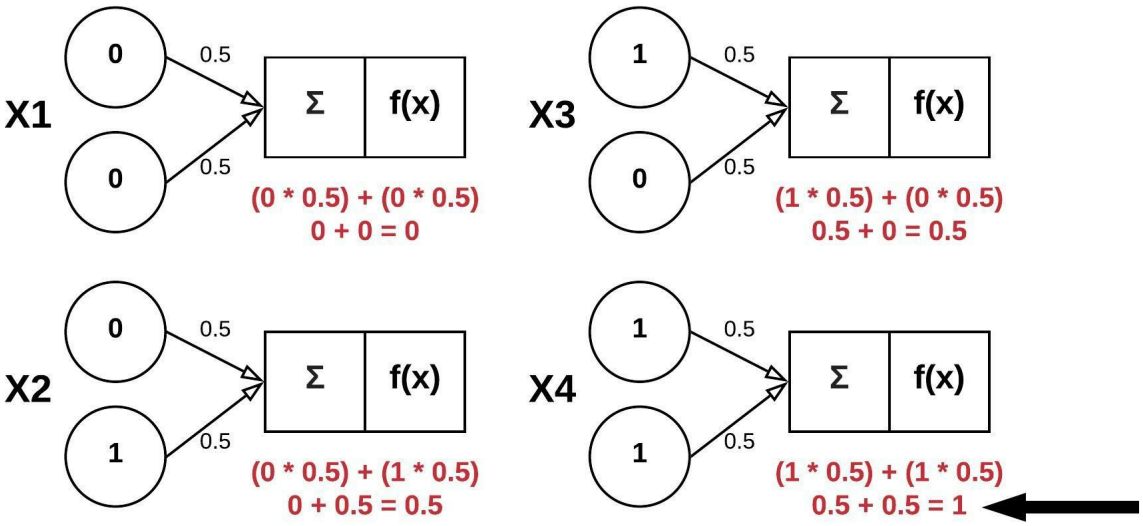
Precisamos então treinar nossa rede para que ao final do processo se chegue ao valor esperado (X4 = 1). Em redes neurais básicas como estas, o que faremos é o simples reajuste dos valores dos pesos e a repetição do processo para ver se o mesmo é corrigido.



Revisando apenas o perceptron X4, alterando os valores dos pesos de 0 para 0.5 e repetindo a função de soma, agora sim temos o valor esperado 1. (Tabela AND 1 e 1 = 1.

Uma vez encontrado o valor de pesos que solucionou o problema a

nível de X4, hora de aplicar este mesmo valor de pesos a toda rede neural e verificar se este valor funciona para tudo.



Aplicando o valor de peso 0.5 a todos neurônios e refeitos os devidos cálculos dentro de nossa função soma, finalmente temos os valores corrigidos. Agora seguindo a lógica explicada anteriormente, como os valores de X2 e X3 são menores que 1, na step function os mesmos serão reduzidos a 0, sendo apenas X4 o perceptron ativado nessa rede. o que era esperado de acordo com a tabela AND.

Em X1 0 e 0 = 0, em X2 0 e 1 = 0, em X3 1 e 0 = 0 e em X4 1 e 1 = 1.

Partindo para o Código:

```
[1]    1 import numpy as np
```

Por parte do código, todo processo sempre se inicia com a importação das bibliotecas e módulos necessários para que possamos fazer o uso deles. Nesse caso todas as operações que faremos serão operações nativas do Python ou funções internas da biblioteca Numpy. Para sua importação basta executar o comando import numpy, por convenção também as referenciaremos como np por meio do comando as, dessa forma, sempre que quisermos “chamar” alguma função sua basta usar np.

```
[2]  1 entradas = np.array([[0,0],[0,1],[1,0],[1,1]])
     2 saidas = np.array([0,0,0,1])
     3 pesos = np.array([0.0,0.0])
```

Em seguida criamos três variáveis que ficarão responsáveis por guardar os valores das entradas, saídas e dos pesos. Para isso declaramos uma nova variável de nome entradas que recebe como atributo uma array numpy através do comando np.array que por sua vez recebe como parâmetros os valores de entrada. Repare na sintaxe e também que para cada item dessa lista estão contidos os valores X1 e X2, ou seja, cada linha de nossa tabela AND. Da mesma forma criamos uma variável saidas que recebe uma array numpy com uma lista dos valores de saída. Por fim criamos uma variável pesos que recebe uma array numpy de valores iniciais zerados. Uma vez que aqui neste exemplo estamos explorando o que de fato é o aprendizado de máquina, usaremos pesos inicialmente zerados que serão corrigidos posteriormente.

```
[3]  1 taxaAprendizado = 0.5
```

Logo após criamos uma variável de nome taxaAprendizado que como próprio nome sugere, será a taxa com que os valores serão atualizados e testados novamente nesse perceptron. Você pode testar valores diferentes, menores, e posteriormente acompanhar o processo de aprendizagem.

```
[4]  1 def Soma(e,p):
     2     return e.dot(p)
```

Na sequência criamos a nossa função Soma, que tem como parâmetro as variáveis temporárias e e p, internamente ela simplesmente tem o comando de retornar o produto escalar de e sobre p, em outras palavras, a soma das multiplicações dos valores atribuídos as entradas com seus respectivos pesos.

```
[5]  1 s = Soma(entradas, pesos)
```

Criamos uma variável de nome s que como atributo recebe a função Soma que agora finalmente possui como parâmetro as variáveis entradas e pesos. Dessa forma, podemos instanciar a variável s em qualquer bloco de código que ela imediatamente executará a soma das entradas pelos pesos,

guardando seus dados internamente.

```
[6]
def stepFunction(soma):
  if (soma >= 1):
    return 1
  return 0
```

Seguindo com o código, criamos nossa stepFunction, em tradução livre, nossa função degrau, aquela que pegará os valores processados para consequentemente determinar a ativação deste perceptron ou não em meio a rede. Para isso definimos uma função stepFunction que tem como parâmetro a variável temporária soma. Internamente criamos uma estrutura condicional básica onde, se soma (o valor atribuído a ela) for igual ou maior que 1, retorna 1 (resultando na ativação), caso contrário retornará 0 (resultando na não ativação do mesmo).

```
[7]
def calculoSaida(reg):
  s = reg.dot(pesos)
  return stepFunction(s)
```

Assim como as outras funções, é preciso criar também uma função que basicamente pegará o valor processado e atribuído como saída e fará a comparação com o valor que já sabemos de saída da tabela AND, para que assim se possa identificar onde está o erro, subsequentemente realizar as devidas modificações para se se aprenda qual os valores corretos de saída. Simplesmente criamos uma função, agora de nome calculoSaida que tem como parâmetro a variável temporária reg, internamente ela tem uma variável temporária s que recebe como atributo o produto escalar de reg sobre pesos, retornando esse valor processado pela função degrau.

```
[8]
def aprendeAtualiza():
  erroTotal = 1
  while (erroTotal != 0):
    erroTotal = 0
    for i in range (len(saidas)):
      calcSaida = calculoSaida(np.array(entradas[i]))
      erro = abs(saidas[i] - calcSaida)
      erroTotal += erro
      for j in range(len(pesos)):
        pesos[j] = pesos[j] + (taxaAprendizado * entradas[i][j] * erro)
        print('Pesos Atualizados> ' + str(pesos[j]))
    print('Total de Erros: ' +str(erroTotal))
```

Muita atenção a este bloco de código, aqui será realizado o processo de aprendizado de máquina propriamente dito.

Inicialmente criamos uma função de nome aprendeAtualiza, sem parâmetros mesmo. Internamente inicialmente criamos uma variável de nome erroTotal, com valor 1 atribuído simbolizando que existe um erro a ser corrigido.

Em seguida criamos um laço de repetição que inicialmente tem como parâmetro a condição de que se erroTotal for diferente de 0 será executado o bloco de código abaixo. Dentro desse laço erroTotal tem seu valor zerado, na sequência uma variável i percorre todos valores de saidas, assim como uma nova variável calcSaida recebe os valores de entrada em forma de array numpy, como atributo de calculoSaida. É criada também uma variável de nome erro que de forma absoluta (sem considerar o sinal negativo) recebe como valor os dados de saidas menos os de calcSaida, em outras palavras, estamos fazendo nesta linha de código a diferença entre os valores de saída reais que já conhecemos na tabela AND e os que foram encontrados inicialmente pelo perpeptron. Para finalizar esse bloco de código erroTotal soma a si próprio o valor encontrado e atribuído a erro.

Na sequência é criado um novo laço de repetição onde uma variável j irá percorrer todos valores de pesos, em sequida pesos recebe como atributo seus próprios valores multiplicados pelos de taxaAprendizado (definido manualmente anteriormente como 0.5), multiplicado pelos valores de entradas e de erro. Para finalizar ele recebe dois comandos print( ) para que no console/terminal sejam exibidos os valores dos pesos atualizados assim como o total de erros encontrados.

```
[9]    1 aprendeAtualiza()
```

```
Pesos Atualizados> 0.0
Pesos Atualizados> 0.0
Pesos Atualizados> 0.0
Pesos Atualizados> 0.0
Pesos Atualizados> 0.0
Pesos Atualizados> 0.0
Pesos Atualizados> 0.5
Pesos Atualizados> 0.5
Total de Erros: 1
Pesos Atualizados> 0.5
Pesos Atualizados> 0.5
Pesos Atualizados> 0.5
Pesos Atualizados> 0.5
Pesos Atualizados> 0.5
Pesos Atualizados> 0.5
Pesos Atualizados> 0.5
Pesos Atualizados> 0.5
Total de Erros: 0
```

Por fim simplesmente executando a função aprendeAtualiza( ) podemos acompanhar via console/terminal o processo de atualização dos pesos (aprendizado de máquina) e o log de quando não existem mais erros, indicando que tal perceptron finalmente foi treinado e aprendeu de fato a interpretar uma tabela AND, ou em outras palavras, reconhecer o padrão correto de suas entradas e saídas.

Código Completo:

```
[1]
import numpy as np

entradas = np.array([[0,0],[0,1],[1,0],[1,1]])
saidas = np.array([0,0,0,1])
pesos = np.array([0.0,0.0])

taxaAprendizado = 0.5

def Soma(e,p):
    return e.dot(p)

s = Soma(entradas, pesos)

def stepFunction(soma):
  if (soma >= 1):
    return 1
  return 0

def calculoSaida(reg):
  s = reg.dot(pesos)
  return stepFunction(s)

def aprendeAtualiza():
  erroTotal = 1
  while (erroTotal != 0):
    erroTotal = 0
    for i in range (len(saidas)):
      calcSaida = calculoSaida(np.array(entradas[i]))
      erro = abs(saidas[i] - calcSaida)
      erroTotal += erro
      for j in range(len(pesos)):
        pesos[j] = pesos[j] + (taxaAprendizado * entradas[i][j] * erro)
        print('Pesos Atualizados> ' + str(pesos[j]))
    print('Total de Erros: ' +str(erroTotal))

aprendeAtualiza()
```

Usando Spyder:

Perceptron 1 Camada Tabela AND PRONTO.py*

```python
import numpy as np

entradas = np.array([[0,0],[0,1],[1,0],[1,1]])
saidas = np.array([0,0,0,1])
pesos = np.array([0.0,0.0])

taxaAprendizado = 0.5

def Soma(e,p):
    return e.dot(p)

s = Soma(entradas, pesos)

def stepFunction(soma):
  if (soma >= 1):
    return 1
  return 0

def calculoSaida(reg):
  s = reg.dot(pesos)
  return stepFunction(s)

def aprendeAtualiza():
  erroTotal = 1
  while (erroTotal != 0):
    erroTotal = 0
    for i in range (len(saidas)):
      calcSaida = calculoSaida(np.array(entradas[i]))
      erro = abs(saidas[i] - calcSaida)
      erroTotal += erro
      for j in range(len(pesos)):
        pesos[j] = pesos[j] + (taxaAprendizado * entradas[i][j] * erro)
        print('Pesos Atualizados> ' + str(pesos[j]))
    print('Total de Erros: ' +str(erroTotal))

aprendeAtualiza()
```

Explorador de Variáveis:

Explorador de variáveis

Nome	Tipo	Tamanho	Valor
entradas	int32	(4, 2)	[[0 0] [0 1]
pesos	float64	(2,)	[0.5 0.5]
s	float64	(4,)	[0. 0. 0. 0.]
saidas	int32	(4,)	[0 0 0 1]
taxaAprendizado	float	1	0.5

Console:

Console IPython

Console 1/A

```
Pesos Atualizados> 0.0
Pesos Atualizados> 0.0
Pesos Atualizados> 0.0
Pesos Atualizados> 0.0
Pesos Atualizados> 0.5
Pesos Atualizados> 0.5
Total de Erros: 1
Pesos Atualizados> 0.5
Pesos Atualizados> 0.5
Pesos Atualizados> 0.5
Pesos Atualizados> 0.5
Pesos Atualizados> 0.5
Pesos Atualizados> 0.5
Pesos Atualizados> 0.5
Pesos Atualizados> 0.5
Total de Erros: 0
```

Perceptron Multicamada – Tabela XOR

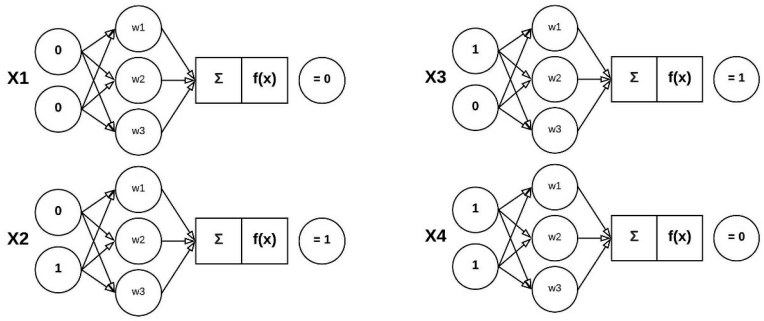
Entendida a lógica do processo de aprendizado de máquina na prática, treinando um perceptron para que aprenda um operador lógico AND, hora de aumentar levemente a complexidade. Como dito anteriormente, gradualmente vamos construindo essa bagagem de conhecimento. O principal diferencial deste capítulo em relação ao anterior é que, em uma tabela AND temos um problema linearmente separável, onde a resposta (saída) era basicamente 1 ou 0, pegando o exemplo de uma tabela XOR temos o diferencial de que este operador lógico realiza a operação lógica entre dois operandos, que resulta em um valor lógico verdadeiro se e somente se o número de operandos com valor verdadeiro for ímpar, dessa forma temos um problema computacional onde, a nível de perceptron, teremos de treinar o mesmo para descobrir a probabilidade desta operação ser verdadeira.

Em outras palavras, teremos de identificar e treinar nosso perceptron para que realize mais de uma camada de correção de pesos, afim de identificar a probabilidade correta da relação entre as entradas e as saídas da tabela XOR que por sua vez possui mais de um parâmetro de ativação. O fato de trabalharmos a partir daqui com uma camada a mais de processamento para rede, camada essa normalmente chamada de camada oculta, permite que sejam aplicados “filtros” que nos ajudam a identificar, treinar e aprender sob nuances de valores muito menores do que os valores brutos do exemplo anterior.

Tabela XOR:

X1	X2	
0	0	0
0	1	1
1	0	1
1	1	0

Estrutura Lógica:



Diretamente ao código:

Perceptron Multicamada XOR.py

```
1 import numpy as np
2
```

Como de costume, todo processo se inicia com a importação das bibliotecas e módulos que utilizaremos ao longo de nossa rede. Agora trabalhando diretamente com o Spyder, podemos em tempo real executar linhas ou blocos de código simplesmente os selecionando com o mouse e aplicando o comando Ctrl + ENTER. Para importação da biblioteca Numpy basta executar o comando import como exemplificado acima.

```
1 entradas = np.array([[0],[1],[1],[0]])
2                      [1,1]])
3                      [1,0],
4                      [0,1],
5 saidas = np.array([[0,0],
```

Em seguida criamos as variáveis entradas e saidas que como seus nomes já sugerem, recebem como atributos os respectivos dados da tabela XOR, em forma de array numpy, assim, uma vez vetorizados, poderemos posteriormente aplicar funções internas desta biblioteca.

Explorador de variáveis

Nome	Tipo	Tamanho	Valor
entradas	int32	(4, 2)	[[0 0] [0 1]
saidas	int32	(4, 1)	[[0] [1]

Executando esse bloco de código (via Ctrl + ENTER) podemos ver que ao lado direito da interface, no Explorador de variáveis são criadas as

respectivas variáveis.



Clicando duas vezes sobre as mesmas podemos as explorar de forma visual, em forma de matrizes como previsto, lado a lado compondo a tabela XOR.

```
 9 pesos0 = np.array([[-0.424, -0.740, -0.961],
10                   [0.358, -0.577, -0.469]])
11 pesos1 = np.array([[-0.017], [-0.893], [0.148]])
```

Em seguida criamos as variáveis dedicadas a guardar os valores dos pesos, aqui, novamente apenas para fins de exemplo, estamos iniciando essas variáveis com valores aleatórios.

Nome	Tipo	Tamanho	Valor
entradas	int32	(4, 2)	[[0 0] [0 1]
pesos0	float64	(2, 3)	[[-0.424 -0.74 -0.961] [0.358 -0.577 -0.469]]
pesos1	float64	(3, 1)	[[-0.017] [-0.893]
saidas	int32	(4, 1)	[[0] [1]

Se a sintaxe estiver correta, ao selecionar e executar este bloco de código podemos ver no explorador de variáveis que tais variáveis foram corretamente criadas.

```
13 ntreinos = 100
14 taxaAprendizado = 0.3
15 momentum = 1
```

Em seguida criamos estas três variáveis auxiliares que nos serão úteis posteriormente. A variável ntreinos, que aqui recebe como atributo o valor 100, diz respeito ao parâmetro de quantas vezes a rede será executada para que haja o devido ajuste dos pesos. Da mesma forma a variável taxaAprendizado aqui com valor 0.3 atribuído é o parâmetro para de quantos em quantos números os pesos serão modificados durante o processo de aprendizado, esse parâmetro, entendido em outras literaturas como a velocidade em que o algoritmo irá aprender, conforme o valor pode inclusive fazer com que o algoritmo piore sua eficiência ou entre em um loop onde fica estagnado em um ponto, logo, é um parâmetro a ser testado com outros valores e comparar a eficiência dos resultados dos mesmos. Por fim momentum é um parâmetro padrão, uma constante na fórmula de correção da margem de erro, nesse processo de atualização de pesos, aqui, para este exemplo, segue com o valor padrão 1 atribuído, mas pode ser testado com outros valores para verificar se as últimas alterações surtiram melhoria da eficiência do algoritmo.

```
17 def sigmoid(soma):
18     return 1 / (1 + np.exp(-soma))
```

Logo após criamos nossa Função Sigmoide, que neste modelo, substitui a Função Degrau que utilizamos anteriormente. Existem diferentes funções de ativação, as mais comuns, Degrau e Sigmoide utilizaremos com frequência ao longo dos demais exemplos. Em suma, a grande diferença entre elas é que a Função Degrau retorna valores absolutos 0 ou 1 (ou valores definidos pelo usuário, mas sempre neste padrão binário, um ou outro), enquanto a Função Sigmoide leva em consideração todos valores intermediários entre 0 e 1, dessa forma, conseguimos verificar a probabilidade de um dado valor se aproximar de 0 ou se aproximar de 1, de acordo com suas características.

Para criar a função Sigmoide, simplesmente definimos sigmoid( ) que recebe como parâmetro a variável temporária soma, internamente ela simplesmente retorna o valor 1 dividido pela soma de 1 pela exponenciação de soma, o que na prática nos retornará um valor float entre 0 e 1 (ou seja, um número com casas decimais).

```
for i in range(ntreinos):
    camadaEntrada = entradas
    somaSinapse0 = np.dot(camadaEntrada, pesos0)
    camadaOculta = sigmoid(somaSinapse0)

    somaSinapse1 = np.dot(camadaOculta, pesos1)
    camadaSaida = sigmoid(somaSinapse1)

    erroCamadaSaida = saidas - camadaSaida
    mediaAbsoluta = np.mean(np.abs(erroCamadaSaida))
```

Prosseguindo criamos o primeiro bloco de código onde de fato ocorre a interação entre as variáveis. Criamos um laço de repetição que de acordo com o valor setado em ntreinos executa as seguintes linhas de código. Inicialmente é criada uma variável temporária camadaEntrada que recebe como atributo o conteúdo de entradas, em seguida é criada uma variável somaSinapse0, que recebe como atributo o produto escalar (soma das multiplicações) de camadaEntrada por pesos0. Apenas relembrando, uma sinapse é a termologia da conexão entre neurônios, aqui, trata-se de uma variável que faz a interligação entre os nós da camada de entrada e da camada oculta. Em seguida é criada uma variável camadaOculta que recebe como atributo a função sigmoid( ) que por sua vez tem como parâmetro o valor resultante dessa operação sobre somaSinapse0.

Da mesma forma é necessário criar a estrutura de interação entre a camada oculta e a camada de saída. Para isso criamos a variável somaSinapse1 que recebe como atributo o produto escalar entre camadaOculta e pesos1, assim como anteriormente fizemos, é criada uma variável camadaSaida que por sua vez recebe como atributo o valor gerado pela função sigmoid( ) sobre somaSinapse1.

Por fim são criados por convenção duas variáveis que nos mostrarão parâmetros para avaliar a eficiência do processamento de nossa rede. erroCamadaSaida aplica a fórmula de erro, fazendo a simples diferença entre os valores de saidas (valores que já conhecemos) e camadaSaida (valores encontrados pela rede). Posteriormente criamos a variável mediaAbsoluta que simplesmente, por meio da função .mean( ) transforma os dados de erroCamadaSaida em um valor percentual (Quantos % de erro existe sobre nosso processamento).

Nome	Tipo	Tamanho	Valor
camadaEntrada	int32	(4, 2)	[[0 0] [0 1]]
camadaOculta	float64	(4, 3)	[[0.5 0.5 0.5] [0.5885562 0.35962319 0.38485296 ...
camadaSaida	float64	(4, 1)	[[0.40588573] [0.43187857]]
entradas	int32	(4, 2)	[[0 0] [0 1]]
erroCamadaSaida	float64	(4, 1)	[[-0.40588573] [0.56812143]]
i	int	1	99
mediaAbsoluta	float64	1	0.49880848923713045

Selecionando e executando esse bloco de código são criadas as referentes variáveis, das camadas de processamento assim como as variáveis auxiliares. Podemos clicando duas vezes sobre as mesmas visualizar seu conteúdo. Por hora, repare que neste primeiro processamento é criada a variável mediaAbsoluta que possui valor 0.49, ou seja 49% de erro em nosso processamento, o que é uma margem muito grande. Pode ficar tranquilo que valores altos nesta etapa de execução são perfeitamente normais, o que faremos na sequência é justamente trabalhar a realizar o aprendizado de máquina, fazer com que nossa rede identifique os padrões corretos e reduza esta margem de erro ao menor valor possível.

```
31 def sigmoideDerivada(sig):
32     return sig * (1-sig)
33 sigDerivada = sigmoid(0.5)
34 sigDerivada1 = sigmoideDerivada(sigDerivada)
```

Dando sequência criamos nossa função sigmoideDerivada( ) que como próprio nome sugere, faz o cálculo da derivada, que também nos será útil na fase de aprendizado de máquina. Basicamente o que fazemos é definir uma função sigmoideDerivada( ) que tem como parâmetro a variavel temporária sig. Internamente é chamada a função que retorna o valor obtido pela multiplicação de sig (do valor que estiver atribuído a esta variável) pela multiplicação de 1 menos o próprio valor de sig.

Em seguida, aproveitando o mesmo bloco de código dedicado a esta etapa, criamos duas variáveis sigDerivada, uma basicamente aplica a função sigmoid( ) com o valor de 0.5, a outra, aplica a própria função sigmoideDerivada( ) sobre sigDerivada.

```
derivadaSaida = sigmoideDerivada(camadaSaida)
deltaSaida = erroCamadaSaida * derivadaSaida
```

Da mesma forma, seguimos criando mais variáveis que nos auxiliarão a acompanhar o processo de aprendizado de máquina, dessa vez criamos uma variável derivadaSaida que aplica a função sigmoideDerivada( ) para a camadaSaida, por fim criamos a variável deltaSaida que recebe como atributo o valor da multiplicação entre erroCamadaSaida e derivadaSaida.

Vale lembrar que o que nomeamos de delta em uma rede neura normalmente se refere a um parâmetro usado como referência para correto ajuste da descida do gradiente. Em outras palavras, para se realizar um ajuste fino dos pesos e consequentemente conseguir minimizar a margem de erro dos mesmos, são feitos internamente uma série de cálculos para que esses reajustes sejam feitos da maneira correta, parâmetros errados podem prejudicar o reconhecimento dos padrões corretos por parte da rede neural.



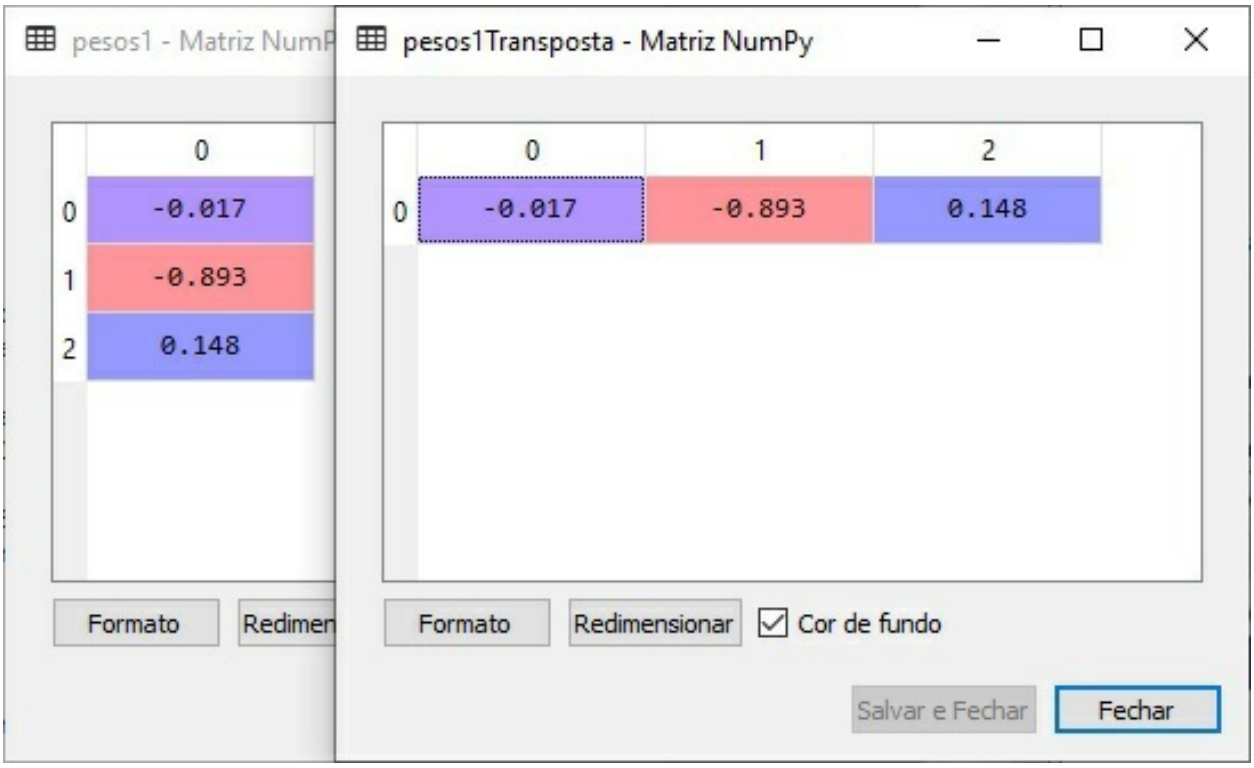
Podemos sempre que quisermos, fazer a visualização das variáveis por meio do explorador, aqui, os dados encontrados executando a fórmula do cálculo da derivada sobre os valores de saída assim como os valores encontrados para o delta, em outras palavras, pelo sinal do delta podemos entender como (em que direção no grafo) serão feitos os reajustes dos pesos,

uma vez que individualmente eles poder ter seus valores aumentados ou diminuídos de acordo com o processo.

```
39 pesos1Transposta = pesos1.T
40 deltaSaidaXpesos = deltaSaida.dot(pesos1Transposta)
41 deltaCamadaOculta = deltaSaidaXpesos * sigmoideDerivada(camadaOculta)
```

Dando sequência, teremos de fazer um reajuste de formado de nossos dados em suas devidas matrizes para que as operações aritméticas sobre as mesmas possam ser realizadas corretamente. Raciocine que quando estamos trabalhando com vetores e matrizes temos um padrão de linhas e colunas que pode ou não ser multiplicável. Aqui nesta fase do processo teremos inconsistência de formatos de nossas matrizes de pesos, sendo necessário realizar um processo chamado Matriz Transposta, onde transformaremos linhas em colunas e vice versa de forma que possamos realizar as operações e obter valores corretos.

Inicialmente criamos uma variável pesos1Transposta que recebe como atributo pesos1 com aplicação da função .T( ), função interna da biblioteca Numpy que realizará essa conversão. Em seguida criamos uma variável de nome deltaSaidaXpesos que recebe como atributo o produto escalar de deltaSaida por pesos1Transposta. Por fim aplicamos novamente a fórmula do delta, dessa vez para camada oculta, multiplicando deltaSaidaXpesos pelo resultado da função sigmoide sob o valor de camadaOculta.



Apenas visualizando a diferença entre pesos1 e pesos1Transposta (colunas transformadas em linhas).

Se você se lembra do algoritmo explicado em capítulos anteriores verá que até este momento estamos trabalhando na construção da estrutura desse perceptron, assim como sua alimentação com valores para seus nós e pesos. A esta altura, obtidos todos valores iniciais, inclusive identificando que tais valores resultam em erro se comparado ao esperado na tabela XOR, é hora de darmos início ao que para alguns autores é chamado de backpropagation, ou seja, realizar de fato os reajustes dos pesos e o reprocessamento do perceptron, repetindo esse processo até o treinar de fato para a resposta correta.

Todo processo realizado até o momento é normalmente tratado na literatura como feed forward, ou seja, partindo dos nós de entrada fomos alimentando e processando em sentido a saída, a ativação desse perceptron, a etapa de backpropagation como o nome já sugere, retroalimenta o perceptron, faz o caminho inverso ao feed forward, ou em certos casos, retorna ao ponto inicial e refaz o processamento a partir deste.

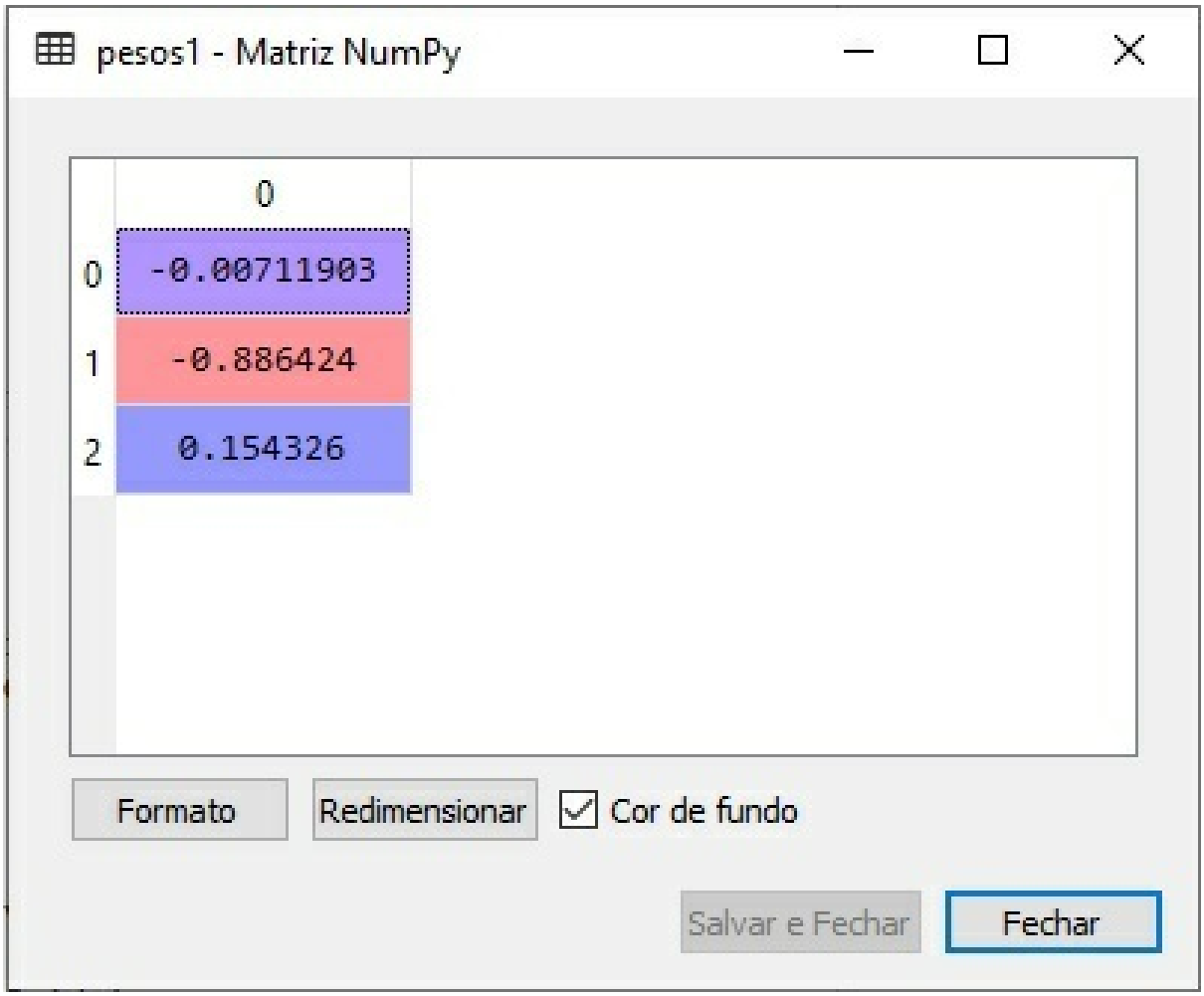
O processo de backpropagation inclusive tem uma fórmula a ser aplicada para que possamos entender a lógica desse processo como operações sobre

nossas variáveis.

peso(n + 1) = (peso(n) + momento) + (entrada * delta * taxa de aprendizagem)

```
43 camadaOcultaTransposta = camadaOculta.T
44 pesos3 = camadaOcultaTransposta.dot(deltaSaida)
45 pesos1 = (pesos1 * momentum) + (pesos3 * taxaAprendizado)
```

Da mesma forma como fizemos anteriormente, criamos uma variável camadaOcultaTransposta que simplesmente recebe a camadaOculta de forma transposta por meio da função .T( ). Logo após usamos a variável pesos3 que recebe como atributo o produto escalar de camadaOcultaTransposta pelo deltaSaida. Finalmente, fazemos o processo de atualização dos valores de pesos1, atribuindo ao mesmo os valores atualizados de pesos1 multiplicado pelo momentum somado com os valores de pesos3 multiplicados pelo parâmetro de taxaAprendizado.



Executando esse bloco de código você pode ver que de fato houve a atualização dos valores de pesos1. Se nesse processo não houver nenhuma inconsistência ou erro de sintaxe, após a execução desses blocos de código temos a devida atualização dos pesos da camada oculta para a camada de saída.

```
47 camadaEntradaTransposta = camadaEntrada.T
48 pesos4 = camadaEntradaTransposta.dot(deltaCamadaOculta)
49 pesos0 = (pesos0 * momentum) + (pesos4 * taxaAprendizado)
```

O mesmo processo é realizado para atualização dos valores de pesos0. Ou seja, agora pela lógica backpropagation voltamos mais um passo ao início do perceptron, para que possamos fazer as devidas atualizações a partir da camada de entrada.

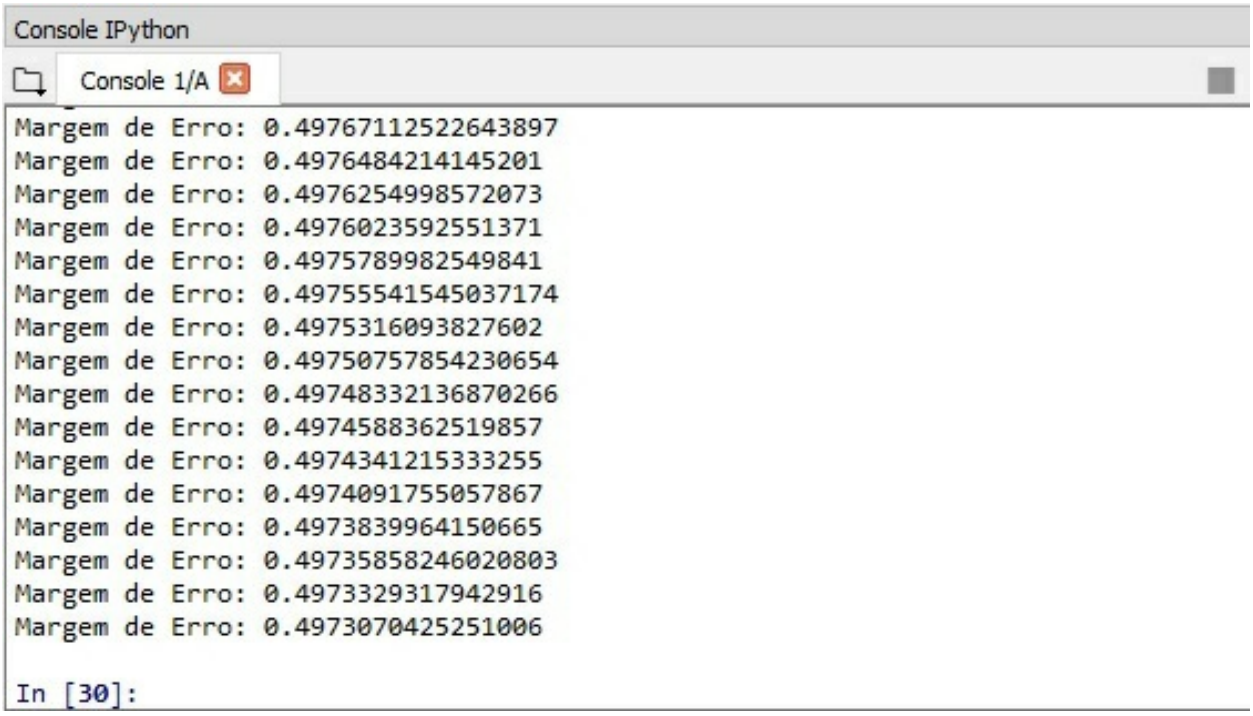
```
25 for i in range(ntreinos):
26     camadaEntrada = entradas
27     somaSinapse0 = np.dot(camadaEntrada, pesos0)
28     camadaOculta = sigmoid(somaSinapse0)
29 
30     somaSinapse1 = np.dot(camadaOculta, pesos1)
31     camadaSaida = sigmoid(somaSinapse1)
32 
33     erroCamadaSaida = saidas - camadaSaida
34     mediaAbsoluta = np.mean(np.abs(erroCamadaSaida))
35 
36     derivadaSaida = sigmoideDerivada(camadaSaida)
37     deltaSaida = erroCamadaSaida * derivadaSaida
38 
39     pesos1Transposta = pesos1.T
40     deltaSaidaXpesos = deltaSaida.dot(pesos1Transposta)
41     deltaCamadaOculta = deltaSaidaXpesos * sigmoideDerivada(camadaOculta)
42 
43     camadaOcultaTransposta = camadaOculta.T
44     pesos3 = camadaOcultaTransposta.dot(deltaSaida)
45     pesos1 = (pesos1 * momentum) + (pesos3 * taxaAprendizado)
46 
47     camadaEntradaTransposta = camadaEntrada.T
48     pesos4 = camadaEntradaTransposta.dot(deltaCamadaOculta)
49     pesos0 = (pesos0 * momentum) + (pesos4 * taxaAprendizado)
```

Lembrando que Python é uma linguagem interpretada e de forte indentação, o que em outras palavras significa que o interpretador lê e executa linha após linha em sua sequência normal, e executa blocos de código quando estão uns dentro dos outros de acordo com sua indentação. Sendo assim, todo código praticamente pronto, apenas reorganizamos o mesmo para que não haja problema de interpretação. Dessa forma, todos

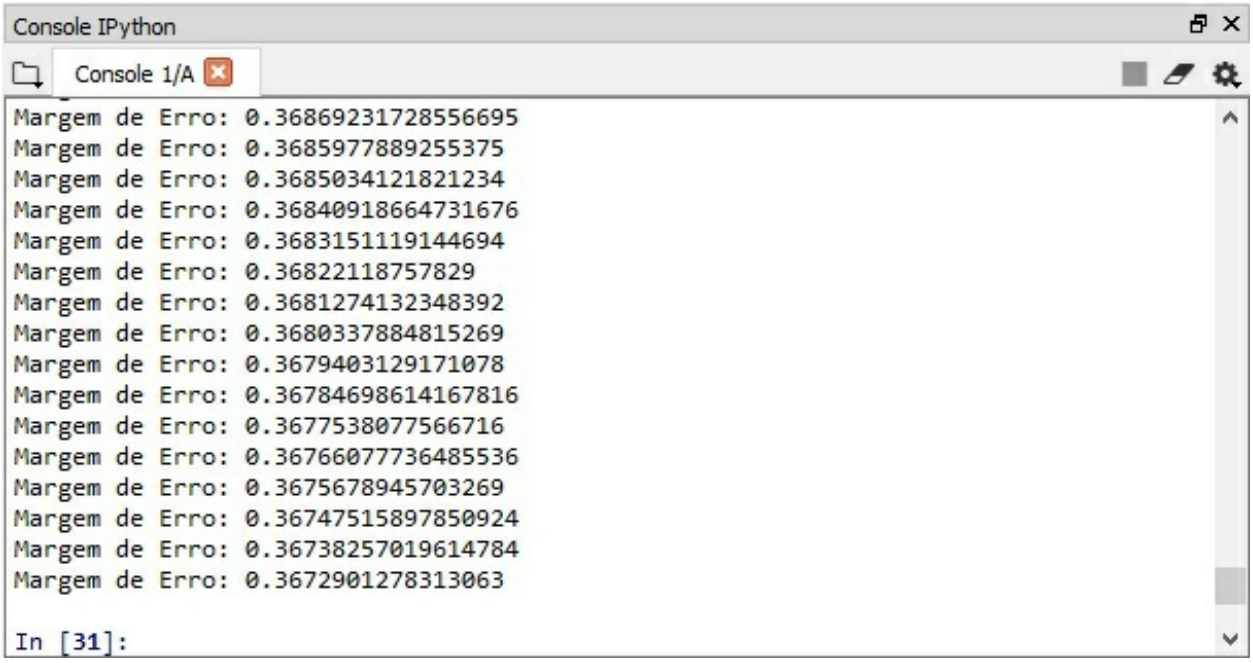
blocos dedicados aos ajustes dos pesos ficam dentro daquele nosso laço de repetição, para que possamos executá-los de acordo com os parâmetros que iremos definir para as épocas e taxa de aprendizado.

```
51    print('Margem de Erro: ' +str(mediaAbsoluta))
```

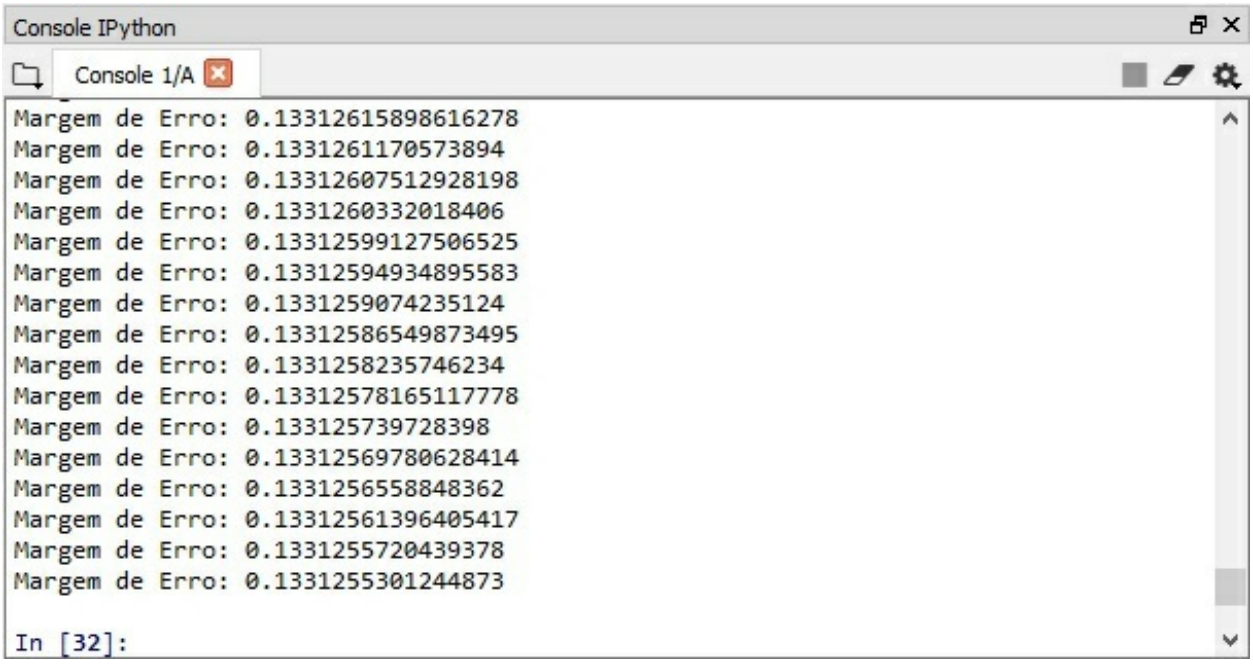
Finalmente criamos uma função print( ) dedicada a nos mostrar via console a margem de erro deste processamento. Agora selecionando e executando todo o código, podemos acompanhar via console o aprendizado de máquina acontecendo. Lembrando que inicialmente havíamos definido o número de vezes a serem executado o código inicialmente como 100. Esse parâmetro pode ser livremente modificado na variável ntreinos.



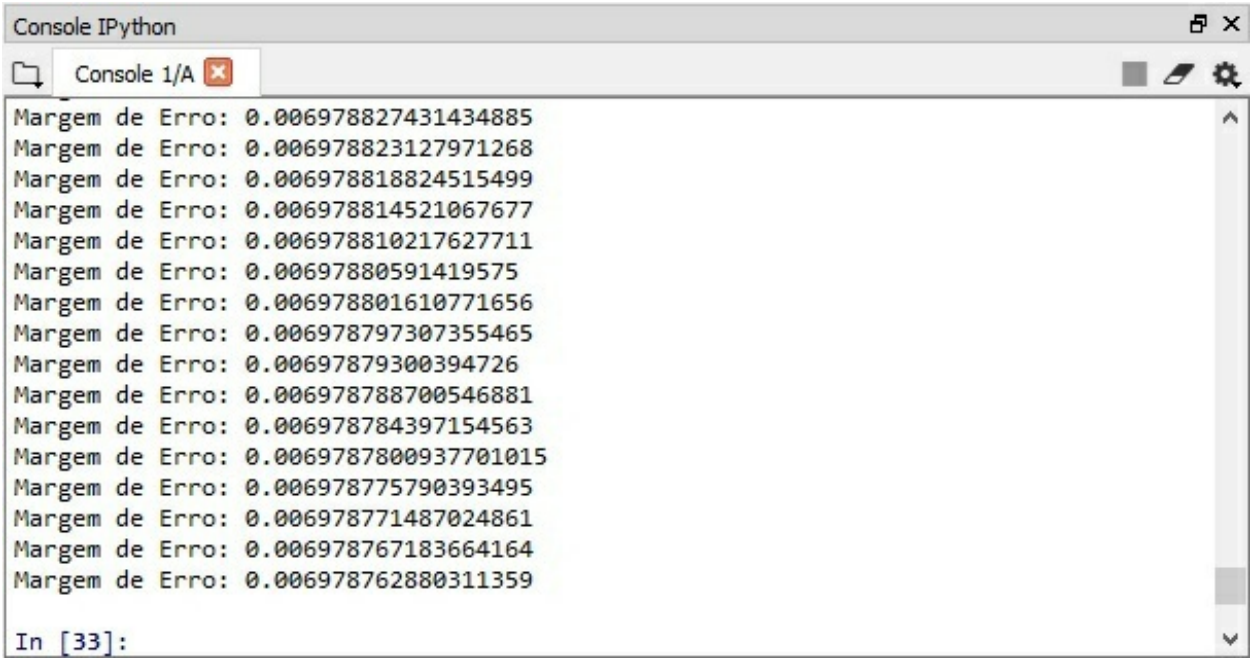
Executando 100 vezes a margem de erro cai para 0.497.



Executando 1000 vezes, a margem de erro cai para 0.367.



Executando 100000 vezes, a margem de erro cai para 0.133.



Por fim, rede executada 1000000 de vezes, a margem de erro cai para 0.006, em outras palavras, agora a rede está treinada para identificar os padrões de entrada e saída de uma tabela XOR, com 0,006% de margem de erro (ou 99,994% de precisão). Lembrando que em nossa amostragem inicial tínhamos 51% de acerto, agora quase 100% de acerto é um resultado excelente.

Código Completo:

```python
import numpy as np

entradas = np.array([[0,0],
                     [0,1],
                     [1,0],
                     [1,1]])
saidas = np.array([[0],[1],[1],[0]])

pesos0 = np.array([[-0.424, -0.740, -0.961],
                   [0.358, -0.577, -0.469]])
pesos1 = np.array([[-0.017], [-0.893], [0.148]])

ntreinos = 1000000
taxaAprendizado = 0.3
momentum = 1

def sigmoid(soma):
    return 1 / (1 + np.exp(-soma))

def sigmoideDerivada(sig):
    return sig * (1-sig)
sigDerivada = sigmoid(0.5)
sigDerivada1 = sigmoideDerivada(sigDerivada)
```

```python
for i in range(ntreinos):
    camadaEntrada = entradas
    somaSinapse0 = np.dot(camadaEntrada, pesos0)
    camadaOculta = sigmoid(somaSinapse0)

    somaSinapse1 = np.dot(camadaOculta, pesos1)
    camadaSaida = sigmoid(somaSinapse1)

    erroCamadaSaida = saidas - camadaSaida
    mediaAbsoluta = np.mean(np.abs(erroCamadaSaida))

    derivadaSaida = sigmoideDerivada(camadaSaida)
    deltaSaida = erroCamadaSaida * derivadaSaida

    pesos1Transposta = pesos1.T
    deltaSaidaXpesos = deltaSaida.dot(pesos1Transposta)
    deltaCamadaOculta = deltaSaidaXpesos * sigmoideDerivada(camadaOculta)

    camadaOcultaTransposta = camadaOculta.T
    pesos3 = camadaOcultaTransposta.dot(deltaSaida)
    pesos1 = (pesos1 * momentum) + (pesos3 * taxaAprendizado)

    camadaEntradaTransposta = camadaEntrada.T
    pesos4 = camadaEntradaTransposta.dot(deltaCamadaOculta)
    pesos0 = (pesos0 * momentum) + (pesos4 * taxaAprendizado)

    print('Margem de Erro: ' +str(mediaAbsoluta))
```

REDES NEURAIS ARTIFICIAIS

Processamento de Linguagem Natural – Minerando Emoções

Uma das aplicações mais interessantes quando se trata de aprendizado de máquina simples é o chamado processamento de linguagem natural, é o processo de reconhecimento de padrões a partir de textos, de forma que possamos treinar uma rede neural a minerar informações a partir dos mesmos.

Áreas como, por exemplo, direito, tem feito o uso de tais modelos como ferramenta para extração de informações relevantes a partir de textos jurídicos enormes, agilizando dessa forma seus processos. Importante salientar que o uso de redes neurais para tal fim jamais irá substituir os profissionais deste meio, até porque em direito sempre a palavra final e de maior peso se dá pela interpretação do juiz quanto ao caso.

Obviamente um computador não possui nenhuma capacidade de abstração para entender de fato uma linguagem natural, porém, graças a gramática das linguagens, existem formas padronizadas de se transliterar qualquer coisa em forma de texto. A máquina por sua vez, terá plenas condições de identificar e aprender os padrões explícitos da linguagem, tendo a partir disto, capacidade de minerar informações em meio ao texto.

Partindo para prática, por meio da biblioteca nltk entenderemos passo a passo como o processo de mineração de dados.

mineracao.py | base.py

```
1  import nltk
2  from base import base
3  #nltk.download('popular')
4
5  stopwords = nltk.corpus.stopwords.words('portuguese')
6
```

Para este projeto, inicialmente iremos criar os respectivos arquivos (caso você opte por o fazer modularizado) e realizar as devidas importações da bibliotecas, módulos e pacotes envolvidos neste processo.

Inicialmente, em nosso arquivo mineração.py, executamos o comando import nltk para importar a mesma. Da mesma forma, por meio do comando from base import base estamos importando o conteúdo da variável base, de nosso pacote base.py

Note no comentário situado na linha 3, caso você ainda não tenha o feito, é interessante descomentar essa linha e a executar para que a biblioteca nltk baixe todas as suas dependências. Parametrizando a função download com ‘popular’ será feito o download das dependências mais comuns em pronetos, de acordo com sua preferência você pode parametrizar a mesma com ‘all’, sendo assim, serão baixadas todas as dependências da biblioteca nltk. Uma vez feita essa etapa, de fato temos condições de prosseguir com o código sem erros.

Em seguida é criada a variável stopwords, que por sua vez recebe como atributo o dicionário de stopwords em língua portuguesa por meio do comando exibido acima.

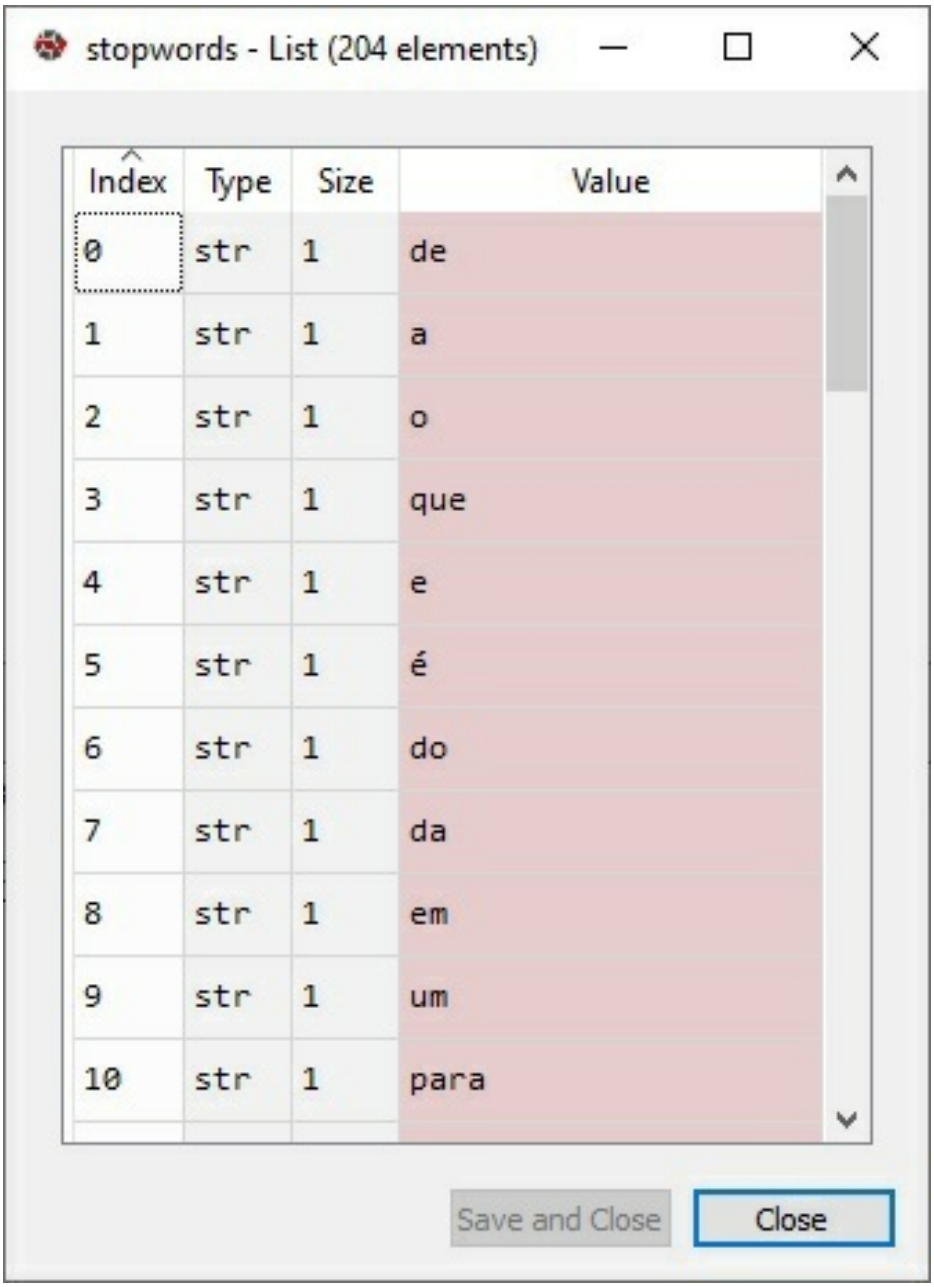
Entendendo de forma simples, stopwords nada mais são do que palavras que consideraremos irrelevantes ou que até mesmo atrapalham o processo de mineração.

mineracao.py | base.py

```
base = [('estou muito feliz','emoção positiva'),
        ('sou uma pessoa feliz','emoção positiva'),
        ('tudo bem comigo','emoção positiva'),
        ('alegria é o meu lema','emoção positiva'),
        ('comigo está tudo ok','emoção positiva'),
        ('muito bom ser amado','emoção positiva'),
        ('estou empolgado em começar','emoção positiva'),
        ('fui elogiado por meu trabalho','emoção positiva'),
        ('vencemos a partida','emoção positiva'),
        ('ganhei um presente','emoção positiva'),
        ('recebi uma promoção de cargo','emoção positiva'),
        ('estou bem, obrigada','emoção positiva'),
        ('o dia está muito bonito','emoção positiva'),
        ('estou bem, obrigado','emoção positiva'),
        ('me sinto satisfeito','emoção positiva'),
        ('fui aprovado','emoção positiva'),
        ('de bem com a vida','emoção positiva'),
        ('fui bem recebido em casa','emoção positiva'),
        ('estou com medo','emoção negativa'),
        ('estou com muito medo','emoção negativa'),
        ('estou um pouco triste','emoção negativa'),
        ('isto me deixou com raiva','emoção negativa'),
        ('fui demitida','emoção negativa'),
        ('esta comida está horrível','emoção negativa'),
        ('tenho pavor disso','emoção negativa'),
        ('de mal a pior','emoção negativa'),
        ('estou incomodado','emoção negativa'),
        ('estou sentindo dor','emoção negativa'),
        ('perdi a aposta','emoção negativa'),
        ('fui enganada','emoção negativa'),
        ('fiquei desmotivada com o resultado','emoção negativa'),
        ('que situação agoniante','emoção negativa'),
        ('estou doente','emoção negativa'),
        ('fui reprovado','emoção negativa')]
```

Em nosso arquivo base.py temos uma variável de nome base que, desculpe a redundância, basicamente guarda dados em forma de elementos de uma lista.

Repare que cada elemento está em um formato onde existe uma string com uma mensagem seguido da respectiva descrição quanto ao tipo de emoção. Para esse exemplo básico trabalharemos com uma base de dados bastante reduzida e categorizadas em apenas duas possíveis emoções, positiva ou negativa.



Explorando a variável stopwords podemos entender que aqui são filtrados e guardados todas as palavras que podem ser eliminadas do contexto sem atrapalhar os padrões necessários.

De nossa base de dados, composta por 34 frases com seus respectivos significados em emoções, foram filtrados 204 elementos os quais não representam relevância ao contexto da mineração.

```
7  def removestopwords(texto):
8      frases = []
9      for (palavras, emocao) in texto:
10         removesw = [p for p in palavras.split() if p not in stopwords]
11         frases.append((removesw, emocao))
12     return frases
13
```

Em seguida criamos a função que irá, nessa fase de tratamento de nossos dados, remover as stopwords de nossa base de dados.

Para isto, criamos uma função de nome removestopwords( ) que receberá um texto. Dentro de seu corpo é criada uma variável de nome frases que inicialmente é uma lista vazia.

Em seguida, por meio de um laço for, percorreremos cada elemento atribuído a texto em sua forma de palavra = emoção. Isso é feito por meio da variável removesw que realiza uma verificação onde, cada elemento de cada palavra que não constar em nossa lista de stopwords será adicionado para a lista de nossa variável frases, retornado essa variável composta de todos os elementos que estiverem nessas condições.

```
14 def reduzpalavras(texto):
15     steemer = nltk.stem.RSLPStemmer()
16     frases_redux = []
17     for (palavras, emocao) in texto:
18         reduzidas = [str(steemer.stem(p)) for p in palavras.split() if p not in stopwords]
19         frases_redux.append((reduzidas, emocao))
20     return frases_redux
21
```

Na sequência realizamos um segundo tratamento, agora reduzindo as palavras quanto ao seu radical, e isso é feito por meio da função RSLPStemmer( ). Basicamente o que é feito aqui é a extração dos sufixos das palavras para tornar nossa base de dados ainda mais enxuta.

Por exemplo, o radical das palavras alegria, alegre, alegrando, alegrar, alegremente é alegr. Para o interpretador é apenas isto que importa, inclusive não tendo nenhum prejuízo quanto a eliminação desses sufixos.

O processo de redução/radicalização será feito por meio da função reduzpalavras( ) que recebe como parâmetro um texto a ser processado. Inicialmente é criada a variável stemmer que chama a função RSLPStemmer( ) da biblioteca nltk.

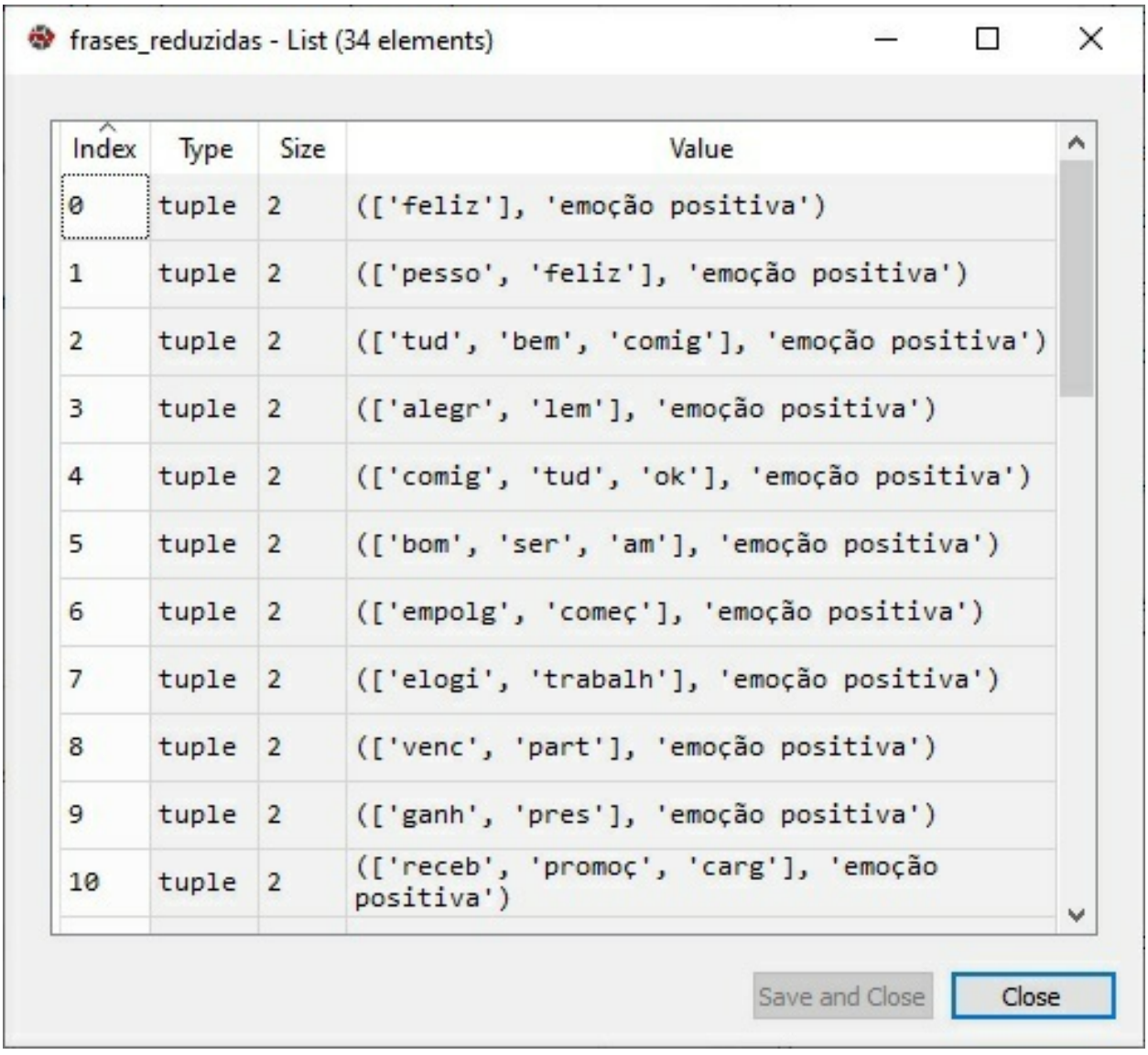
Em seguida é criada a variável frases_reduz que inicialmente trata-se de

uma lista vazia.

Exatamente da mesma forma como a função anterior foi criada, aqui percorreremos cada elemento verificando se o mesmo não está em nossa lista negra de stopwords e se pode ser reduzido pela função RSLPStemmer( ), adicionando esses elementos para lista atribuída a frases_redux, retornando a mesma.

```
22      frases_reduzidas = reduzpalavras(base)
23
```

Encerrando essa etapa, criamos no escopo global de nosso código a variável frases_reduzidas, que será nossa base de dados pós-tratamento.



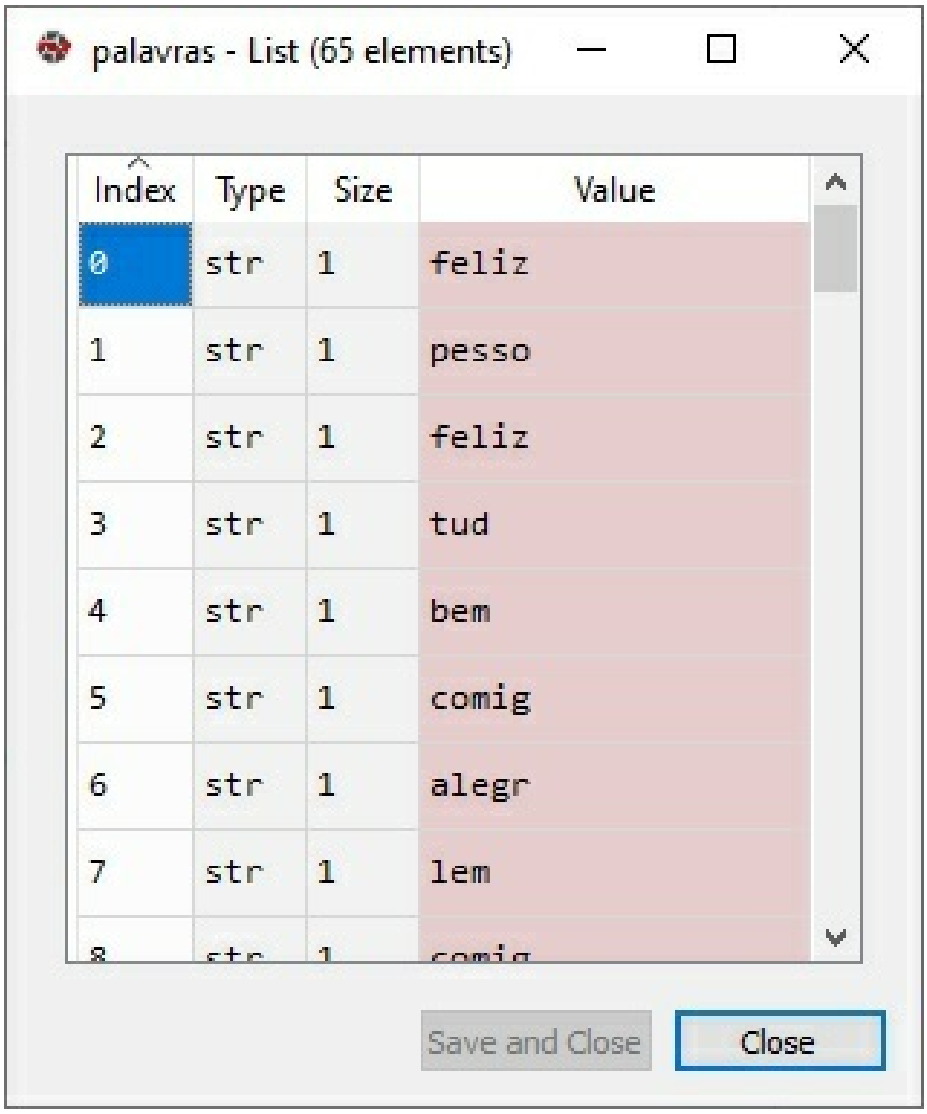
Por meio do explorador de variáveis podemos ver que agora frases_reduzidas possui seus dados em forma onde cada string teve seus elementos separados, reduzidos e associados a sua respectiva emoção.

```
24  def buscapalavras(frases):
25      todaspalavras = []
26      for (palavras, emocao) in frases:
27          todaspalavras.extend(palavras)
28      return todaspalavras
29
30  palavras = buscapalavras(frases_reduzidas)
31
```

Dando sequência, é criada a função buscapalavras( ) que recebe uma frase. Basicamente o que essa função fará é receber uma frase, aplicar o pré-processamento, fazendo uma varredura na mesma a preparando para comparar com nossa base de dados.

Note que assim como nas funções anteriores, os dados são processados sempre respeitando a equivalência das palavras contidas na frase com sua respectiva emoção.

Na sequência é criada uma variável de nome palavras, que chama a função buscapalavras( ) parametrizando a mesma com o conteúdo de frases_reduzidas.



Via explorador de variáveis é possível visualizar de forma clara a base de palavras geradas, palavras estas que serão referência para as devidas associações da mineração.

```
32  def buscafrequencia(palavras):
33      freq_palavras = nltk.FreqDist(palavras)
34      return freq_palavras
35
36  frequencia = buscafrequencia(palavras)
37
38  def buscapalavrasunicas(frequencia):
39      freq = frequencia.keys()
40      return freq
41
42  palavrasunicas = buscapalavrasunicas(frequencia)
43
```

Seguindo com nosso exemplo, podemos também criar funções dedicadas a nos retornar a frequência com que certas palavras aparecem em nossos dados assim como as palavras totalmente únicas.

Tenha em mente que, em aplicações reais, nossa base de dados será muito maior, e essa diferença de incidência/frequência com que certas palavras são assimiladas influencia a margem de acertos de nosso interpretador no que diz respeito as previsões geradas pelo mesmo.

Para a frequência de palavras criamos a função buscafrequencia( ) que recebe palavras como parâmetro. Dentro do corpo da função, temos a variável freq._palavras que chama a função FreqDist( ) da biblioteca nltk, parametrizando a mesma com palavras, retornando freq._palavras.

Nos mesmos moldes é criada a função buscapalavrasunicas( ) que recebe como parâmetro para si frequencia. No corpo da função temos uma variável de nome freq que nada mais faz do que pegar em forma numérica os dados de frequência, uma vez que os mesmos estão dispostos em forma parecida a de um dicionário, retornando esse valor.

```
44  def extrator(documento):
45      doc = set(documento)
46      caracteristicas = {}
47      for palavra in palavrasunicas:
48          caracteristicas['%s' % palavra] = (palavra in doc)
49      return caracteristicas
50
51  baseprocessada = nltk.classify.apply_features(extrator, frases_reduzidas)
52  #print(baseprocessada[0])
53
```

Terminados todos os tratamentos dos dados de nossa base, assim como as validações necessárias, hora de criar o mecanismo o qual irá de fato receber um texto a ser processado.

Para isso criamos a função extrator( ) que recebe um documento. Dentro do corpo dessa função inicialmente criamos uma variável de nome doc que recebe documento em forma de set.

Em seguida criamos uma variável de nome características que inicialmente é um dicionário vazio.

Na sequência por meio do laço for, iremos percorrer cada palavra de palavras únicas realizando a comparação da mesma com as palavras que

estarão em documento. Isso internamente nada mais é do que o rápido tratamento do texto recebido, de forma que ele terá seus elementos comparados um a um com todas as palavras já classificadas em nossa base de dados, marcando as mesmas como True quando houverem combinações e como False quando não houverem. Havendo combinações, nosso interpretador fará a associação com a emoção referente aquela palavra.

Por fim é criada a variável baseprocessada que chama a função apply_features( ) do módulo classify da biblioteca nltk, parametrizando a mesma com o conteúdo de extrator e de frases reduzidas, justamente para comparar a equivalência das mesmas em busca de emoções.

```
56  teste = str(input('Digite como você está se sentindo: '))
57  teste_redux = []
58  redux = nltk.stem.RSLPStemmer()
59  for (palavras_treino) in teste.split():
60      reduzida = [p for p in palavras_treino.split()]
61      teste_redux.append(str(redux.stem(reduzida[0])))
62
63  resultado = extrator(teste_redux)
64  print(classificador.classify(resultado))
65
```

Encerrando nossa linha de raciocínio, uma vez criada toda a estrutura que processará os elementos, podemos finalmente criar o mecanismo que lê os dados e os encaminha para processamento.

De início criamos uma variável de nome teste que pede ao usuário que digite como o mesmo está se sentindo. Também é criada uma variável de nome teste_redux que inicialmente é uma simples lista vazia.

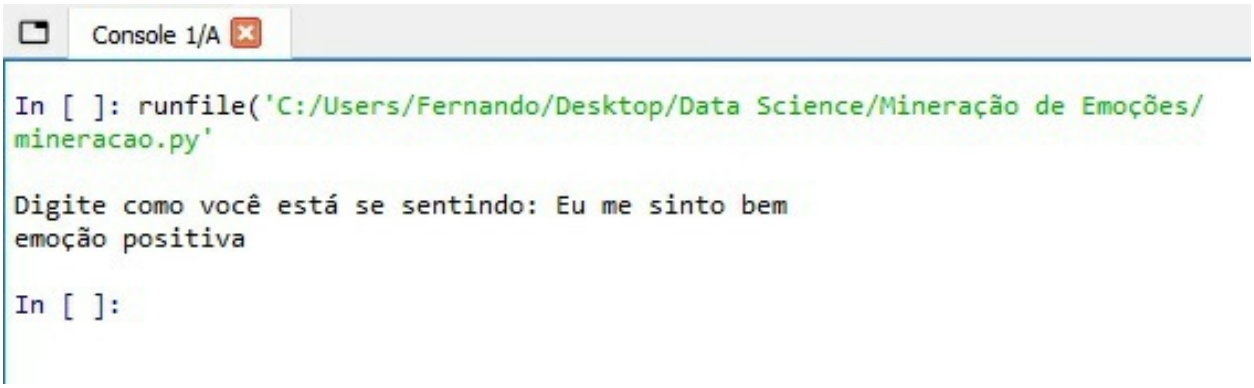
Da mesma forma também é criada a variável redux, que chama a função RSLPStemmer( ) sem parâmetros mesmo.

Na sequência por meio dos laço for realizamos aquele pré-tratamento dos dados, lendo os mesmos da variável teste, reduzindo e radicalizando os mesmos, guardando os mesmos em teste_redux.

Por fim, criamos uma variável de nome resultado que chama a função extrator( ) ativando toda a cadeia de processamento sobre os dados de teste_redux. O resultado é exibido por meio da função print( ) parametrizada com a função classify de nosso classificador, sobre os dados de resultado.

* Repare que para este exemplo, todo o processamento interno realizado pela

máquina é totalmente implícito, por meio das métricas das funções da própria biblioteca nltk. Nos exemplo subsequentes, usaremos de outras bibliotecas que nos permitirão uma margem maior de customização dos meios e métodos de processamento de nossos dados.



Pondo em teste nosso código, ao executar o mesmo é exibida via console a mensagem definida em nossa variável teste. Supondo que o usuário tenha digitado “Eu me sinto bem”, o resultado, como esperado, é emoção positiva.

Código Completo:

```
1   import nltk
2   #from base import *
3   #nltk.download('popular')
4
5   stopwords = nltk.corpus.stopwords.words('portuguese')
6
7   def removestopwords(texto):
8       frases = []
9       for (palavras, emocao) in texto:
10          removesw = [p for p in palavras.split() if p not in stopwords]
11          frases.append((removesw, emocao))
12      return frases
13
14  def reduzpalavras(texto):
15      steemer = nltk.stem.RSLPStemmer() # faz o radical das palavras
16      frases_redux = []
17      for (palavras, emocao) in texto:
18          reduzidas = [str(steemer.stem(p)) for p in palavras.split() if p nc
19          frases_redux.append((reduzidas, emocao))
20      return frases_redux
21
22  frases_reduzidas = reduzpalavras(base)
23
```

```
def buscapalavras(frases):
    todaspalavras = []
    for (palavras, emocao) in frases:
        todaspalavras.extend(palavras)
    return todaspalavras

palavras = buscapalavras(frases_reduzidas)

def buscafrequencia(palavras):
    freq_palavras = nltk.FreqDist(palavras)
    return freq_palavras

frequencia = buscafrequencia(palavras)

def buscapalavrasunicas(frequencia):
    freq = frequencia.keys()
    return freq

palavrasunicas = buscapalavrasunicas(frequencia)

def extrator(documento):
    doc = set(documento)
    caracteristicas = {}
    for palavra in palavrasunicas:
        caracteristicas['%s' % palavra] = (palavra in doc)
    return caracteristicas

baseprocessada = nltk.classify.apply_features(extrator, frases_reduzidas)
#print(baseprocessada[0])

classificador = nltk.NaiveBayesClassifier.train(baseprocessada)

teste = str(input('Digite como você está se sentindo: '))
teste_redux = []
redux = nltk.stem.RSLPStemmer()
for (palavras_treino) in teste.split():
    reduzida = [p for p in palavras_treino.split()]
    teste_redux.append(str(redux.stem(reduzida[0])))

resultado = extrator(teste_redux)
print(classificador.classify(resultado))
```

Classificação Multiclasse – Aprendizado de Máquina Simples – Iris Dataset

Em 1936, o estatístico e biólogo britânico Ronald Fisher publicou seu artigo intitulado *The use of multiple measurements in taxonomic problems*, que em tradução livre seria algo como O Uso de Múltiplas Medições em Problemas Taxonômicos, onde publicou seu estudo classificatório das plantas do tipo Íris quanto às características dos formatos de suas pétalas e sépalas em relação ao seu tamanho.



Fazendo o uso desse artigo científico como base, Edgar Anderson coletou e quantificou os dados para finalmente classificar estas amostras em 3 tipos de flores diferentes a partir de métodos computacionais estatísticos publicando em 2012 o seu estudo *An Approach for Iris Plant Classification Using Neural Network*, que em tradução livre seria algo como Uma Abordagem para Classificação das Plantas Íris Usando Redes Neurais.

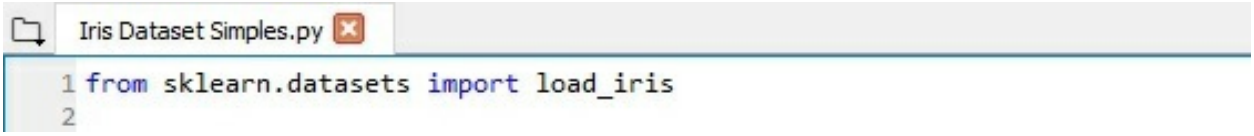
O método de classificação utilizado em ambos estudos foi igualmente dividir o conjunto em grupos de amostras e separá-las de acordo com o comprimento e a largura em centímetros das pétalas e das sépalas das mesmas, já que este era um padrão entre aqueles tipos de flores na natureza. Vale lembrar que a metodologia de classificação dessas amostras em sua época foi manual e estatística, agora o que faremos é aplicar um modelo computacional para classificação das mesmas e, até mais importante que isso, criar um modelo que possa ser aplicado para novas classificações.

Por fim, para pormos em prática os métodos de classificação de dados utilizaremos da Iris Dataset, que nada mais é do que um dos conjuntos de dados inclusos na biblioteca SKLearn, baseado no trabalho de Ronald Fisher. *Tal modelo pode ser aplicado para diferentes situações e tipos de dados, aqui utilizaremos o Iris Dataset para inicialmente entendermos as mecânicas usadas quando temos de classificar amostras de vários tipos.

Inicialmente faremos uma abordagem mais simples aplicando alguns modelos classificatórios e no capítulo subsequente estaremos aplicando uma metodologia baseada em rede neural artificial densa.

Como já comentado anteriormente, estaremos aprendendo de forma procedural, quanto mais formos avançando nossos estudos, mais prática será a abordagem.

Partindo para o Código:



Como sempre, todo processo se inicia com a importação das bibliotecas e módulos que serão utilizados ao longo do código e que não fazem parte das bibliotecas pré-alocadas da IDE. Aqui inicialmente iremos utilizar a Iris Dataset que está integrada na biblioteca SKLearn para fins de estudo.

Quando estamos importando uma biblioteca inteira simplesmente executamos o comando import seguido do nome da biblioteca, aqui o que faremos é importar parte do conteúdo de uma biblioteca, é bastante comum importarmos algum módulo ou função no lugar de toda uma biblioteca. Por meio do comando from seguido de sklearn.datasets seguido de import load_iris, estamos importando a sub biblioteca de exemplo load_iris, do módulo datasets da biblioteca sklearn.

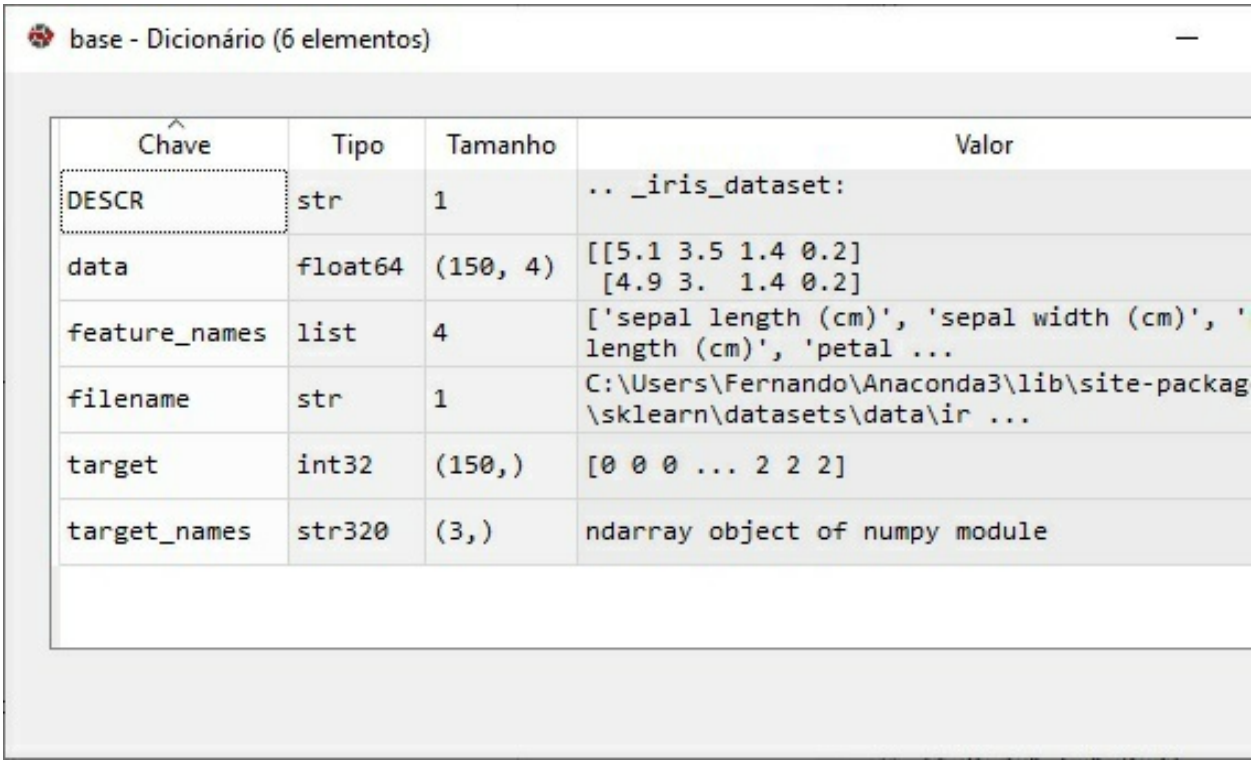
Selecionando e executando esse código (Ctrl + ENTER) se não houve nenhum erro de sintaxe você simplesmente não terá nenhuma mensagem no console, caso haja algum traceback significa que houve algum erro no processo de importação.

```
3 base = load_iris()
4
```

Em seguida criamos uma variável de nome base que recebe como atributo todo conteúdo contido em load_iris. Sem parâmetros mesmo.

Nome	Tipo	Tamanho	Valor
base	utils.Bunch	6	Bunch object of sklearn.utils module

Dando uma olhada no Explorador de Variáveis, é possível ver que foi criada uma variável reconhecida como objeto parte da biblioteca sklearn.



Clicando duas vezes sobre a variável é possível explorar a mesma de forma visual. Assim é possível reconhecer que existem 4 colunas (Chave, Tipo, Tamanho e Valor) assim como 6 linhas (DESCR, data, feature_names, filename, target e target_names).

Uma das etapas que será comum a praticamente todos os exemplos que iremos utilizar é uma fase onde iremos fazer o polimento desses dados, descartando os que não nos interessam e reajustando os que usaremos de forma a poder aplicar operações aritméticas sobre os mesmos.

Podemos aplicar essa etapa diretamente na variável ou reaproveitar

partes do conteúdo da mesma em outras variáveis. Aqui, inicialmente faremos o reaproveitamento, em exemplos futuros trabalharemos aplicando esse polimento diretamente sobre o banco de dados inicial.

```
5 print(base.data)
6 print(base.target)
7 print(base.target_names)
```

Podemos visualizar tais dados de forma manual por meio da função .print( ), dessa forma, teremos a exibição do conteúdo em sua forma normal no console/terminal.

```
    ...: print(base.data)
[[5.1 3.5 1.4 0.2]
 [4.9 3.  1.4 0.2]
 [4.7 3.2 1.3 0.2]
 [4.6 3.1 1.5 0.2]
 [5.  3.6 1.4 0.2]
 [5.4 3.9 1.7 0.4]
 [4.6 3.4 1.4 0.3]
 [5.  3.4 1.5 0.2]
 [4.4 2.9 1.4 0.2]
 [4.9 3.1 1.5 0.1]
 [5.4 3.7 1.5 0.2]
 [4.8 3.4 1.6 0.2]
 [4.8 3.  1.4 0.1]
 [4.3 3.  1.1 0.1]
 [5.8 4.  1.2 0.2]
 [5.7 4.4 1.5 0.4]
 [5.4 3.9 1.3 0.4]
```

Retorno obtido em print(base.data), aqui se encontram os dados das medições em forma de uma matriz com 150 amostras/registros.

```
In [  ]: print(base.target)
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2]
```

Retorno obtido em print(base.target), onde é possível ver que para cada uma das 150 amostras anteriores existe um valor referente a sua classificação indo de 0 a 2.

```
In [  ]: print(base.target_names)
['setosa' 'versicolor' 'virginica']
```

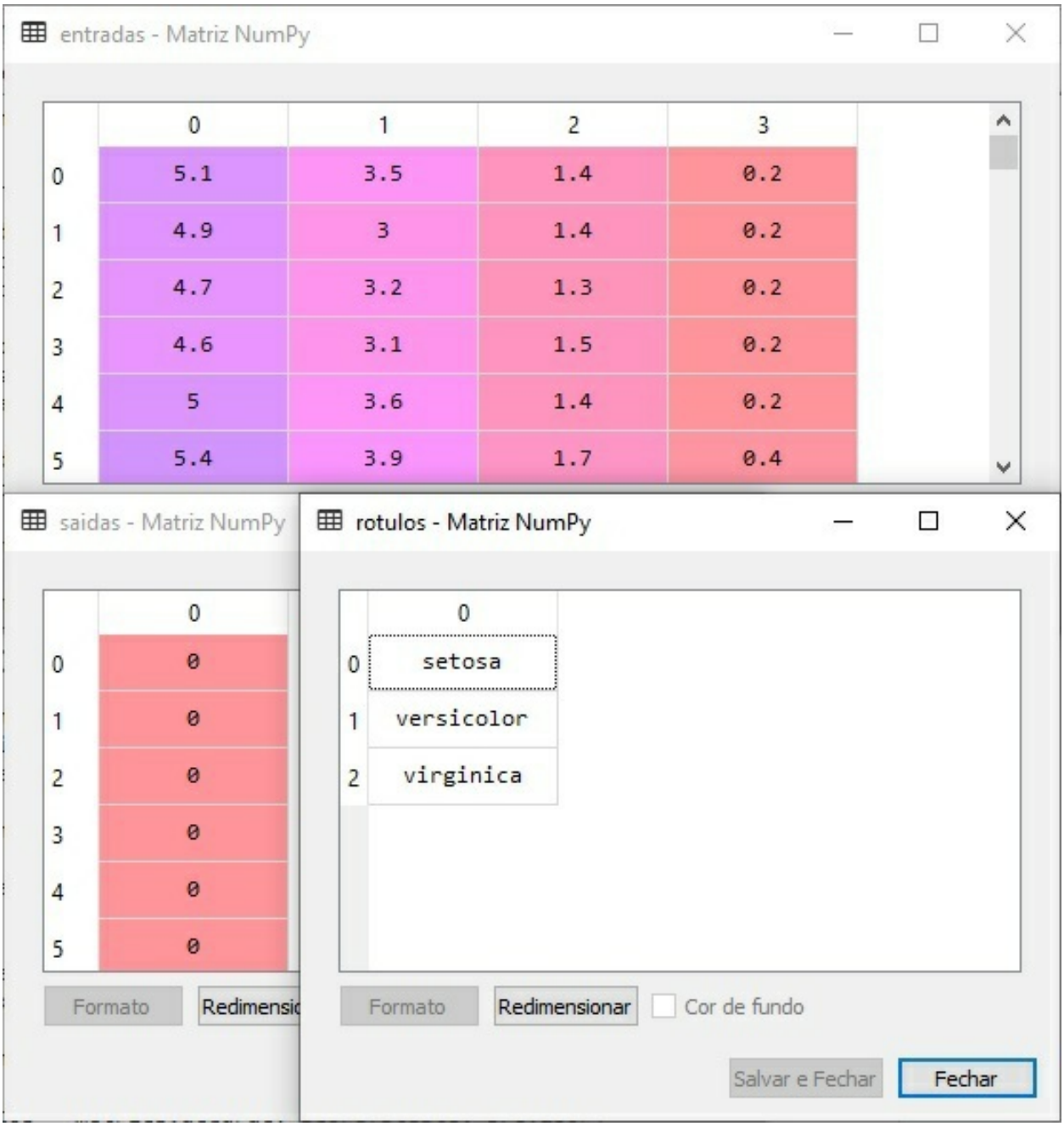
Retorno obtido em print(base.target_names), onde podemos relacionar a numeração anterior 0, 1 e 2 agora com os nomes dos três tipos de plantas, 0 para setosa, 1 para versicolor e 2 para virgínica.

```
 9 entradas = base.data
10 saidas = base.target
11 rotulos = base.target_names
```

Em seguida criamos três variávels, entradas que recebe como atributo os dados contidos apenas em data da variável base. saidas que recebe como atributo os dados de target da variável base e por fim uma variável de nome rotulos que recebe o conteúdo de target_names da variável base.

Nome	Tipo	Tamanho	Valor
base	utils.Bunch	6	Bunch object of sklearn.utils module
entradas	float64	(150, 4)	[[5.1 3.5 1.4 0.2] [4.9 3. 1.4 0.2]
rotulos	str320	(3,)	ndarray object of numpy module
saidas	int32	(150,)	[0 0 0 ... 2 2 2]

Se todo processo correu normalmente no explorador de variáveis é possível visualizar tanto as variáveis quanto seu conteúdo.



Relacionando os dados, em entradas temos os quatro atributos que foram coletados para cada uma das 150 amostras (tamanho e comprimento das pétalas e das sépalas). Em saidas temos a relação de que, de acordo com os parâmetros contidos em entradas tal planta será classificada em um dos três tipos, 0, 1 e 2. Por fim, de acordo com essa numeração podemos finalmente saber de acordo com a classificação qual o nome da planta.

```
13 print(base.data.shape)
14 print(base.target.shape)
```

Por meio do comando .shape podemos de fato verificar o tamanho dessas matrizes de dados para que possamos fazer o cruzamento e/ou aplicar funções aritméticas sobre os mesmos.

```
In [  ]: print(base.data.shape)
    ...: print(base.target.shape)
(150, 4)
(150,)
```

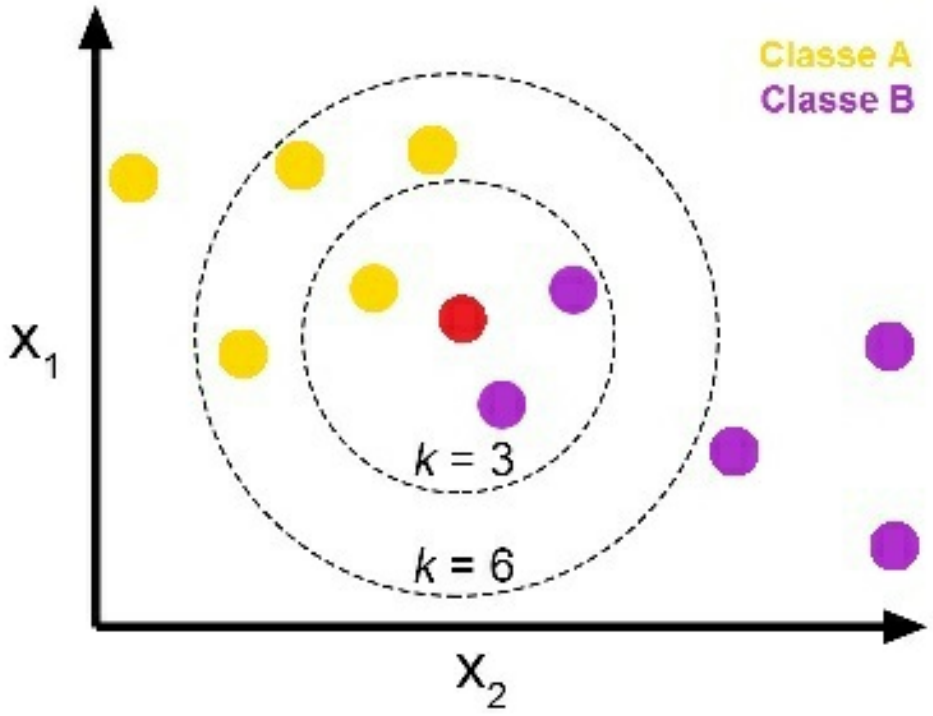
Selecionando e executando esse bloco de código podemos verificar via terminal que data tem um formato de 150 linhas e 4 colunas, assim como saidas tem 150 linhas e uma coluna.

Terminada a fase de polimento dos dados hora de começarmos a finalmente processar os mesmos para transformá-los de dados brutos para informação estatística.

Aplicando o Modelo KNN

Agora iremos pôr em prática a análise estatística dos nossos dados, para isto começaremos usando o modelo KNN, que do inglês K Nearest Neighbors, em tradução livre significa Vizinhos Próximos de K. Neste tipo de modelo estatístico teremos um item de verificação por convenção chamado K e o interpretador com base nos dados fornecidos fará a leitura e exibirá como resultado os dados mais próximos, que mais se assemelham a K.

Em outras palavras, definiremos um parâmetro, onde de acordo com a proximidade (ou distância) do valor da amostra para esse parâmetro, conseguimos obter uma classificação.



Repare no gráfico, o ponto vermelho representa K, o número/parâmetro que definiremos para análise KNN, próximo a K existem 3 outros pontos, um referente a classe A e dois referentes a classe B. O modelo estatístico KNN trabalhará sempre desta forma, a partir de um número K ele fará a classificação dos dados quanto a proximidade a ele.

Pela documentação do KNN podemos verificar que existe uma série de etapas a serem seguidas para aplicação correta deste modelo:

1º - Pegamos uma série de dados não classificados.

2º - O interpretador irá mensurar a proximidade desses dados quanto a sua semelhança.

3º - Definimos um número K, para que seja feita a análise deste e de seus vizinhos.

4º - Aplicamos os métodos KNN.

5º - Analisamos e processamos os resultados.

```
16 from sklearn.neighbors import KNeighborsClassifier
17
```

Para aplicar o modelo estatístico KNN não precisamos criar a estrutura lógica mencionada acima do zero, na verdade iremos usar tal classificador que já vem pronto e pré-configurado no módulo neighbors da biblioteca sklearn.

Assim como fizemos no início de nosso código, iremos por meio do comando from importar apenas o KneighborsClassifier de dentro de sklearn, muita atenção a sintaxe do código, para importação correta você deve respeitar as letras maiúsculas e minúsculas da nomenclatura.

```
18 knn = KNeighborsClassifier(n_neighbors = 1)
19 knn.fit(entradas, saidas)
20 knn.predict([[5.1,3.1,1.4,0.2]])
```

Logo após criamos uma variável de nome knn que recebe como atributo KneighborsClassifier, que por sua vez tem o parâmetro n_neighbors = 1, o que significa que tais dados serão classificados quanto a proximidade de 1 número em relação a sua amostra vizinha. Este parâmetro, inclusive, pode ser alterado para realização de outros testes de performance posteriormente.

Em seguida aplicamos sobre knn a função .fit( ), essa função por sua vez serve para alimentar o classificador com os dados a serem processados. Repare que knn.fit( ) recebe como parâmetro todos dados de entradas e saidas. Por fim, também podemos fazer uma simples previsão por meio da função .predict( ), aqui, pegando os dados de uma linha de nossa base de dados podemos realizar uma previsão sobre ela.

```
In [  ]: from sklearn.neighbors import KNeighborsClassifier
    ...:
    ...: knn = KNeighborsClassifier(n_neighbors = 1)
    ...: knn.fit(entradas, saidas)
    ...: knn.predict([[5.1,3.1,1.4,0.2]])
Out[  ]: array([0])
```

Após selecionar todo bloco de código anterior e executá-lo, acompanhando via console podemos ver que a criação do classificador foi feita de maneira correta assim como a alimentação do mesmo com os valores contidos em entradas e saidas. Via console também é possível ver o retorno da função de previsão sobre os valores retirados de uma das linhas da matriz

base.

O retorno por hora foi um simples “0” o que de imediato não significa muita coisa. Será comum você ter de apresentar seus dados e suas conclusões, seja para fins de estudo ou profissionais, e mostrar um zero pode não significar nada para quem não entende do assunto. Para um resultado mais claro vamos associar este resultado inicial com os rótulos que separamos lá no início desse código.

```
22 especie = knn.predict([[5.9,3,5.1,1.8]])[0]
23 print(especie)
24 rotulos[especie]
```

Para isso criamos uma variável de nome especie que recebe como atributo nosso classificador knn e a função .predict( ) que por sua vez recebe como parâmetro outra linha da matriz escolhida aleatoriamente. Note que ao final do código de atribuição existe um parâmetro em forma de posição de lista [0].

Executando uma função print( ) sobre especie, e associando o conteúdo da mesma com o de rotulos como retorno deveremos ter o rotulo associado a posição 0 da lista de especies.

```
In [  ]: especie = knn.predict([[5.9,3,5.1,1.8]])[0]
    ...: print(especie)
    ...: rotulos[especie]
2
Out[  ]: 'virginica'
```

De fato, é o que acontece, agora como resultado das associações temos como retorno ‘virginica’ o que é uma informação muito mais relevante do que ‘0’.

Seguindo com a análise dos dados, um método bastante comum de ser aplicado é o de treino e teste do algoritmo, para que de fato se possa avaliar a performance do modelo. Na prática imagine que você tem uma amostra a ser classificada, feita toda a primeira etapa anterior, uma maneira de obter um viés de confirmação é pegar novamente a base de dados, dividir ela em partes e as testar individualmente, assim é possível comparar e avaliar a performance com base nos testes entre si e nos testes em relação ao resultado anterior.

Todo processo pode ser realizado por uma ferramenta dedicada a isso da biblioteca sklearn, novamente, tal ferramenta já vem pré-configurada para uso, bastando fazer a importação, a alimentação e a execução da mesma.

```
26 from sklearn.model_selection import train_test_split
27
```

Executando o comando acima, respeitando a sintaxe, importamos a ferramenta train_test_split do módulo model_selection da biblioteca sklearn.

```
28 etreino, eteste, streino, steste = train_test_split(entradas,
29                                                    saidas,
30                                                    test_size = 0.25)
```

A seguir criamos quatro variáveis dedicadas a conter amostras dos dados para treino e para teste do modelo, por convenção as chamamos de etreino (entradas par treino), eteste (entradas para teste), streino (saídas para treino) e steste (saídas para teste).

Todas elas recebem como atributo a função train_test_split( ) que por sua vez recebe como parâmetro os dados contidos em entradas, saidas e um terceiro parâmetro chamado test_size, aqui definido como 0.25, onde 25% das amostras serão dedicadas ao treino, consequentemente 75% das amostras serão usadas para teste.

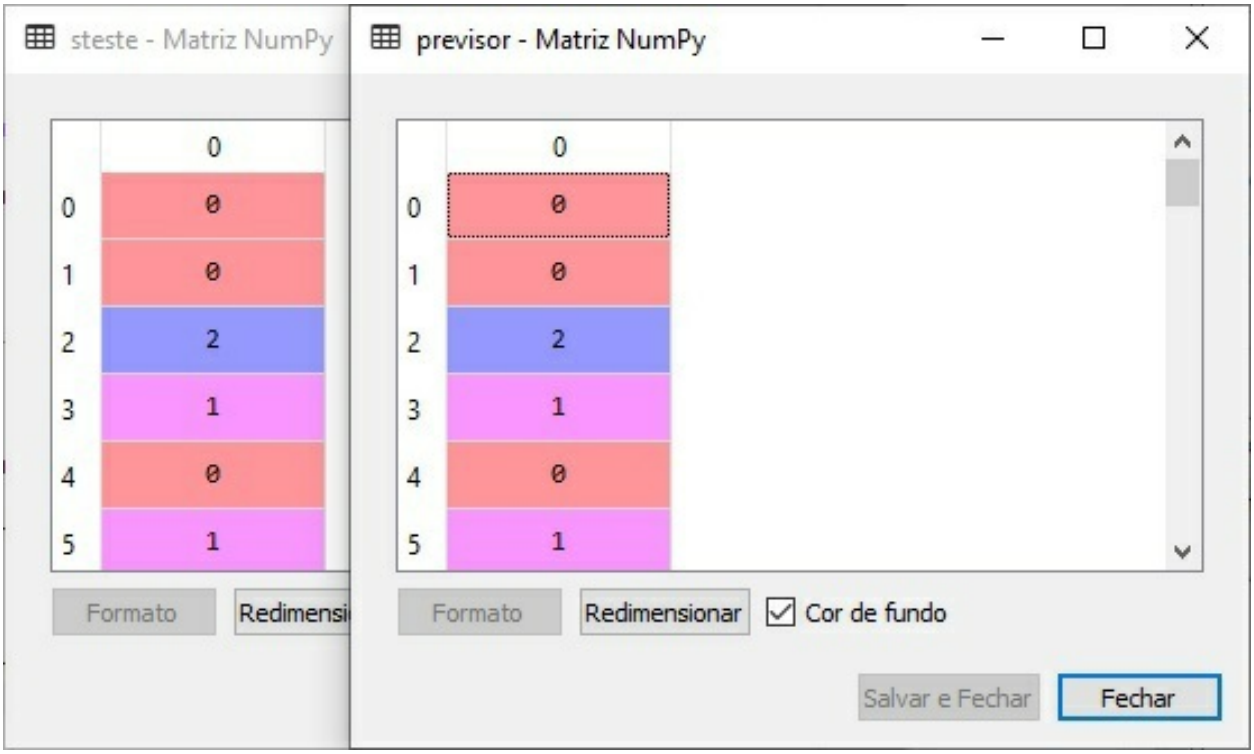
Você pode alterar esse valor livremente, porém é notado que se obtém melhores resultados quando as amostras não são divididas igualmente (50%/50%), lembrando também que uma base de treino muito pequena pode diminuir a precisão dos resultados.

```
31 knn.fit(etreino, streino)
32 previsor = knn.predict(eteste)
```

Em seguida usamos novamente nossa variável de nome knn que já executa sobre si a função .fit( ) que por sua vez recebe como parâmetro os dados contidos em etreino e streino. Dando sequência criamos uma variável de nome previsor que executa como atributo a função knn.predict( ) passando como parâmetro para a mesma os dados contidos em eteste.

Nome	Tipo	Tamanho	Valor
base	utils.Bunch	6	Bunch object of sklearn.utils module
entradas	float64	(150, 4)	[[5.1 3.5 1.4 0.2] [4.9 3. 1.4 0.2]
especie	int32	1	2
eteste	float64	(38, 4)	[[4.7 3.2 1.3 0.2] [5.2 3.4 1.4 0.2]
etreino	float64	(112, 4)	[[6.2 2.2 4.5 1.5] [6.1 2.6 5.6 1.4]
previsor	int32	(38,)	[0 0 2 ... 2 0 1]
rotulos	str320	(3,)	ndarray object of numpy module

Selecionando e executando todo bloco de código acima é possível ver que as devidas variáveis foram criadas.



Selecionando as variáveis com dados de saída é possível fazer a comparação entre elas e avaliar de forma visual se os valores encontrados em ambos os testes condizem. É normal haver disparidade entre esses dados, e isso é uma consequência normal de termos uma base de dados pequena, e a reduzir ainda mais no processo de a dividir para treino e teste. Raciocine que aqui nossa base de dados contém apenas 150 amostras, dependendo do

contexto você terá bases de dados de centenas de milhares de amostras/registros, além de seus atributos. Quanto menor for a base de dados, menos é a precisão na taxa de acertos para previsões.

Além de dividir as amostras e as testar individualmente, outro processo bastante comum é usar ferramentas que de fato avaliam a performance de nossos modelos, uma das mais comuns a ser usada é a ferramenta accuracy_score da biblioteca sklearn.

```
34 from sklearn import metrics
35
```

Um pouco diferente das importações realizadas anteriormente, aqui não temos como importar apenas a ferramenta em questão, ela depende de outras funções contidas no mesmo pacote em seu módulo, sendo necessário importar todo o módulo do qual ela faz parte. Para isso simplesmente importamos o módulo metrics da biblioteca sklearn por meio do código acima.

```
36 margem_acertos = metrics.accuracy_score(steste, previsor)
37
```

Prosseguindo com o código, criamos uma nova variável de nome margem_acertos que recebe como atributo a execução de metrics.accuracy_score, que por sua vez recebe como parâmetro os dados de steste e de previsor.

Nome	Tipo	Tamanho	Valor
etreino	float64	(112, 4)	[[6.2 2.2 4.5 1.5] [6.1 2.6 5.6 1.4]
margem_acertos	float64	1	0.9473684210526315
previsor	int32	(38,)	[0 0 2 ... 2 0 1]

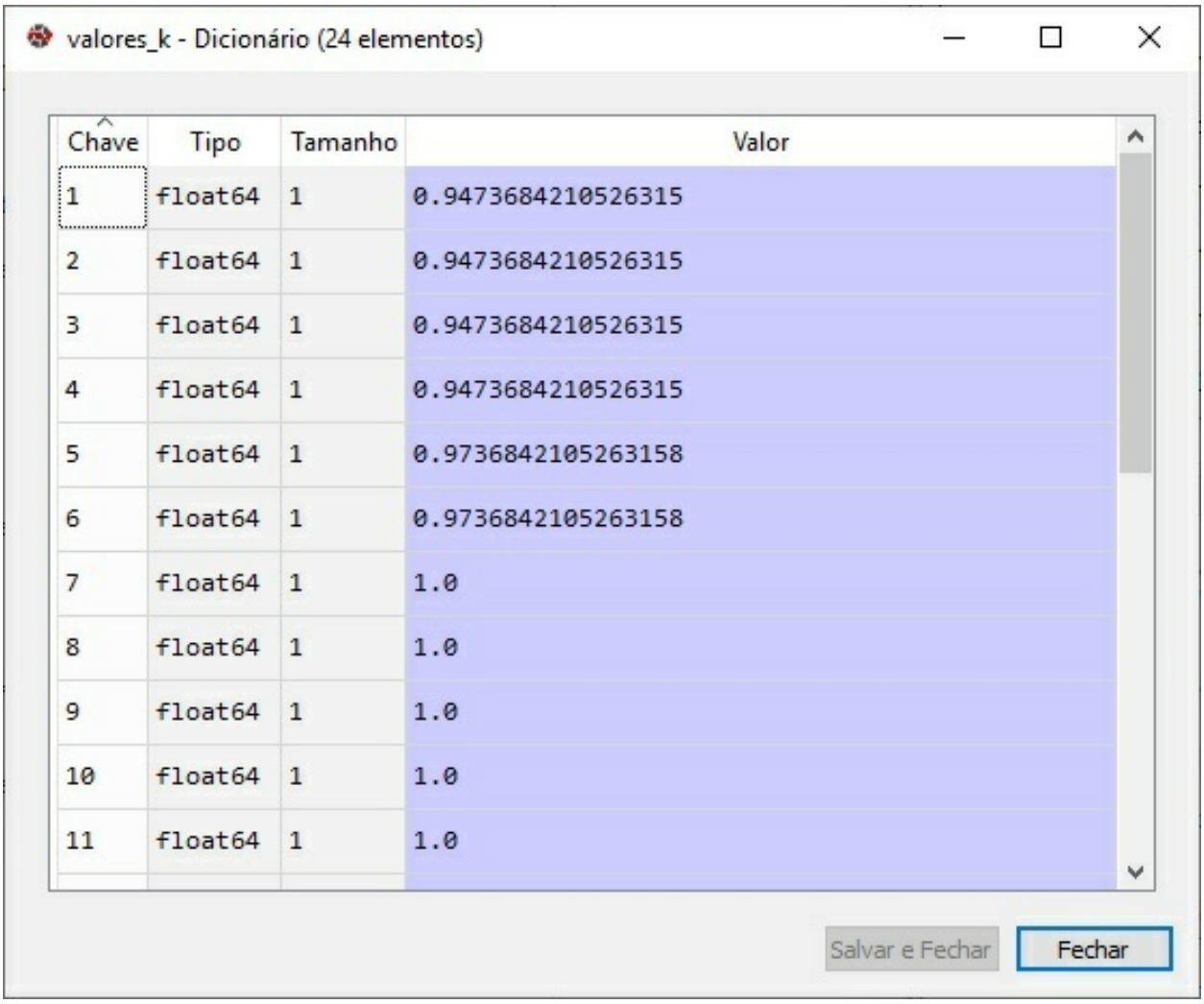
Selecionando e executando o bloco de código anterior é criada a variável margem_acertos, que nos mostra de forma mais clara um valor de 0.947, o que em outras palavras significa que o resultado das comparações dos valores entre steste e previsores tem uma margem de acerto de 94%.

Outra metodologia bastante comum é, retornando ao KNN, alterar aquele parâmetro n_neighbors definido anteriormente como 1 e realizar novos testes, inclusive é possível criar um laço de repetição que pode testar e exibir resultados com vários parâmetros dentro de um intervalo.

```
38 valores_k = {}
39 k=1
40 while k < 25:
41     knn = KNeighborsClassifier(n_neighbors=k)
42     knn.fit(etreino, streino)
43     previsores_k = knn.predict(eteste)
44     acertos = metrics.accuracy_score(steste, previsores_k)
45     valores_k[k] = acertos
46     k += 1
```

Aqui vamos diretamente ao bloco de código inteiro, supondo que você domina a lógica de laços de repetição em Python. Inicialmente o que fazemos é criar uma variável valores_k que recebe como atributo uma chave vazia. Em seguida definimos que o valor inicial de k é 1. Sendo assim, criamos uma estrutura de laço de repetição onde, enquanto o valor de k for menor que 25 é executado o seguinte bloco de código: Em primeiro lugar nossa variável knn recebe o classificador mas agora seu parâmetro não é mais 1, mas o valor que estiver atribuído a k. Em seguida knn é alimentada pela função .fit( ) com os dados contidos em etreino e streino. Logo após criamos uma variável previsores_k que recebe a função previsora sob os dados contidos em eteste.

Na sequência criamos uma variável de nome acertos que realiza o teste de previsão sobre os valores contidos em steste e previsores_k. Sendo assim valores_k recebe cada valor atribuído a k e replica esse valor em acertos. Por fim k é incrementado em 1 e todo laço se repete até que k seja igual a 25.



Selecionando e executando todo esse código é criada a variável valores_k onde podemos visualizar que foram realizados testes, cada um com um valor de k diferente, entre 1 e 25 com sua respectiva margem de acerto.

Dessa forma, para essa pequena base de dados, podemos ver que vários valores diferentes de k tem uma altíssima margem de acertos, em bases maiores ou com dados mais heterogêneos, é mais comum que esses valores de k sejam bastante divergentes entre si. Aqui nesse exemplo o valor mais baixo encontrado foi de 94% de precisão mesmo.

Outra situação bastante comum é, uma vez que temos informações relevantes a partir de nossos dados, criar mecanismos para que os mesmos possam ser exibidos de forma mais clara. Uma das coisas mais comuns em ciência de dados é apresentar os dados em forma de gráfico. Para isso também usaremos uma ferramenta de fácil integração e uso em nossa IDE,

chamada matplotlib.

```
48 import matplotlib.pyplot as plt
49
```

A importação dessa biblioteca segue os mesmos moldes usados anteriormente, por meio do comando import e podemos referenciá-la como plt por comodidade.

```
50 plt.plot(list(valores_k.keys()),
51          list(valores_k.values()))
```

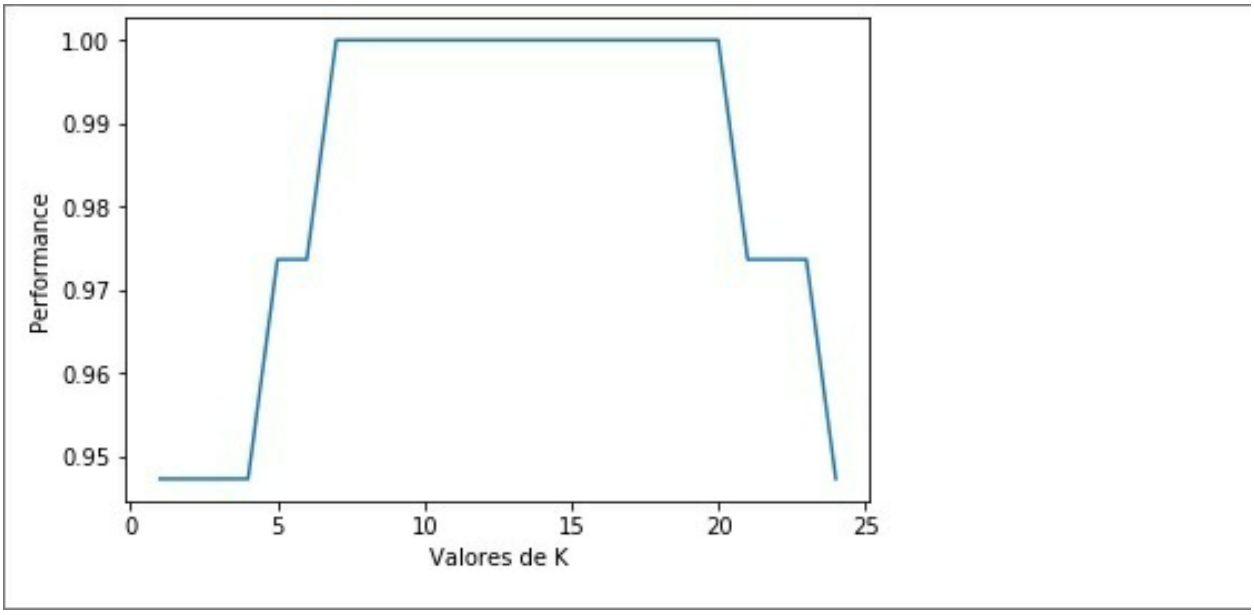
Raciocine que esta plotagem em gráfico é uma forma simples e visual de apresentar os dados via console/terminal. Sendo assim, primeiramente criamos uma função plt.plot( ) que recebe como atributos, em forma de lista, as chaves e os valores de valores_k, respectivamente as referenciando por .keys( ) e por .values( ). Assim podemos ter um plano cartesiano com eixos X e Y onde tais dados serão exibidos.

```
52 plt.xlabel('Valores de K')
53 plt.ylabel('Performance')
```

Inclusive uma prática comum é definir rótulos para os eixos X e Y do gráfico para que fique de ainda mais fácil interpretação.

```
54 plt.show()
55
```

Executando a função plt.show( ) o gráfico será gerado e exibido em nosso console.



Analisando o gráfico podemos ver que no eixo X (horizontal) existe a relação dos valores de k no intervalo entre 1 e 25. Já no eixo Y (vertical) podemos notar que foram exibidos valores entre 95% e 100% de margem de acerto.

Apenas por curiosidade, muito provavelmente você replicou as linhas de código exatamente iguais as minhas e seu gráfico é diferente, note que a cada execução do mesmo código o gráfico será diferente porque a cada execução do mesmo ele é reprocessado gerando novos valores para todas as variáveis.

Aplicando Regressão Logística

Outra prática bastante comum que também pode ser aplicada sobre nosso modelo é a chamada Regressão Logística, método bastante usado para realizar previsões categóricas. Ou seja, testar amostras para descobrir qual a probabilidade da mesma fazer parte de um tipo ou de outro.

```
56 from sklearn.linear_model import LogisticRegression
57
```

Como sempre, o processo se inicia com a importação do que for necessário para nosso código, nesse caso, importamos a ferramenta LogisticRegression do módulo linear_model da biblioteca sklearn por meio do comando acima.

```
58 regl = LogisticRegression()
59 regl.fit(etreino, streino)
```

Inicialmente criamos uma variável de nome regl que recebe como atributo o regressor LogisticRegression, sem parâmetros mesmo. Em seguida já o alimentamos com os dados de etreino e streino por meio da função .fit( ).

```
Out[  ]:
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='warn', n_jobs=None, penalty='l2',
                   random_state=None, solver='warn', tol=0.0001, verbose=0,
                   warm_start=False)
```

Repare em seu console que, como nesse caso, quando criamos uma variável e atribuímos a ela alguma ferramenta, sem passar nenhum parâmetro,

a mesma é criada normalmente com seus parâmetros pré-configurados, se no código não haviam parâmetros não significa que tal ferramenta internamente não possua parâmetros, toda ferramenta já vem pré-configurada dessa forma, com valores padrão, você pode se sentir à vontade para ler a documentação de tal ferramenta e estudar a fundo o que cada parâmetro faz.

Aqui ao longo dos exemplos serão dadas as devida explicações para os parâmetros que estão em uso, os internos devem ser consultados na documentação.

```
60 regl.predict([[6.2,3.4,5.4,2.3]])
61 regl.predict_proba([[6.2,3.4,5.4,2.3]])
```

Dando sequência, aplicando sob nossa variável regl as funções .predict( ) e .predict_proba( ), passando como parâmetro uma linha escolhida aleatoriamente em nossa amostra, é realizada a devida previsão em forma de regressão logística.

```
In [  ]: regl.predict([[6.2,3.4,5.4,2.3]])
Out[  ]: array([2])

In [  ]: regl.predict_proba([[6.2,3.4,5.4,2.3]])
Out[  ]: array([[0.00167941, 0.1447824 , 0.85353819]])
```

Via console é possível ver o retorno da função .predict( ) que é array([2]), lembrando que nossa classificação é para valores entre 0 e 1 e 2, tal amostra analisada é do tipo 2 (virginica). Já o retorno de predict_proba( ) é interessante pois podemos ver que ele nos mostra uma array([[0.00, 0.14, 0.85]]) o que nos mostra que tal amostra tinha 0% de chance de ser do tipo 0, 14% de chance de ser do tipo 1 e 85% de chance de ser do tipo 2, o que é uma informação relevante do funcionamento do algoritmo.

```
62 previsor_regl = regl.predict(eteste)
63 margem_acertos_regl = metrics.accuracy_score(steste, previsor_regl)
```

Por fim criamos uma variável de nome previsor_regl que recebe como atributo o previsor de regl sob os valores de eteste. Por último criamos uma variável de nome margem_acertos_regl que aplica a função de taxa de precisão com base nos valores de steste e previsor_regl.

Nome	Tipo	Tamanho	Valor
k	int	1	25
margem_acertos	float64	1	0.9473684210526315
margem_acertos_regl	float64	1	1.0
previsor	int32	(38,)	[0 0 2 ... 2 0 1]
previsor_regl	int32	(38,)	[0 0 2 ... 2 0 1]
previsores_k	int32	(38,)	[0 0 2 ... 2 0 1]
rotulos	str320	(3,)	ndarray object of numpy module

Executando o último bloco de código podemos ver que são criadas as respectivas variáveis, uma com os valores previstos com base nos testes, assim como sua taxa de precisão (usando inclusive um dos valores de k que haviam dado como resultado 100% de precisão anteriormente), pois a taxa de previsão sobre as amostras se manteve em 100%.

Sendo assim terminamos o entendimento básico do processamento e análise de dados sobre a Iris Dataset usando modelo de rede neural simples.

Código Completo:

```python
from sklearn.datasets import load_iris
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn import metrics
import matplotlib.pyplot as plt
from sklearn.linear_model import LogisticRegression

base = load_iris()
entradas = base.data
saidas = base.target
rotulos = base.target_names

knn = KNeighborsClassifier(n_neighbors = 1)
knn.fit(entradas, saidas)
knn.predict([[5.1,3.1,1.4,0.2]])

especie = knn.predict([[5.9,3,5.1,1.8]])[0]
print(especie)
rotulos[especie]

etreino, eteste, streino, steste = train_test_split(entradas,
                                                    saidas,
                                                    test_size = 0.25)
knn.fit(etreino, streino)
previsor = knn.predict(eteste)

margem_acertos = metrics.accuracy_score(steste, previsor)

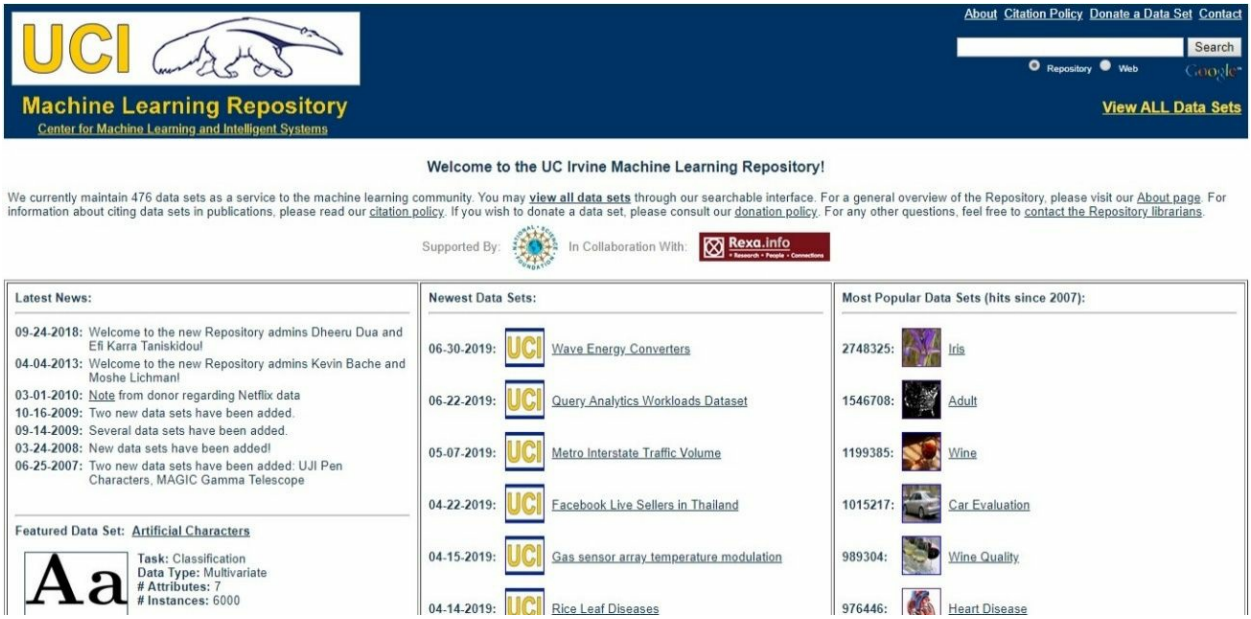
valores_k = {}
k=1
while k < 25:
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(etreino, streino)
    previsores_k = knn.predict(eteste)
    acertos = metrics.accuracy_score(steste, previsores_k)
    valores_k[k] = acertos
    k += 1

plt.plot(list(valores_k.keys()),
         list(valores_k.values()))
plt.xlabel('Valores de K')
plt.ylabel('Performance')
plt.show()

regl = LogisticRegression()
regl.fit(etreino, streino)
regl.predict([[6.2,3.4,5.4,2.3]])
regl.predict_proba([[6.2,3.4,5.4,2.3]])
previsor_regl = regl.predict(eteste)
margem_acertos_regl = metrics.accuracy_score(steste, previsor_regl)
```

Classificação Multiclasse via Rede Neural Artificial – Iris Dataset

Ao longo deste livro estaremos usando alguns datasets de exemplo disponibilizados em domínio público por meio de repositórios. Um dos maiores repositórios é o UCI Machine Learning Repository, mantido por algumas instituições educacionais e por doadores, conta atualmente com mais de 470 datasets organizados de acordo com o tipo de aprendizado de máquina a ser estudado e treinado.



Na página inicial é possível ver que, da listagem de datasets mais populares o Iris ocupa a primeira posição, até mesmo em outras comunidades ele costuma ser o mais popular, sendo o passo inicial para a maioria dos cientistas de dados que iniciam seus estudos em aprendizado de máquina.

UCI

Machine Learning Repository

Center for Machine Learning and Intelligent Systems

Iris Data Set

Download: Data Folder, Data Set Description

Abstract: Famous database; from Fisher, 1936



Data Set Characteristics:	Multivariate	**Number of Instances:**	150	**Area:**	Life
Attribute Characteristics:	Real	**Number of Attributes:**	4	**Date Donated**	1988-07-01
Associated Tasks:	Classification	**Missing Values?**	No	**Number of Web Hits:**	2748327

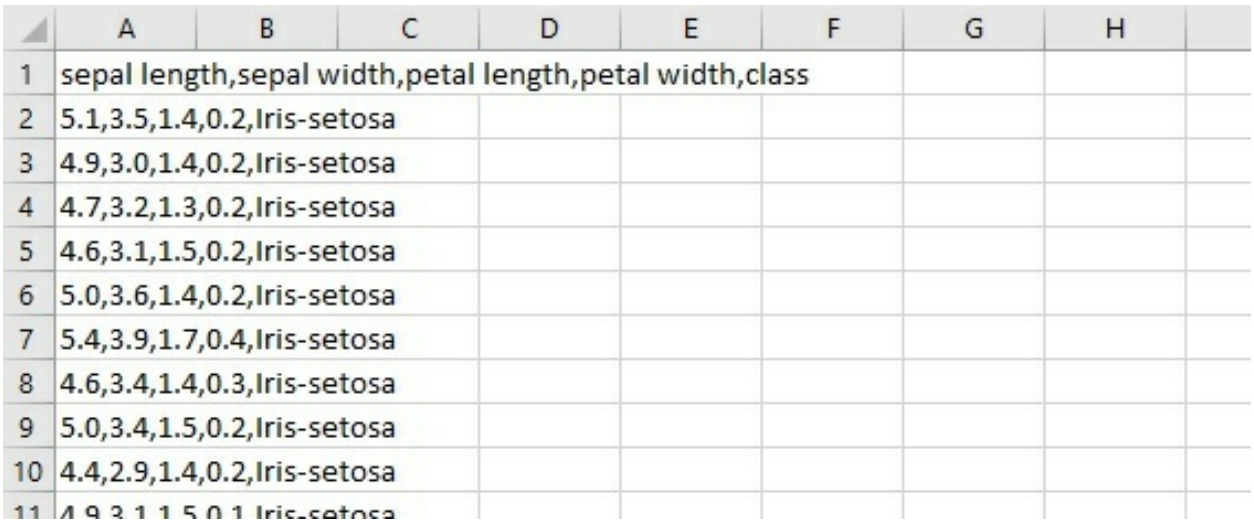
Source:

Creator:

R.A. Fisher

Abrindo a página dedicada ao Iris Dataset podemos ter uma descrição detalhada sobre do que se trata esse banco de dados assim como suas características. O que de imediato nos importa é que, como trabalhado anteriormente, trata-se de um banco de dados com 150 amostras das flores do tipo iris, divididas em 4 atributos previsores que usaremos para classificar tais amostras em 3 tipos de plantas diferentes.

A única diferença por hora é que antes estávamos usando a base de dados exemplo contida na biblioteca SKLearn, agora iremos aprender como trabalhar com dados a partir de planilhas Excel, que será o mais comum eu seu dia-a-dia. Na própria página da Iris Dataset é possível fazer o download do banco de dados que iremos utilizar, uma vez feito o download do arquivo iris.csv podemos dar início ao código.



	A	B	C	D	E	F	G	H
1	sepal length,sepal width,petal length,petal width,class							
2	5.1,3.5,1.4,0.2,Iris-setosa							
3	4.9,3.0,1.4,0.2,Iris-setosa							
4	4.7,3.2,1.3,0.2,Iris-setosa							
5	4.6,3.1,1.5,0.2,Iris-setosa							
6	5.0,3.6,1.4,0.2,Iris-setosa							
7	5.4,3.9,1.7,0.4,Iris-setosa							
8	4.6,3.4,1.4,0.3,Iris-setosa							
9	5.0,3.4,1.5,0.2,Iris-setosa							
10	4.4,2.9,1.4,0.2,Iris-setosa							
11	4.9,3.1,1.5,0.1,Iris-setosa							

Abrindo o arquivo iris.csv em nosso Excel é possível verificar como os dados estão organizados nessa base de dados.

Partindo para o Código:

Iris Dataset Densa.py

```
1 import pandas as pd
2
```

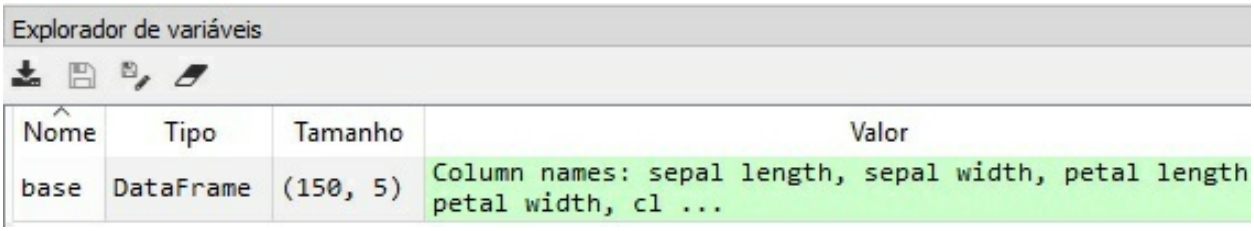
Todo processo se inicia com a importação das bibliotecas e módulos que iremos utilizar ao longo de nosso código. Para este exemplo usaremos a biblioteca Pandas, o processo de importação segue os mesmos moldes de sempre, por meio do comando import assim como a referência pelo comando as como no exemplo acima.

```
3 base = pd.read_csv('iris.csv')
4
```

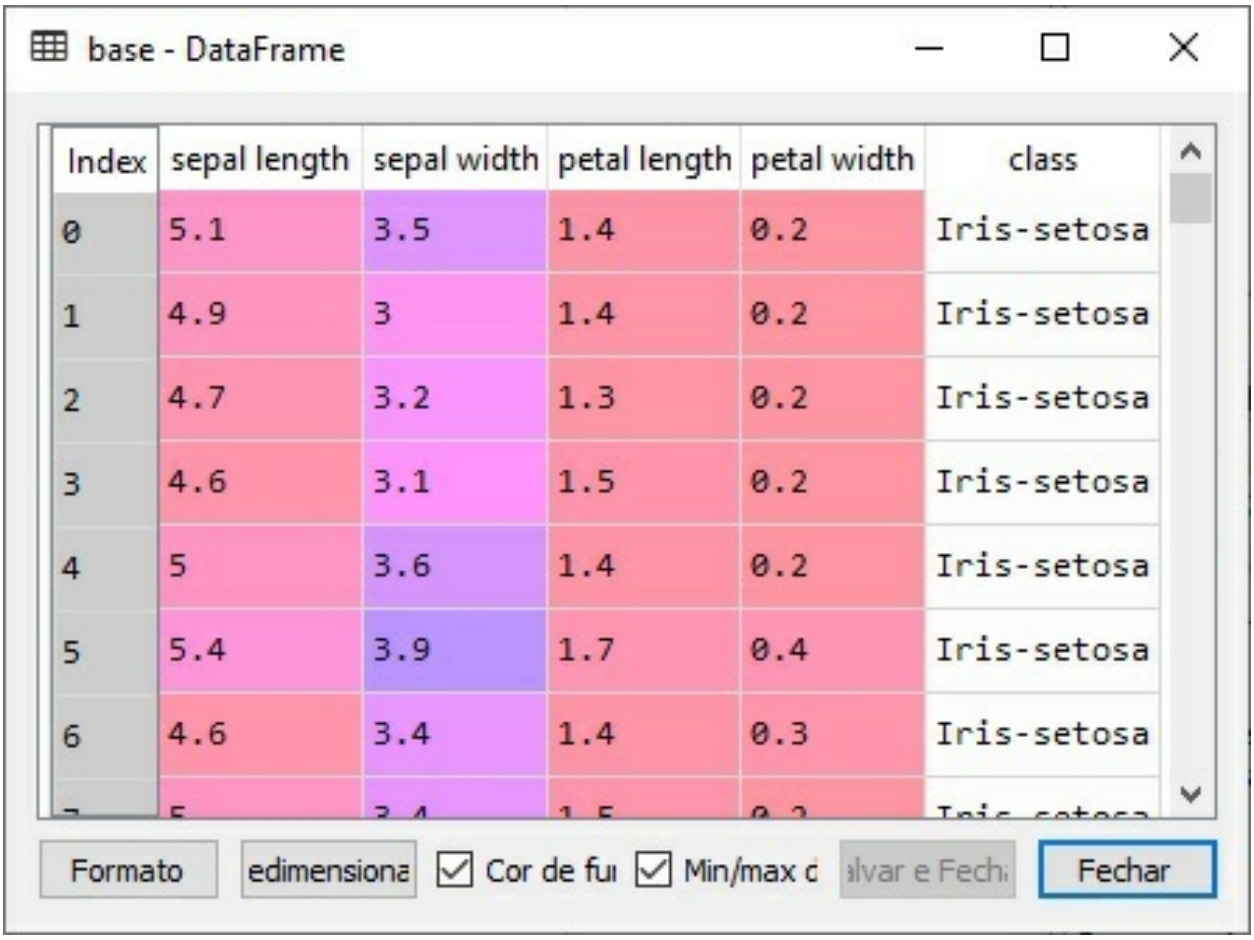
Em seguida criamos uma variável de nome base, por meio da função read_csv( ) do pandas conseguimos fazer a importação dos dados a partir de uma planilha Excel. Note que no parâmetro, em formato de string passamos o nome do arquivo assim como sua extensão, caso você tenha o arquivo com outro nome basta fazer a importação com o nome de seu arquivo.

Outro ponto interessante é que nesse formato o interpretador busca o arquivo no mesmo local onde está salvo seu arquivo de código .py, caso você tenha salvo em uma pasta diferente é necessário colocar todo o caminho

em forma de string como parâmetro.



Se o processo de importação ocorreu da maneira certa no Explorador de Variáveis será possível ver a variável base assim como explorar seu conteúdo de forma visual.



Repare que o objeto em questão é um DataFrame, este é um modelo parecido com o de matriz usado no exemplo anterior, onde tínhamos uma array do tipo Numpy, aqui, o sistema de interpretador do Pandas transforma os dados em um DataFrame, a grosso modo a grande diferença é que um DataFrame possui uma modelo de indexação próprio que nos será útil posteriormente quando for necessário trabalhar com variáveis do tipo

Dummy em exemplos mais avançados.

Por hora, você pode continuar entendendo que é uma forma de apresentação dos dados com linhas e colunas, parecido com o modelo matricial.

```
entradas = base.iloc[:, 0:4].values
saidas = base.iloc[:, 4].values
```

Logo após criamos nossas variáveis que armazenarão os dados separados como características das amostras (entradas) assim como o tipo de planta (saidas). Para isso criamos nossa variável entradas que por meio da função .iloc[ ] e do comando .values pega todos valores contidos em todas as linhas e das 4 primeiras colunas de base (desconsiderando a coluna de índice).

O mesmo é feito com nossa variável saidas, por meio da mesma função pega como atributo todos os valores de todas as linhas, mas somente da 4ª coluna (lembrando que as colunas com informação começam em 0, a 4ª coluna é a que contém os dados de saída).

Nome	Tipo	Tamanho	Valor
base	DataFrame	(150, 5)	Column names: sepal length, sepal width, petal le petal width, cl ...
entradas	float64	(150, 4)	[[5.1 3.5 1.4 0.2] [4.9 3. 1.4 0.2]
saidas	object	(150,)	ndarray object of numpy module

Dessa forma separamos os dados de entrada e de saída de nossa base de dados, permitindo o cruzamento desses dados ou a aplicação de funções sobre os mesmos.

```
from sklearn.preprocessing import LabelEncoder

labelencoder = LabelEncoder()
```

Quando estamos fazendo nossos ajustes iniciais, na chamada fase de polimento dos dados, um dos processos mais comuns a se fazer é a conversão de nossos dados de base, todos para o mesmo tipo. Raciocine que é impossível fazer, por exemplo, uma multiplicação entre texto e número.

Até o momento os dados que separamos estão em duas formas diferentes, os atributos de largura e comprimento das pétalas e sépalas em

nossos dados de entrada são numéricos enquanto os dados de saída, referente a nomenclatura dos tipos de plantas classificadas, estão em formato de texto. Você até pode tentar fazer o cruzamento desses dados, mas terá um traceback mostrando a incompatibilidade entre os mesmos.

Sendo assim, usaremos uma ferramenta capaz de converter texto em número para que possamos finalmente aplicar funções sobre os mesmos. A ferramenta em questão é a LabelEncoder, do módulo preprocessing da biblioteca sklearn. O processo de importação é feito como de praxe. Em seguida criamos uma variável labelencoder que inicializa essa ferramenta.

Nome	Tipo	Tamanho	Valor
base	DataFrame	(150, 5)	Column names: sepal length, sepal width, petal le petal width, cl ...
entradas	float64	(150, 4)	[[5.1 3.5 1.4 0.2] [4.9 3. 1.4 0.2]
saidas	object	(150,)	ndarray object of numpy module

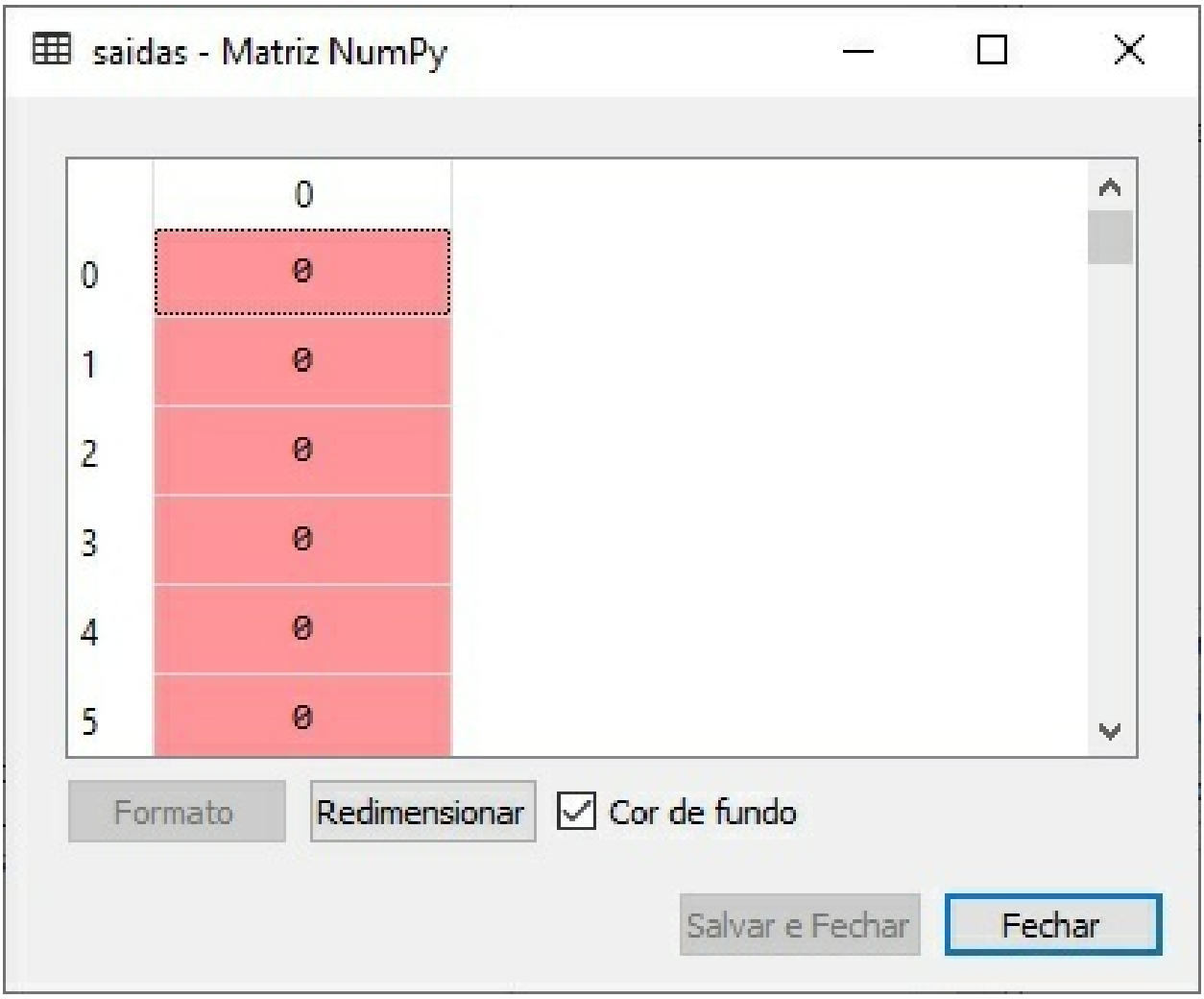
Repare no explorador de variáveis que por hora o objeto saidas não é reconhecido corretamente pelo Numpy.

```
11 saidas = labelencoder.fit_transform(saidas)
12
```

Aplicando a função .fit_transform( ) de labelencoder sobre nossa variável saidas fazemos a devida conversão de dados do formato string para int.

Nome	Tipo	Tamanho	Valor
base	DataFrame	(150, 5)	Column names: sepal length, sepal width, petal le petal width, cl ...
entradas	float64	(150, 4)	[[5.1 3.5 1.4 0.2] [4.9 3. 1.4 0.2]
saidas	int32	(150,)	[0 0 0 ... 2 2 2]

Executada a linha de código anterior, note que agora saidas possui seus dados em formato numérico, dentro de um intervalo entre 0 e 2.



Na sequência, outro processo que teremos de fazer, uma vez que processaremos nossos dados em uma rede neural densa, é a criação de variáveis do tipo Dummy. Variáveis Dummy são variáveis que criam uma indexação para nossa saída.

Lembre-se que aqui estamos classificando 150 amostras em 3 tipos diferentes, porém os neurônios da camada de saída retornarão valores 0 ou 1. Dessa forma, precisamos criar um padrão de indexação que faça sentido tanto para o usuário quanto para o interpretador.

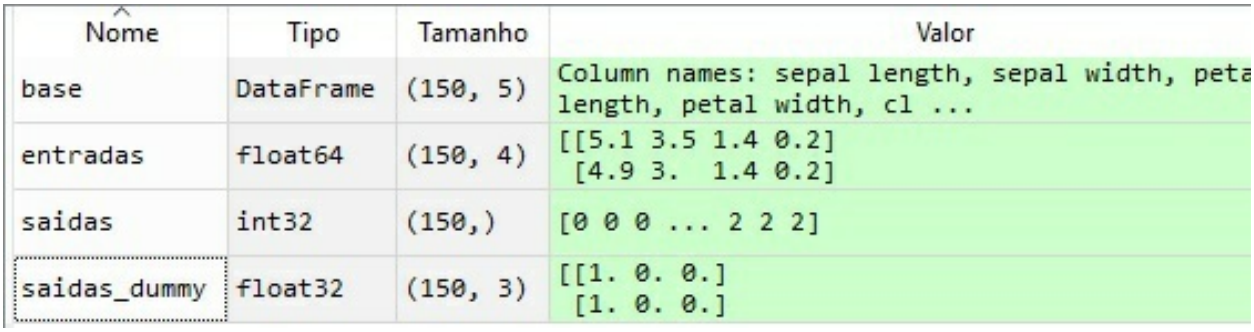
Em outras palavras, teremos de criar um padrão numérico binário para 3 saídas diferentes de acordo, nesse caso, com o tipo de planta. Por exemplo

Tipo de Planta	Neurônio 1	Neurônio 2	Neurônio 3

Setosa	0	0	1
Versicolor	0	1	0
Virginica	1	0	0

```
13 from keras.utils import np_utils
14
15 saidas_dummy = np_utils.to_categorical(saidas)
```

A nível de código, fazemos a importação da ferramenta np_utils, parte do módulo utils da biblioteca keras. Em seguida criamos uma variável de nome saidas_dummy que recebe sobre si como atributo a função np_utils.to_categorical( ) para que sejam criados os respectivos números de indexação como na tabela exemplo acima, para cada uma das amostras da base de dados.



Nome	Tipo	Tamanho	Valor
base	DataFrame	(150, 5)	Column names: sepal length, sepal width, peta length, petal width, cl ...
entradas	float64	(150, 4)	[[5.1 3.5 1.4 0.2] [4.9 3. 1.4 0.2]]
saidas	int32	(150,)	[0 0 0 ... 2 2 2]
saidas_dummy	float32	(150, 3)	[[1. 0. 0.] [1. 0. 0.]

Selecionando e executando mais esse bloco de código podemos ver que foi criada a variável saidas_dummy, abrindo-a você verá o padrão como o da tabela, para cada uma das 150 amostras.

```
17 from sklearn.model_selection import train_test_split
18
19 etreino, eteste, streino, steste = train_test_split(entradas,
20                                                    saidas_dummy,
21                                                    test_size = 0.25)
```

Dando sequência, como havíamos feito no exemplo anterior, aqui novamente podemos dividir nossa base de dados em partes e realizar testes com essas partes para avaliação da performance do modelo. Também é possível aplicar o processamento da rede diretamente sobre nossa base de dados, porém uma prática recomendável para se certificar da integridade dos dados é os processar em partes distintas e realizar a comparação.

Dessa forma, criamos quatro variáveis dedicadas a partes para

treino e teste de nossa rede neural. Logo etreino, eteste, streino, steste recebem como atributo train_test_split, que por sua vez recebe como parâmetro os dados contidos em entradas, saidas_dummy e por fim é definido que 25% das amostras sejam separadas para teste, enquanto os outros 75% serão separadas para treino.

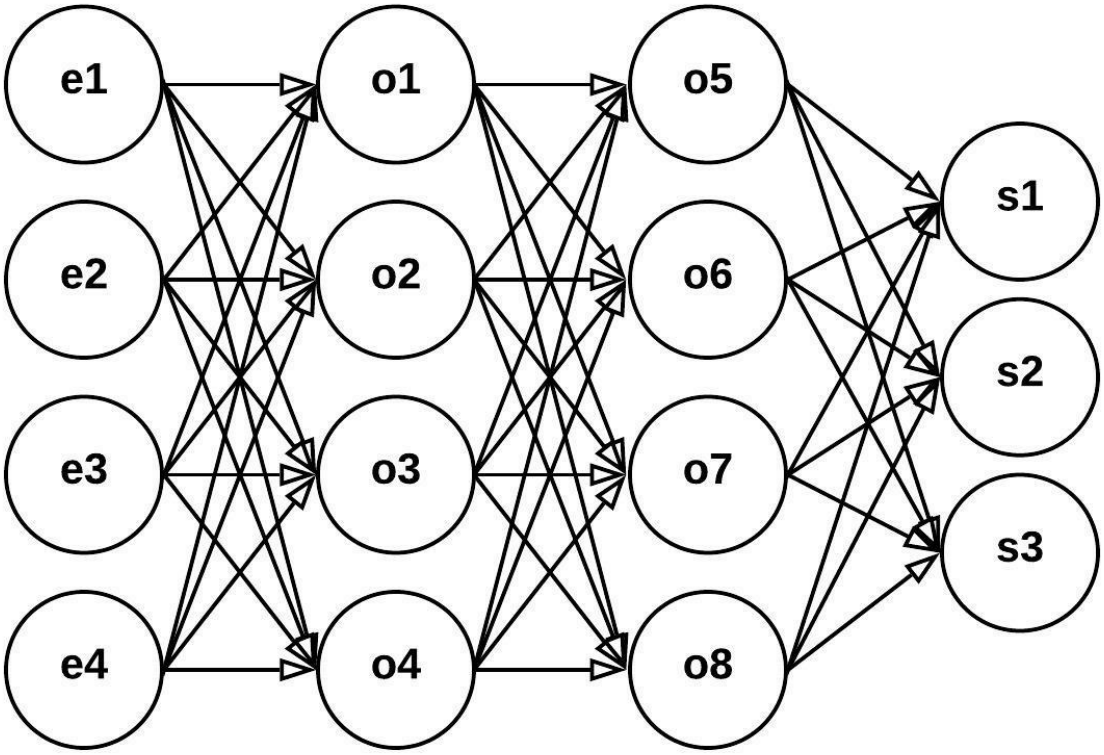
Nome	Tipo	Tamanho	Valor
base	DataFrame	(150, 5)	Column names: sepal length, sepal width, peta length, petal width, cl ...
entradas	float64	(150, 4)	[[5.1 3.5 1.4 0.2] [4.9 3. 1.4 0.2]]
eteste	float64	(38, 4)	[[5.8 2.7 4.1 1.] [6.7 3.3 5.7 2.5]]
etreino	float64	(112, 4)	[[6.7 2.5 5.8 1.8] [4.9 3. 1.4 0.2]]
saidas	int32	(150,)	[0 0 0 ... 2 2 2]
saidas_dummy	float32	(150, 3)	[[1. 0. 0.] [1. 0. 0.]]
steste	float32	(38, 3)	[[0. 1. 0.] [0. 0. 1.]

Se todo processo correu normalmente é possível visualizar no explorador de variáveis tais objetos.

```
23 from keras.models import Sequential
24 from keras.layers import Dense
```

Finalmente vamos dar início a criação da estrutura de nossa rede neural densa, como sempre, inicialmente é necessário fazer as devidas importações para que possamos usufruir das ferramentas adequadas.

Então do módulo models da biblioteca keras importamos Sequential, da mesma forma, do módulo layers importamos Dense. Sequential é um modelo pré-configurado onde conseguimos criar e fazer as devidas conexões entre camadas de neurônios, Dense por sua vez permite a conexão entre todos nós da rede assim como a aplicação de funções em todas etapas da mesma.



Numa representação visual como fizemos anteriormente com os perceptrons, uma rede densa como a que estaremos criando agora seguiria uma estrutura lógica como a da figura acima.

```
26 classificador = Sequential()
27 classificador.add(Dense(units = 4,
28                         activation = 'relu',
29                         input_dim = 4))
```

Inicialmente criamos uma variável de nome classificador, que recebe como atributo a inicialização da ferramenta Sequential, aqui, por enquanto, sem parâmetros mesmo. Em seguida por meio da função .add( ) começamos de fato a criação da camada de entrada e da sua comunicação com a primeira camada oculta.

Para isso passamos como parâmetro para add Dense que por sua vez tem como parâmetros units = 4, ou seja, 4 neurônios na primeira camada oculta, activation = ‘relu’ (Rectfied Linear Units, em tradução livre Unidades Linearmente Retificadas, algo como a StepFunction usada no exemplo anterior, porém entre camadas de redes neurais quando aplicamos uma

função degrau é por meio da relu), que é o método de ativação dessa camada, e por fim input_dim = 4, que é o parâmetro que determina quantos neurônios existem em nossa camada de entrada, aqui, 4 tipos de características das plantas representam 4 neurônios da camada de entrada.

```
30 classificador.add(Dense(units = 4,
31                         activation = 'relu'))
```

Logo após criamos mais uma camada oculta, também com 4 neurônios e também recebendo como função de ativação relu. Repare que aqui não é mais necessário especificar os neurônios de entrada, raciocine que esta camada é a conexão entre a primeira camada oculta e a camada de saída dessa rede neural.

```
32 classificador.add(Dense(units = 3,
33                         activation = 'softmax'))
```

Dando sequência criamos a camada de saída dessa rede, nos mesmos moldes das anteriores, porém com a particularidade que na camada de saída temos 3 neurônios, já que estamos classificando as amostras em 3 tipos, e o método de ativação agora é o ‘softmax’, método esse que segundo a documentação do keras é o que mais se aproxima da função sigmoide que usamos anteriormente, a diferença basicamente é que sigmoid oferece a probabilidade de ser de um tipo ou de outro (2 tipos), enquanto softmax suporta mostrar a probabilidade de uma amostra ser classificada em 3 ou mais tipos.

```
34 classificador.compile(optimizer = 'adam',
35                       loss = 'categorical_crossentropy',
36                       metrics = ['categorical_accuracy'])
```

Prosseguindo, por meio da função .compile( ) criamos o compilador para nossa rede neural, nesta etapa é necessário passar três parâmetros, o primeiro deles, optimizer = ‘adam’, com o nome autoexplicativo, aplica alguns métodos para otimização dos resultados dos processamentos, loss = ‘categorical_crossentropy’ é a nossa função de perda, raciocine que em uma rede neural há uma pequena parcela das amostras que podem ter dados inconsistentes ou erros de processamento, estas caem nessa categoria, o parâmetro da função de perda por sua vez tentará pegar essas amostras fora da curva e tentar encaixar de acordo com a semelhança/proximidade, a outras amostras que foram processadas corretamente, por fim metrics = [‘categorical_accuracy’] faz a avaliação interna do modelo, mensurando sua

precisão quanto ao processo de categorizar corretamente as amostras.

```
37 classificador.fit(etreino,
38                   streino,
39                   batch_size = 10,
40                   epochs = 1000)
```

Por fim, por meio da função .fit( ) faremos a alimentação de nossa rede e definiremos os parâmetros de processamento dela. Sendo assim, .fit( ) recebe como parâmetros os dados de etreino e de streino, batch_size = 10 diz respeito a taxa de atualização dos pesos, ou seja, de quantas em quantas amostras serão testadas, corrigidas e testadas e por fim epochs = 1000 define quantas vezes a rede neural será executada.

Esses parâmetros na prática devem inclusive serem modificados e testados para tentar sempre encontrar a melhor performance do algoritmo. Em certos casos a taxa de atualização pode influenciar bastante a diminuição da margem de erro, assim como executar a rede neural mais vezes faz o que em algumas literaturas é chamado de aprendizagem por reforço, toda rede neural começa com rápidas atualizações e correções, porém ela chega em níveis onde após a execução de algumas milhares (ou até mesmo milhões) de vezes executada ela pode acabar estagnada em um valor.

```
112/112 [==============================] - 0s 125us/step - loss: 0.0585 -
categorical_accuracy: 0.9821
Epoch 996/1000
112/112 [==============================] - 0s 125us/step - loss: 0.0584 -
categorical_accuracy: 0.9732
Epoch 997/1000
112/112 [==============================] - 0s 125us/step - loss: 0.0584 -
categorical_accuracy: 0.9821
Epoch 998/1000
112/112 [==============================] - 0s 125us/step - loss: 0.0552 -
categorical_accuracy: 0.9821
Epoch 999/1000
112/112 [==============================] - 0s 125us/step - loss: 0.0551 -
categorical_accuracy: 0.9821
Epoch 1000/1000
112/112 [==============================] - 0s 143us/step - loss: 0.0562 -
categorical_accuracy: 0.9821
Out[ ]:
```

Selecionando todo bloco de código da estrutura da rede neural e executando, você pode acompanhar pelo terminal a mesma sendo executada, assim como acompanhar a atualização da mesma. Aqui, neste exemplo, após a execução da mesma 1000 vezes para o processo de classificação,

categorical_accuracy: 0.98 nos mostra que a taxa de precisão, a taxa de acertos na classificação das 150 amostras foi de 98%.

Tenha em mente que bases de dados grandes ou redes que são executadas milhões de vezes, de acordo com seu hardware essa rede pode levar um tempo considerável de processamento.

Aqui nesse exemplo, 150 amostras, 3 categorias, 1000 vezes processadas, toda tarefa é feita em poucos segundos, mas é interessante ter esse discernimento de que grandes volumes de dados demandam grande tempo de processamento.

Terminada a fase de processamento da rede neural, hora de fazer algumas avaliações de sua performance para conferir e confirmar a integridade dos resultados.

```
42 avalPerformance = classificador.evaluate(eteste, steste)
43
44 previsoes = classificador.predict(eteste)
45 previsoesVF = (previsoes > 0.5)
```

Para isso inicialmente criamos uma variável de nome avalPerformance que recebe como atributo a função .evaluate( ) de nosso classificador, tendo como parâmetros os dados de eteste e steste. Também criamos uma variável de nome previsoes, que recebe a função .predict( ) sobre os dados de eteste.

Por fim, criamos uma variável de nome previsoesVF que basicamente retorna os resultados de previsões em formato True ou False, por meio da condicional previsoes > 0.5.

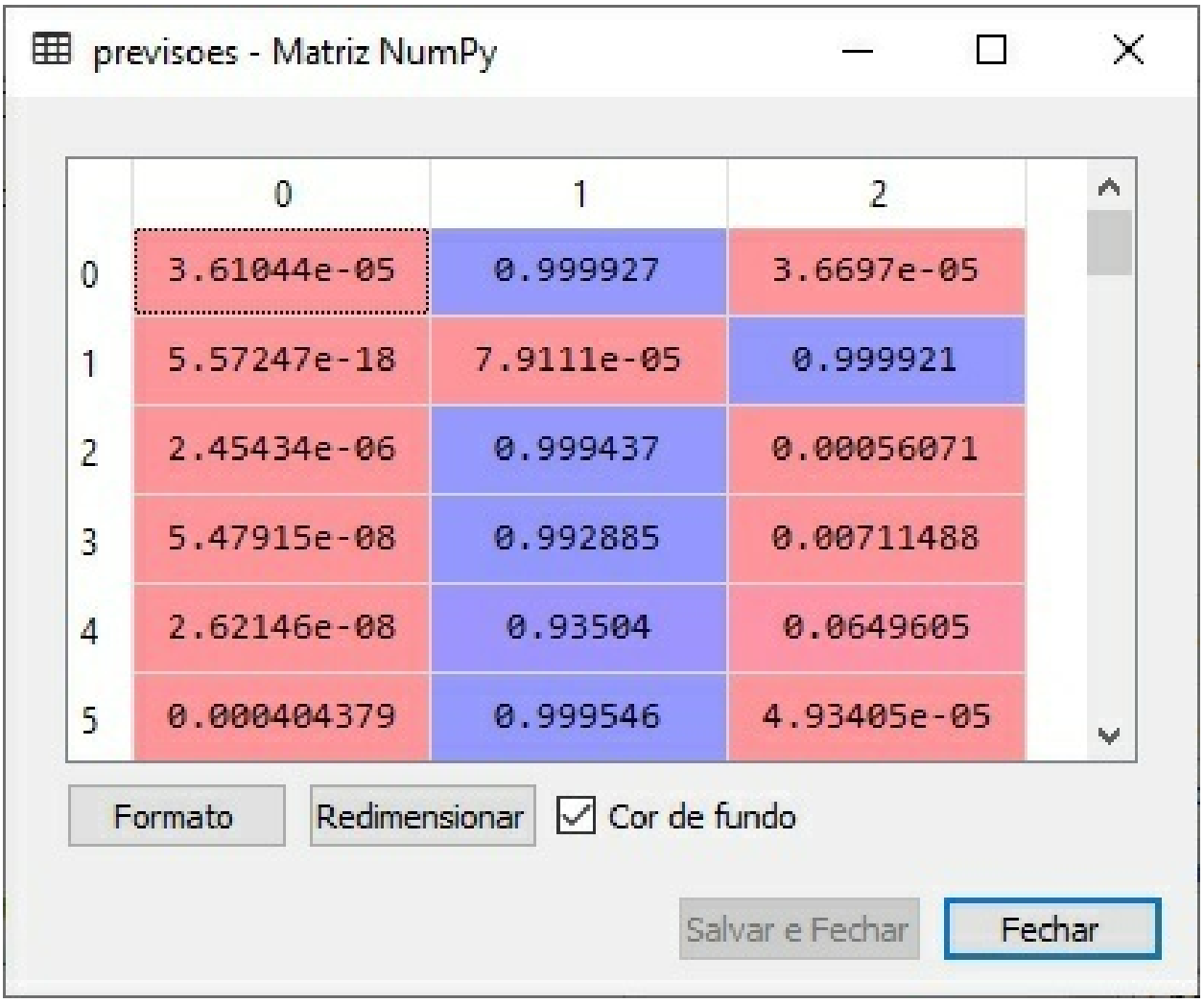
Nome	Tipo	Tamanho	Valor
avalPerformance	list	2	[0.03765642162608473, 0.9736842105263158]
base	DataFrame	(150, 5)	Column names: sepal length, sepal width, petal length, petal width, cl ...
entradas	float64	(150, 4)	[[5.1 3.5 1.4 0.2] [4.9 3. 1.4 0.2]]
eteste	float64	(38, 4)	[[5.8 2.7 4.1 1.] [6.7 3.3 5.7 2.5]]
etreino	float64	(112, 4)	[[6.7 2.5 5.8 1.8] [4.9 3. 1.4 0.2]]
previsoes	float32	(38, 3)	[[3.6104400e-05 9.9992716e-01 3.6696962e-0 [5.5724660e-18 7.9111029 ...
previsoesVF	bool	(38, 3)	[[False True False] [False False True]

Selecionando e executando esse bloco de código é possível ver a criação das devidas variáveis.

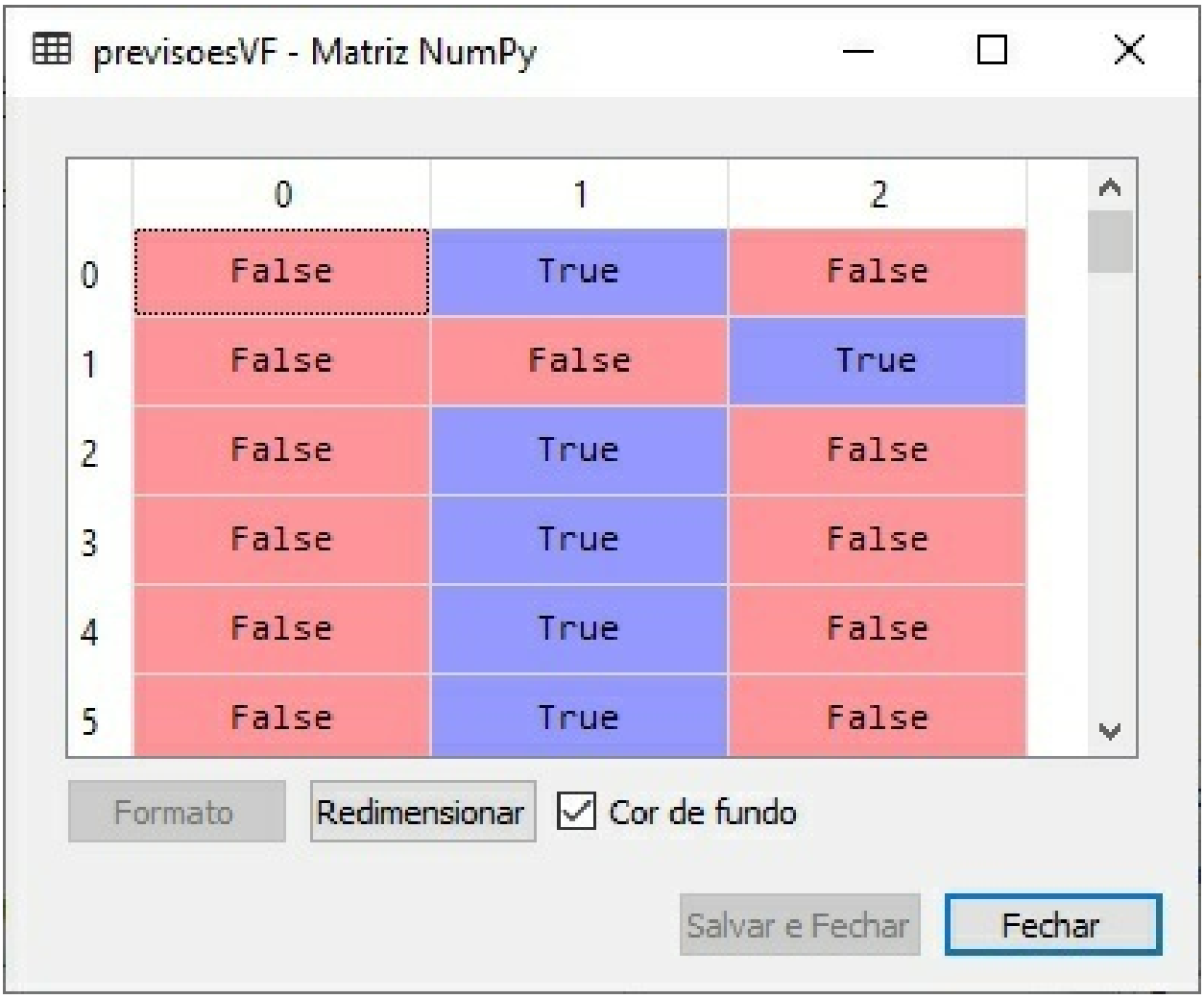


Abrindo avalPerformance temos dados para comparar com os do processamento da rede neural. A rede neural havia nos retornado uma taxa de precisão de 98%, enquanto aqui realizado testes de outras formas pela função .evaluate( ) o retorno foi de 97%. O retorno de loss function da rede havia sido de 0.05%, o da evaluate 0.03%.

O que confirma uma excelente precisão nos resultados dos processamentos, tenha em mente que essa fase de avaliação é dada até mesmo como opcional em algumas literaturas, porém para uma atividade profissional é imprescindível que você se certifique ao máximo da integridade de seus resultados.



Abrindo previsoes podemos ver a probabilidade de cada amostra ser de um determinado tipo, por exemplo, na linha 0, em azul, há o dado probabilístico de 99% de chance dessa amostra ser do tipo 1 (versicolor), na segunda linha, 99% de chance de tal amostra ser do tipo 2 (virginica) e assim por diante.



Abrindo previsoesVF podemos ver de forma ainda mais clara que a amostra da linha 0 é do tipo 1, que a amostra da linha 1 é do tipo 2, e assim subsequente para todas as amostras.

Outra metodologia de avaliação de nosso modelo é feita pela chamada Matriz de Confusão, método que nos mostrará quantas classificações foram feitas corretamente e as que foram classificadas erradas para cada saída. Para aplicar a matriz de confusão em nosso modelo precisaremos fazer algumas alterações em nossos dados, assim como fizemos lá no início na fase de polimento.

```
47 import numpy as np
48
49 steste2 = [np.argmax(t) for t in steste]
50 previsoes2 = [np.argmax(t) for t in previsoes]
```

Dessa vez importamos a biblioteca Numpy e a referenciamos como

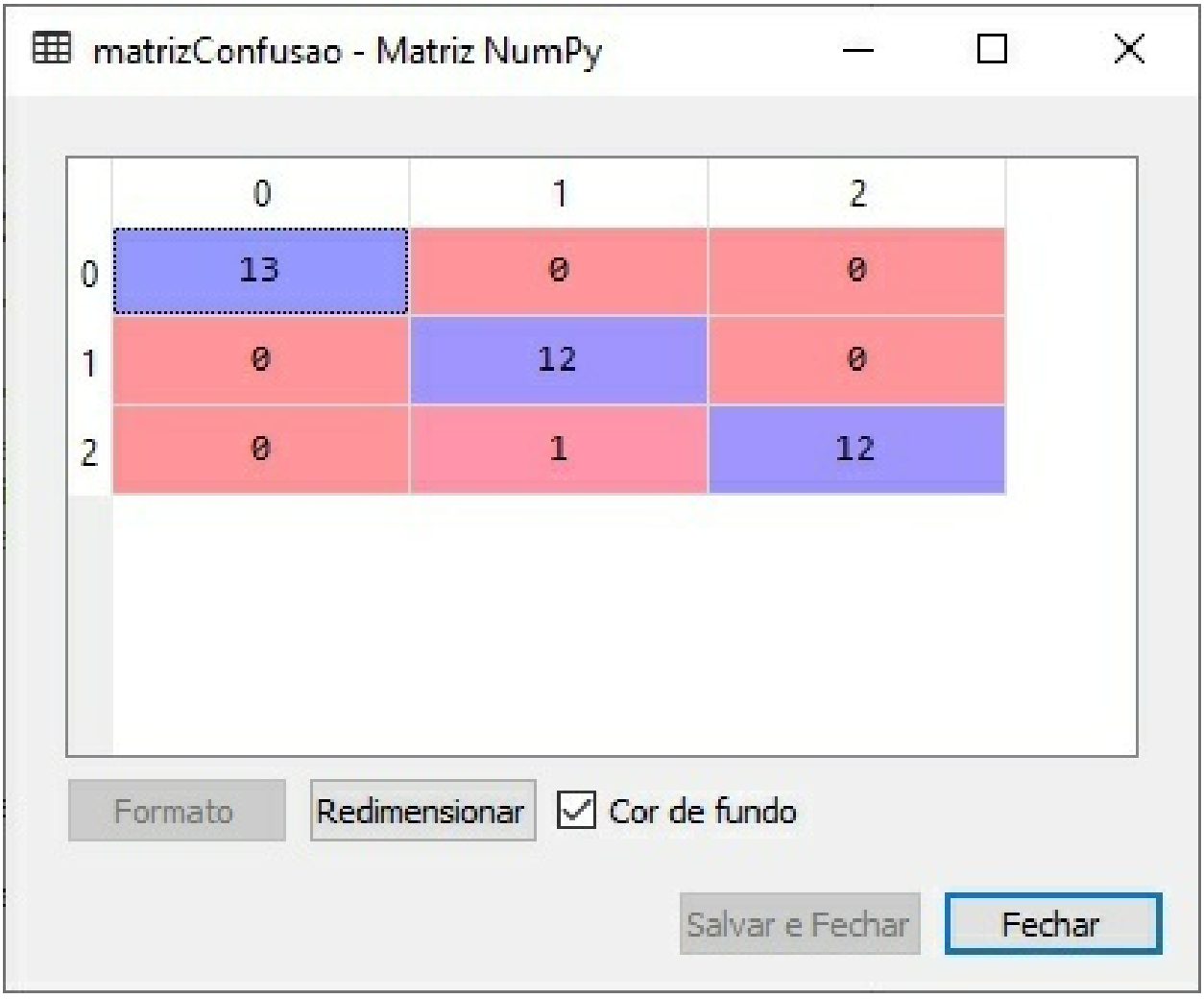
np. Em seguida criamos uma variável de nome steste2 que recebe como atributo a função np.argmax(t), que pega os valores de steste e gera uma indexação própria, contendo a posição da matriz onde a planta foi classificada. O mesmo é feito em previsoes e atribuído para previsoes2.

```
52 from sklearn.metrics import confusion_matrix
53
```

Logo a seguir fazemos mais uma importação necessária, dessa vez, da ferramenta confusion_matrix do módulo metrics da biblioteca sklearn.

```
54 matrizConfusao = confusion_matrix(previsoes2, steste2)
55
```

Dando sequência criamos uma variável de nome matrizConfusao que por sua vez realiza a aplicação dessa ferramenta sobre previsoes2 e steste2.



Selecionando e executando o bloco de código acima é possível ver que foi criada a variável matrizConfusao, explorando seu conteúdo podemos fazer a leitura, em forma de verificação que: Para o tipo de planta 0, foram classificadas 13 mostras corretamente, para o tipo de planta 1, foram classificadas 12 amostras corretamente, e para o tipo de planta 2, foram classificadas 12 amostras de maneira correta e 1 amostra de maneira incorreta.

Finalizando este modelo, outra prática muito usada que é interessante já darmos início a seu entendimento é a chamada Validação Cruzada. Este método é de suma importância para se certificar de que seus testes não estão "viciados", uma vez que uma rede neural mal configurada pode entrar em loops de repetição ou até mesmo retornar resultados absurdos. A técnica em si consiste em pegar nossa base de dados inicial, separar a mesma em partes iguais e aplicar o processamento da rede neural individualmente para cada uma das partes, para por fim se certificar que o padrão de resultados está realmente correto.

```
56 from sklearn.model_selection import cross_val_score
57
```

Dessa vez importamos a ferramenta cross_val_score do módulo model_selection da biblioteca sklearn.

```
58 def valCruzada():
59     classificadorValCruzada = Sequential()
60     classificadorValCruzada.add(Dense(units = 4,
61                                activation = 'relu',
62                                input_dim = 4))
63     classificadorValCruzada.add(Dense(units = 4,
64                                activation = 'relu'))
65     classificadorValCruzada.add(Dense(units = 3,
66                                activation = 'softmax'))
67     classificadorValCruzada.compile(optimizer = 'adam',
68                               loss = 'categorical_crossentropy',
69                               metrics = ['categorical_accuracy'])
70     return classificadorValCruzada
```

E m seguida criamos uma rede interna em forma de função chamada ValCruzada( ), sem parâmetros mesmo. Repare que sua estrutura interna é a mesma da rede criada anteriormente com a exceção que agora não a alimentaremos por meio da função .fit( ) mas sim fazendo o uso do KerasClassifier.

```
72 from keras.wrappers.scikit_learn import KerasClassifier
73
```

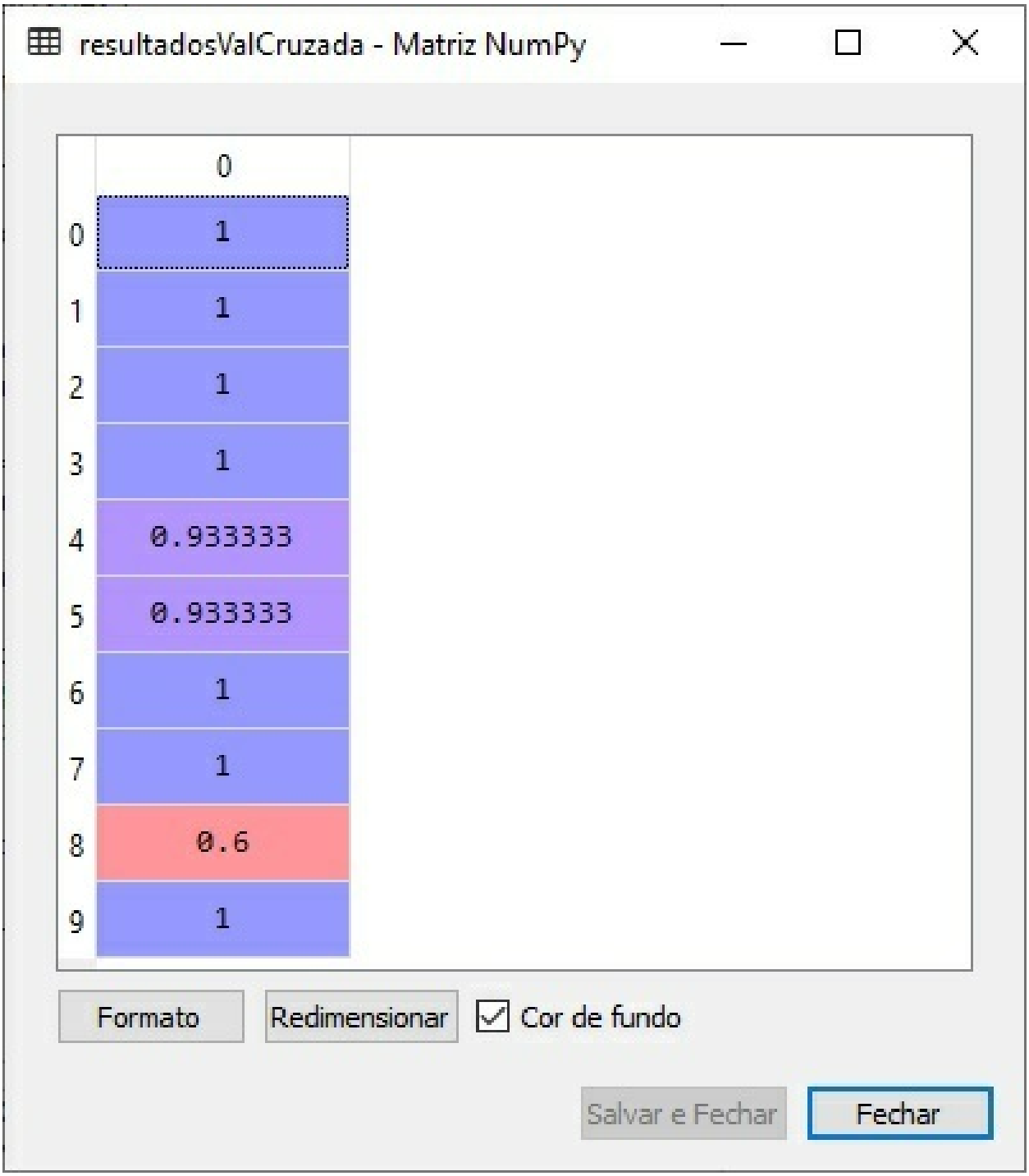
Como o KerasClassifier ainda não havia sido incorporado em nosso código é necessário fazer a importação do mesmo, nos mesmos moldes de sempre.

```
74 classificadorValCruzada = KerasClassifier(build_fn = valCruzada,
75                                           epochs = 1000,
76                                           batch_size = 10)
```

Criamos uma variável a executar a ferramenta KerasClassifier de nome classificadorValCruzada. Tal ferramenta por sua vez recebe como parâmetros buind_fn = valCruzada, o que significa que a função de ativação desse classificador é a própria rede neural criada anteriormente de nome valCruzada, outro parâmetro é epochs = 1000, ou seja, definindo que o processamento da rede será feito mil vezes e por fim batch_size = 10, parâmetro que defini de quantos em quantos pesos seus valores serão corrigidos.

```
78 resultadosValCruzada = cross_val_score(estimator = classificadorValCruzada,
79                                         X = entradas,
80                                         y = saidas,
81                                         cv = 10,
82                                         scoring = 'accuracy')
```

Na sequência criamos uma variável chamada resultadosValCruzada que executará a própria ferramenta cross_val_score, que por sua vez recebe como parâmetros estimator = classificadorValCruzada, o que em outras palavras nada mais é do que aplicar essa função sobre o processamento da rede valCruzada( ), X = entradas e y = saidas, auto sugestivo, variáveis temporárias com os dados contidos em entradas e saidas, cv = 10, parâmetro que define em quantas partes nossa base de dados será dividida igualmente para ser processada pela rede e por fim scoring = ‘accuracy’ define o método a ser formatados os resultados, aqui, com base em precisão de resultados.



Abrindo a variável resultadosValCruzada é possível ver de forma bruta, mas fácil conversão para percentual, os resultados do processamento da rede sobre cada amostra. Note que aqui houve inclusive uma exceção onde o processamento em cima da amostra nº 8 teve uma margem de acerto de apenas 60%, enquanto para todas as outras a margem ficou dentro de um intervalo entre 93% e 100%. Isto pode acontecer por diversos fatores, os mais

comuns, erros de leitura dos dados da base ou aplicação de parâmetros errados sobre a rede neural.

```
84 media = resultadosValCruzada.mean()
85 desvio = resultadosValCruzada.std()
```

Por fim, apenas para apresentar nossos dados de forma mais clara, criamos uma variável de nome media que recebe o valor médio entre todos valores obtidos em resultadosValCruzada, assim como é interessante apresentar o cálculo do desvio (algo como a margem de erro em percentual), aqui inclusive, este valor vai sofrer um grande impacto devido ao resultado da validação cruzada sobre a 8ª amostra.

Nome	Tipo	Tamanho	Valor
base	DataFrame	(150, 5)	Column names: sepal length, sepal wi
desvio	float64	1	0.11850925889754119
entradas	float64	(150, 4)	[[5.1 3.5 1.4 0.2] [4.9 3. 1.4 0.2]
eteste	float64	(38, 4)	[[6.8 3.2 5.9 2.3] [5.1 3.8 1.6 0.2]
etreino	float64	(112, 4)	[[6.1 3. 4.9 1.8] [6.5 3. 5.2 2.]
matrizConfusao	int64	(3, 3)	[[0 0 0] [13 10 15]
media	float64	1	0.9466666666666667

Via explorador de variáveis podemos ver facilmente os valores de media (taxa de acerto de 94%) e de desvio (11% de margem de erro para mais ou para menos, o que é um número absurdo, porém justificável devido ao erro de leitura da 8ª amostra anterior).

Código Completo:

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import LabelEncoder
from keras.utils import np_utils
from sklearn.model_selection import train_test_split
from keras.models import Sequential
from keras.layers import Dense
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import cross_val_score
from keras.wrappers.scikit_learn import KerasClassifier

base = pd.read_csv('iris.csv')
entradas = base.iloc[:, 0:4].values
saidas = base.iloc[:, 4].values

labelencoder = LabelEncoder()
saidas = labelencoder.fit_transform(saidas)
saidas_dummy = np_utils.to_categorical(saidas)

etreino, eteste, streino, steste = train_test_split(entradas,
                                                    saidas_dummy,
                                                    test_size = 0.25)
classificador = Sequential()
classificador.add(Dense(units = 4,
                        activation = 'relu',
                        input_dim = 4))
classificador.add(Dense(units = 4,
                        activation = 'relu'))
classificador.add(Dense(units = 3,
                        activation = 'softmax'))
classificador.compile(optimizer = 'adam',
                      loss = 'categorical_crossentropy',
                      metrics = ['categorical_accuracy'])
classificador.fit(etreino,
                  streino,
                  batch_size = 10,
                  epochs = 1000)
```

```
avalPerformance = classificador.evaluate(eteste, steste)
previsoes = classificador.predict(eteste)
previsoesVF = (previsoes > 0.5)
steste2 = [np.argmax(t) for t in steste]
previsoes2 = [np.argmax(t) for t in previsoes]
matrizConfusao = confusion_matrix(previsoes2, steste2)

def valCruzada():
    classificadorValCruzada = Sequential()
    classificadorValCruzada.add(Dense(units = 4,
                                      activation = 'relu',
                                      input_dim = 4))
    classificadorValCruzada.add(Dense(units = 4,
                                      activation = 'relu'))
    classificadorValCruzada.add(Dense(units = 3,
                                      activation = 'softmax'))
    classificadorValCruzada.compile(optimizer = 'adam',
                                    loss = 'categorical_crossentropy',
                                    metrics = ['categorical_accuracy'])
    return classificadorValCruzada

classificadorValCruzada = KerasClassifier(build_fn = valCruzada,
                                          epochs = 1000,
                                          batch_size = 10)

resultadosValCruzada = cross_val_score(estimator = classificadorValCruzada,
                                       X = entradas,
                                       y = saidas,
                                       cv = 10,
                                       scoring = 'accuracy')
media = resultadosValCruzada.mean()
desvio = resultadosValCruzada.std()
```

Classificação Binária via Rede Neural Artificial – Breast Cancer Dataset

Uma das aplicações mais comuns em data science, como você deve ter percebido, é a classificação de dados para os mais diversos fins. Independentemente do tipo de dados e da quantidade de dados a serem processados, normalmente haverá uma fase inicial onde será feito o tratamento desses dados para que se possa extrair informações relevantes dos mesmos.

Tenha em mente que por vezes o problema que estaremos abstraindo é a simples classificação de dados, porém haverão situações onde a classificação será apenas parte do tratamento dos mesmos. Se tratando de redes neurais é comum fazer o seu uso para identificação de padrões a partir de dados alfanuméricos ou imagens e com base nesses padrões realizar aprendizado de máquina para que seja feita a análise de dados.

Seguindo esse raciocínio lógico, uma das aplicações que ganhou destaque e vem evoluindo exponencialmente é a aplicação de redes neurais dentro da imaginologia médica, exames de imagem computadorizados como raios-x, tomografia computadorizada, ressonância magnética hoje possuem tecnologia para se obter imagens de alta definição fidedignas da anatomia humana para que se possam investigar possíveis patologias.

Ainda dentro da radiologia médica, uma das mais importantes aplicações de redes neurais foi para contribuir com a identificação de diferentes tipos de câncer a partir de exames de imagem. Tenha em mente que a ideia não é jamais substituir o profissional médico radiologista que é o especialista nisso, mas oferecer ferramentas que auxiliem ou de alguma forma aumente a precisão de seus diagnósticos para que os pacientes recebam tratamentos mais adequados.

Como exemplo deste tipo de aplicação de rede neural, usaremos um dataset bastante conhecido que com base num banco de dados de mais de 560 imagens de exames de mamografia para treinar uma rede neural artificial que possa a partir dessa estrutura, classificar imagens quanto a presença ou não de

tumores. Por fim esse modelo pode ser adaptado para a partir de novas imagens realizar tal classificação, ou salvo para novos problemas computacionais que de alguma forma possam envolver classificação binária, classificação do tipo “ou uma coisa ou outra”.

Lembrando que, como mencionado algumas vezes nos capítulos anteriores, estaremos trabalhando de forma procedural, cada vez será usado um referencial teórico mais enxuto enquanto avançaremos para exemplos práticos de maior complexidade.

```
1 import pandas as pd
2
3 entradas = pd.read_csv('entradas-breast.csv')
4 saidas = pd.read_csv('saidas-breast.csv')
```

Como sempre o processo se inicia com a importação das bibliotecas, módulos e ferramentas a serem utilizados. De imediato importamos a biblioteca Pandas e a referenciamos como pd, para que na sequência possamos por meio de sua função .read_csv( ) importar dados a partir de planilhas Excel.

Nome	Tipo	Tamanho	Valor
entradas	DataFrame	(569, 30)	Column names: radius_mean, texture_mean, perimeter_mean, area_mean ...
saidas	DataFrame	(569, 1)	Column names: 0

Se o processo de importação correu como esperado no explorador de variáveis é possível visualizar tais variáveis. Como estamos trabalhando inicialmente com a biblioteca pandas note que os dados são importados e reconhecidos como DataFrame, em outras bibliotecas essa nomenclatura é a usual matriz. Vale lembrar que para importação acontecer corretamente o arquivo deve estar na mesma pasta onde você salvou seu arquivo de código, ou especificar o caminho completo no parâmetro de importação.



Abrindo entradas podemos ver que de fato, existem 569 amostras, com 30 parâmetros para cada uma delas (valores individuais, de perda e de média para raio, textura, perímetro, área, borramento, compactação, concavidade, pontos de cavidade, simetria, dimensão fractal). Com base em todos esses dados as amostras serão classificadas como positivo ou negativo para o diagnóstico de câncer.

```
6 from sklearn.model_selection import train_test_split
7
8 etreino, eteste, streino, steste = train_test_split(entradas,
9                                                     saidas,
10                                                    test_size = 0.25)
```

Em seguida realizamos a importação da ferramenta train_test_split e dividimos nossas amostras de nossa base de dados para que possam ser

treinadas e testadas individualmente por nossa rede neural. Lembrando que esta etapa é uma fase de certificação da integridade de nossos processos e resultados, valores próximos em comparação mostram que nossa rede está configurada corretamente.

```
12 import keras
13 from keras.models import Sequential
14 from keras.layers import Dense, Dropout
```

Em seguida fazemos mais umas importações, primeiro, da biblioteca keras, depois dos módulos Sequential e Dense que como já havíamos feito outrora, servem como estruturas pré-configuradas de nossa rede multicamada interconectada. O único diferencial aqui é que do módulo layers da biblioteca keras também importamos Dropout. Esse módulo usaremos na sequência, por hora, raciocine que quando estamos trabalhando com redes neurais com muitas camadas, ou muitos neurônios por camada, onde esses tem valores de dados muito próximos, podemos determinar que aleatoriamente se pegue um percentual desses neurônios e simplesmente os subtraia da configuração para ganharmos performance.

Em outras palavras, imagine uma camada com 30 neurônios, onde cada neurônio desse faz suas 30 conexões com a camada subsequente, sendo que o impacto individual ou de pequenas amostras deles é praticamente nulo na rede, sendo assim, podemos reduzir a quantia de neurônios e seus respectivos nós ganhando performance em processamento e não perdendo consistência dos dados.

```
16 classificador = Sequential()
17 classificador.add(Dense(units = 16,
18                         activation = 'relu',
19                         kernel_initializer = 'random_uniform',
20                         input_dim = 30))
21 classificador.add(Dense(units = 1,
22                         activation = 'sigmoid'))
23 classificador.compile(optimizer = 'adam',
24                       loss = 'binary_crossentropy',
25                       metrics = ['binary_accuracy'])
26 classificador.fit(etreino,
27                   streino,
28                   batch_size = 10,
29                   epochs = 100)
```

Logo após criamos aquela estrutura de rede neural que estamos começando a ficar familiarizados. Repare nas diferenças em relação ao

exemplo anterior, desta vez, aqueles 30 parâmetros de classificação de cada amostra é transformado em neurônios de nossa camada de entrada, a camada oculta por sua vez tem 16 neurônios e a camada de saída apenas um, pois trata-se de uma classificação binária (0 ou 1, 0 para negativo e 1 para positivo de câncer para uma devida amostra).

Outra grande diferença que agora nessa estrutura, na camada onde há a fase de compilação as funções de perda e métricas também são diferentes do modelo anterior ,agora não estamos classificando objetos de forma categórica, mas de forma binária.

Por fim, lembre-se que uma vez definida a fase onde existe a função .fit( ), ao selecionar e executar esse bloco de código a rede começará seu processamento, aqui, alimentada com os dados contidos em streino e streino, atualizando os valores a cada 10 rodadas por meio do parâmetro batch_size = 10 e executando o reprocessamento da rede 100 vezes de acordo com o parâmetro epochs = 100.

```
Epoch 96/100
426/426 [==============================] - 0s 106us/step - loss: 5.9503 -
binary_accuracy: 0.6268
Epoch 97/100
426/426 [==============================] - 0s 108us/step - loss: 5.9503 -
binary_accuracy: 0.6268
Epoch 98/100
426/426 [==============================] - 0s 108us/step - loss: 5.9503 -
binary_accuracy: 0.6268
Epoch 99/100
426/426 [==============================] - 0s 110us/step - loss: 5.9503 -
binary_accuracy: 0.6268
Epoch 100/100
426/426 [==============================] - 0s 108us/step - loss: 5.9503 -
binary_accuracy: 0.6268
Out[ ]:
```

Executada uma primeira vez a rede com esses parâmetros note que ela nos retorna uma taxa de precisão sobre sua classificação binária de apenas 62%, o que pode ser interpretada como uma margem de erro muito alta nesses moldes de processamento, o que precisamos fazer nesses casos são reajustes para aumentar essa taxa de precisão/acertos, em problemas reais, para fins profissionais, dependendo do contexto uma taxa de acerto por volta de 90% já é considerada baixa... 62% está muito fora desse padrão.

```
31 previsor = classificador.predict(eteste)
32
```

Criamos nosso previsor, que realiza as devidas previsões sobre eteste por meio da função .predict( ).

```
33 from sklearn.metrics import confusion_matrix, accuracy_score
34
35 margem_acertos = accuracy_score(steste, previsor)
36 matriz_confusao = confusion_matrix(steste, previsor)
37 resultadoMatrizConfusao = classificador.evaluate(eteste, steste)
```

Realizando as importações das ferramentas confusion_matrix e accuracy_score do módulo metrics da biblioteca sklearn podemos realizar estes testes complementar e verificar se existem erros ou inconsistências no processamento desses dados ou se realmente por hora a eficiência de nossa rede está muito baixa.

Criamos então uma variável de nome margem_acertos que aplica a função da ferramenta accuracy_score sobre os dados de steste e previsor. Da mesma forma criamos uma variável de nome matriz_confusão que aplica o teste homônimo também sobre os dados de steste e previsor. Por fim criamos uma variável de nome resultadoMatrizConfusão que faz o levantamento desse parâmetro de comparação sobre os dados de eteste e steste.

Nome	Tipo	Tamanho	Valor
eteste	DataFrame	(143, 30)	Column names: radius_mean, text
etreino	DataFrame	(426, 30)	Column names: radius_mean, text
margem_acertos	float64	1	0.6293706293706294
matriz_confusao	int64	(2, 2)	[[0 53] [0 90]]
previsor	float32	(143, 1)	[[1.] [1.]]
resultadoMatrizConfusao	list	2	[5.908716241796534, 0.6293706295790372]
saidas	DataFrame	(569, 1)	Column names: 0

Selecionando todo bloco de código e executando é possível verificar nas variáveis criadas que de fato se comprovou que, executando testes individuais, o padrão de 62% de taxa de acerto se manteve. Identificado que o problema é a eficiência de nossa rede neural, hora de trabalhar para corrigir isso. Mas antes, como de praxe, vamos executar a

validação cruzada sobre nossa base de dados.

```
39 from keras.wrappers.scikit_learn import KerasClassifier
40 from sklearn.model_selection import cross_val_score
```

Para isso, vamos aproveitar e já fazer as importações do KerasClassifier assim como do cross_val_score, das suas respectivas bibliotecas como mostrado acima.

```
42 def valCruzada():
43     classificadorValCruzada = Sequential()
44     classificadorValCruzada.add(Dense(units = 16,
45                                       activation = 'relu',
46                                       kernel_initializer = 'random_uniform',
47                                       input_dim = 30))
48     classificador.add(Dropout(0.2))
49     classificadorValCruzada.add(Dense(units = 16,
50                                       activation = 'relu',
51                                       kernel_initializer = 'random_uniform'))
52     classificadorValCruzada.add(Dense(units = 1,
53                                       activation = 'sigmoid'))
54     classificadorValCruzada.compile(optimizer = 'adam',
55                                     loss = 'binary_crossentropy',
56                                     metrics = ['binary_accuracy'])
57     return classificadorValCruzada
```

Dando sequência, assim como feito no exemplo anterior, vamos criar nossa validação cruzada em forma de função. Para isso definimos valCruzada( ), sem parâmetros mesmo, e dentro dela criamos aquela estrutura de rede neural densa e sequencial como estamos habituados.

Note que o único diferencial é que entre a primeira camada oculta e a cama oculta subsequente existe uma cama de Dropout, parametrizada em 0.2, na prática o que esta camada faz é, do seu número total de neurônios da camada, pegar 20% deles aleatoriamente e simplesmente os descartar de forma a não gerar impacto negativo no processamento. Inicialmente essa camada que, havia 16 neurônios, passa a ter 13, e dessa forma, somando os outros métodos de reajustes dos pesos e processamento entre camadas, se busca uma integridade de dados igual, porém com performance maior.

Raciocine que eu uma camada de 16 neurônios este impacto é mínimo, mas em uma rede que, apenas como exemplo, possua mais de 100 neurônios por camada, assim como mais de 5 camadas ocultas, o parâmetro definido como dropout para elas irá gerar grande impacto de performance. Como sempre, é interessante realizar testes com diferentes valores de

parâmetro e comparar os resultados.

```
59 classificador = KerasClassifier(build_fn = valCruzada,
60                                  epochs = 100,
61                                  batch_size = 10)
62 resultadoValCruzada = cross_val_score(estimator = classificador,
63                                        X = entradas,
64                                        y = saidas,
65                                        cv = 10,
66                                        scoring = 'accuracy')
```

Em seguida executamos nosso classificador via KerasClassifier, passando nossa função valCruzada( ) como parâmetro, atualizando os registros a cada 10 rodadas, executando a rede 100 vezes. Logo após é feita de fato a validação cruzada, apenas recordando, esta etapa pega nossa base de dados e divide em um número x de partes iguais e testa individualmente elas para que se verifique as devidas taxas de precisão e margem de erro.

```
68 mediaValCruzada = resultadoValCruzada.mean()
69 desvioValCruzada = mediaValCruzada.std()
```

Por fim, executando as linhas de código acima temos as variáveis dedicadas aos resultados das execuções de nossa validação cruzada.

Nome	Tipo	Tamanho	Valor
desvioValCruzada	float64	1	0.0
entradas	DataFrame	(569, 30)	Column names: radius_mean, texture_
eteste	DataFrame	(143, 30)	Column names: radius_mean, texture_
etreino	DataFrame	(426, 30)	Column names: radius_mean, texture_
mediaValCruzada	float64	1	0.9156015037593985
resultadoValCruzada	float64	(10,)	[0.77192982 0.94736842 0.94736842 0.8 0.94736842 ...
saidas	DataFrame	(569, 1)	Column names: 0

No explorador de variáveis podemos conferir que como resultado de nossa validação cruzada agora temos uma taxa de precisão de 91%. Essa discrepância muito provavelmente está acontecendo por algum erro de leitura dos dados, já que agora, aplicando testes sobre partes individuais o percentual não só saiu do padrão anterior, mas aumentou consideravelmente.

Realizando Tuning do Modelo

Aqui para não continuar na mesmice do exemplo anterior, vamos fazer um processo bastante comum que é o reajuste manual dos parâmetros da rede, assim como os testes para se verificar qual configuração de parâmetros retorna melhores resultados. Até então o que eu lhe sugeri foi usar os parâmetros usuais assim como suas pré-configurações, apenas alterando as taxas de reajustes dos pesos e número de execuções da rede. Porém haverão situações onde este tipo de ajuste, na camada mais superficial da rede não surtirá impacto real, sendo necessário definir e testar parâmetros internos manualmente para se buscar melhores resultados.

Consultando a documentação de nossas ferramentas podemos notar que existe uma série de parâmetros internos a nossa disposição. Como saber qual se adapta melhor a cada situação? Infelizmente em grande parte dos casos teremos de, de acordo com sua descrição na documentação, realizar testes com o mesmo inclusive de comparação com outros parâmetros da mesma categoria para encontrar o que no final das contas resultará em resultados de processamento mais íntegros.

Em algumas literaturas esse processo é chamado de tuning. Para também sair fora da rotina vamos aplicar tal processo a partir de um arquivo novo apenas com o essencial para o processo de tunagem. Outro ponto para por hora finalizar nossa introdução é que essa fase em especial costuma ser a mais demorada, uma vez que de acordo com o número de parâmetros a serem testados, cada teste é uma nova execução da rede neural.

Breast Cancer Dataset Tuning.py

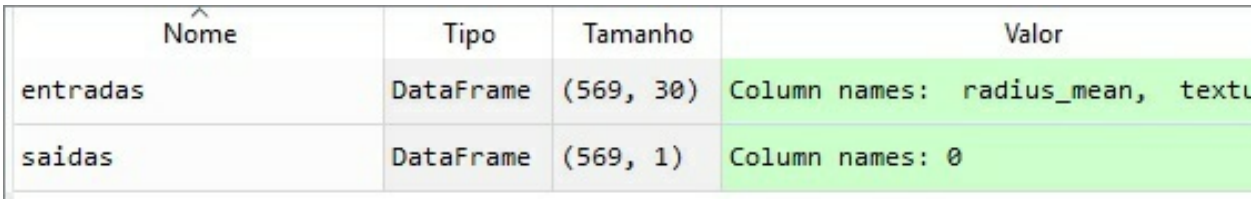
```
1 import pandas as pd
2 import keras
3 from keras.models import Sequential
4 from keras.layers import Dense, Dropout
5 from keras.wrappers.scikit_learn import KerasClassifier
6 from sklearn.model_selection import GridSearchCV
```

Seguindo a mesma lógica de sempre, toda vez que criamos um novo arquivo de código, primeira etapa é realizar as devidas importações das bibliotecas e módulos necessários. Repare que na última linha existe uma ferramenta nova sendo importada, trata-se da GridSearchCV, do módulo model_selection da biblioteca sklearn.

Em suma essa ferramenta permite de fato criarmos parâmetros manualmente para alimentar nossa rede neural, assim como passar mais de um parâmetro a ser testado para por fim serem escolhidos os que em conjunto mostraram melhores resultados de processamento.

```
8 entradas = pd.read_csv('entradas-breast.csv')
9 saidas = pd.read_csv('saidas-breast.csv')
```

Em seguida por meio da função .read_csv( ) da biblioteca pandas importamos novamente nossas bases de dados de entrada e saída.



Nome	Tipo	Tamanho	Valor
entradas	DataFrame	(569, 30)	Column names: radius_mean, textu
saidas	DataFrame	(569, 1)	Column names: 0

S e o processo ocorreu sem erros, as devidas variáveis foram criadas.

```
11 def tuningClassificador(optimizer, loss, kernel_initializer, activation, neuro
12     classificadorTuning = Sequential()
13     classificadorTuning.add(Dense(units = neurons,
14                                   activation = activation,
15                                   kernel_initializer = kernel_initializer,
16                                   input_dim = 30))
17     classificadorTuning.add(Dropout(0.2))
18     classificadorTuning.add(Dense(units = neurons,
19                                   activation = activation,
20                                   kernel_initializer = kernel_initializer))
21     classificadorTuning.add(Dropout(0.2))
22     classificadorTuning.add(Dense(units = 1,
23                                   activation = 'sigmoid'))
24     classificadorTuning.compile(optimizer = optimizer,
25                                 loss = loss,
26                                 metrics = ['binary_accuracy'])
27     return classificadorTuning
```

Logo após já podemos criar nossa rede, dentro de uma função criada especificamente para ela de nome tuningClassificador( ), repare que até então não havíamos passado nenhum tipo de parâmetro em nossas funções, apenas seu bloco de código interno, dessa vez, como iremos definir os parâmetros manualmente, instanciados por uma variável, é interessante declara-los já como parâmetro da função.

Em seguida criamos a mesma estrutura da rede neural anterior, sequencial, 30 neurônios na camada de entrada, 1 na camada de saída e dropouts entre elas, porém note que os parâmetros internos a cada camada

agora possuem nomes genéricos para substituição.

Por fim na camada de compilação também é mantido a métrica binary_accuracy uma vez que esta é uma classificação binária. Encerrando o bloco, apenas é dado o comando para retornar a própria função.

```
29 classificadorTunado = KerasClassifier(build_fn = tuningClassificador)
30
```

A seguir criamos uma variável de nome classificadorTunado, que recebe como atributo o KerasClassifier, que por sua vez tem em sua ativação nossa função tuningClassificador( ).

```
31 parametros = {'batch_size':[10,30],
32               'epochs':[50,100],
33               'optimizer':['adam','sgd'],
34               'loss':['binary_crossentropy','hinge'],
35               'kernel_initializer':['random_uniform','normal'],
36               'activation':['relu','tanh'],
37               'neurons':[10,8]}
```

Da mesma forma criamos uma nova variável, agora de nome parâmetros, que pela sintaxe recebe em forma de dicionário os parâmetros a serem instanciados em nossa rede assim como os valores a serem testados.

Repare por exemplo que ‘activation’ quando instanciada, usará ‘relu’ em uma das suas fases de testes assim como ‘tanh’ em outra fase de teste, posteriormente a GridSearchCV irá verificar qual desses dois parâmetros retornou melhor resultado e irá nos sinalizar este.

```
39 tunagem = GridSearchCV(estimator = classificadorTunado,
40                         param_grid = parametros,
41                         scoring = 'accuracy')
```

Logo após criamos uma variável de nome tunagem, que recebe como atributo a ferramenta GridSearchCV, que por sua vez tem como parâmetros estimator = classificadorTunado, o que em outras palavras significa que ele usará o KerasClassifier rodando nossa função tuningClassificador( ) sob os parâmetros, também tem como parâmetro param_grid = parametros, o que significa que o classificador irá receber os parâmetros que definimos manualmente na variável parametros, por fim scoring = ‘accuracy’ que simplesmente estará aplicando métodos focados em precisão.

```
43 tunagem = tunagem.fit(entradas,saidas)
44
```

Por fim, criamos uma variável de nome tunagem, que aplica sobre si a função .fit( ) passando para a mesma como parâmetros entradas e saidas. Como das outras vezes, ao selecionar essa linha de código e executar, a função .fit( ) dará início a execução da rede neural.

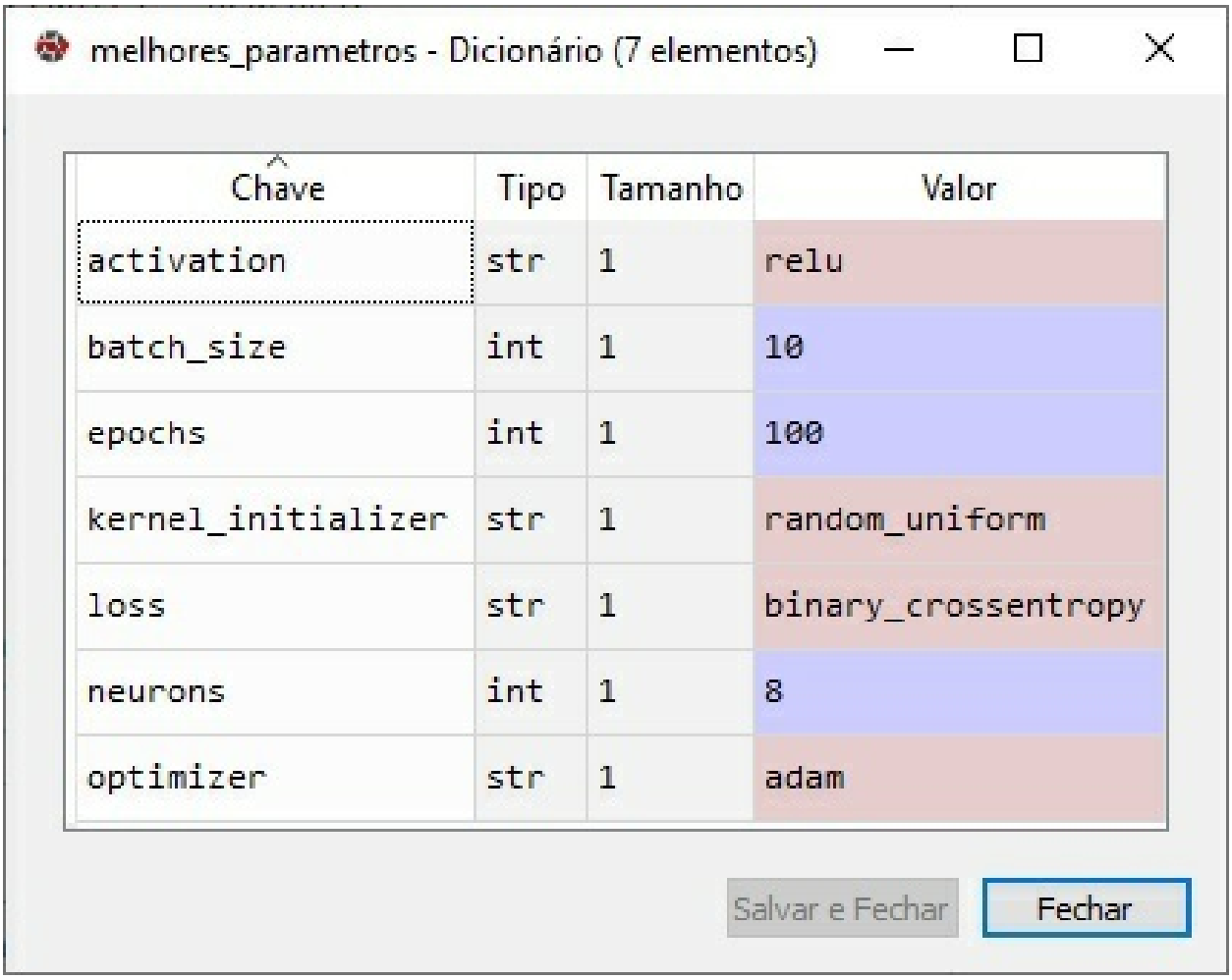
Como dito anteriormente, este processo de testar todos os parâmetros buscando os melhores é bastante demorado, dependendo muito do hardware de sua máquina, dependendo de sua configuração, esse processo pode literalmente demorar algumas horas.

```
binary_accuracy: 0.9686
Epoch 96/100
569/569 [==============================] - 1s 2ms/step - loss: 0.2431 -
binary_accuracy: 0.9663
Epoch 97/100
569/569 [==============================] - 1s 2ms/step - loss: 0.2124 -
binary_accuracy: 0.9504
Epoch 98/100
569/569 [==============================] - 1s 2ms/step - loss: 0.2342 -
binary_accuracy: 0.9663
Epoch 99/100
569/569 [==============================] - 1s 2ms/step - loss: 0.2219 -
binary_accuracy: 0.9651
Epoch 100/100
569/569 [==============================] - 1s 2ms/step - loss: 0.2393 -
binary_accuracy: 0.9716
```

Acompanhando o processo via console você de fato irá notar que essa rede será processada e reprocessada muitas vezes, cada vez voltando a estaca zero e repetindo todo o processo com os parâmetros definidos anteriormente.

```
45 melhores_parametros = tunagem.best_params_
46 melhor_margem_precisao = tunagem.best_score_
```

Para finalizar o processo de tuning, podemos como de costume usar funções que nos mostrem de forma mais clara os resultados. Inicialmente criamos uma variável de nome melhores_parametros que por fim aplica sobre tunagem o método best_params_, por fim criamos uma variável de nome melhor_margem_precisao que sobre tunagem aplica o método best_score_. Selecionando e executando esse bloco de código temos acesso a tais dados.



Via explorador de variáveis você pode verificar quais os parâmetros que foram selecionados quanto a sua margem de precisão. Uma vez que você descobriu quais são os melhores parâmetros você pode voltar ao seu código original, fazer as devidas substituições e rodar novamente sua rede neural artificial.

Nome	Tipo	Tamanho	Valor
entradas	DataFrame	(569, 30)	Column names: radius_mean, textu
melhor_margem_precisao	float64	1	0.9780667838312829
melhores_parametros	dict	7	{'activation':'relu', 'batch_size' 'epochs':100, 'kernel_initializ ..
parametros	dict	7	{'batch_size':[10, 30], 'epochs':[100], 'optimizer':['adam', 'sgd' .
saidas	DataFrame	(569, 1)	Column names: 0

Por fim também está disponível para visualização a variável

dedicada a retornar o percentual de acerto de nosso processo de tuning. Assim como o valor aproximado exibido no console no processo de testes, agora podemos ver de forma clara que usando dos melhores parâmetros essa rede chega a uma margem de precisão de 97% em sua classificação binária.

Código Completo:

```
import pandas as pd
import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.wrappers.scikit_learn import KerasClassifier
from sklearn.model_selection import GridSearchCV

entradas = pd.read_csv('entradas-breast.csv')
saidas = pd.read_csv('saidas-breast.csv')

def tuningClassificador(optimizer, loss, kernel_initializer, activation, neuro
    classificadorTuning = Sequential()
    classificadorTuning.add(Dense(units = neurons,
                                  activation = activation,
                                  kernel_initializer = kernel_initializer,
                                  input_dim = 30))
    classificadorTuning.add(Dropout(0.2))
    classificadorTuning.add(Dense(units = neurons,
                                  activation = activation,
                                  kernel_initializer = kernel_initializer))
    classificadorTuning.add(Dropout(0.2))
    classificadorTuning.add(Dense(units = 1,
                                  activation = 'sigmoid'))
    classificadorTuning.compile(optimizer = optimizer,
                                loss = loss,
                                metrics = ['binary_accuracy'])
    return classificadorTuning
```

```
29 classificadorTunado = KerasClassifier(build_fn = tuningClassificador)
30 parametros = {'batch_size':[10,30],
31               'epochs':[50,100],
32               'optimizer':['adam','sgd'],
33               'loss':['binary_crossentropy','hinge'],
34               'kernel_initializer':['random_uniform','normal'],
35               'activation':['relu','tanh'],
36               'neurons':[10,8]}
37 tunagem = GridSearchCV(estimator = classificadorTunado,
38                        param_grid = parametros,
39                        scoring = 'accuracy')
40 tunagem = tunagem.fit(entradas,saidas)
41 melhores_parametros = tunagem.best_params_
42 melhor_margem_precisao = tunagem.best_score_
```

Realizando Testes Sobre Uma Amostra

Uma das propriedades importantes de uma rede neural após construída, treinada e testada é que a partir deste ponto podemos salvar esse modelo para reutilização. Uma das práticas mais comuns em uma rede como esta, de classificação binária, é a partir do modelo pronto e treinado reutilizar a mesma para que se teste alguma amostra individualmente.

Seguindo a linha de raciocínio deste exemplo, lembre-se que aqui estamos passando uma série de atributos para que seja verificado se naquela imagem de mamografia o resultado é positivo para câncer ou negativo para câncer. Dessa forma, podemos pegar uma amostra individual de nossa base de dados e aplicar o teste da rede neural sobre a mesma.

Breast Cancer Dataset Teste Uma Amostra.py

```
1 import pandas as pd
2 import numpy as np
3 from keras.models import Sequential
4 from keras.layers import Dense, Dropout
```

A partir de um novo arquivo iniciamos como sempre com as importações necessárias.

```
6 entradas = pd.read_csv('entradas-breast.csv')
7 saidas = pd.read_csv('saidas-breast.csv')
```

Realizamos a importação de nossa base de dados, dividida em dados de entrada e de saída a partir dos seus respectivos arquivos do tipo .csv.

Nome	Tipo	Tamanho	Valor
entradas	DataFrame	(569, 30)	Column names: radius_mean, texture_mean, perimeter_mean, area_mean ...
saidas	DataFrame	(569, 1)	Column names: 0

Como de praxe realizamos a verificação se não houve nenhum erro no processo tanto no processo de importação quanto de criação das respectivas variáveis.

```
classificadorA = Sequential()
classificadorA.add(Dense(units = 8,
                         activation = 'relu',
                         kernel_initializer = 'normal',
                         input_dim = 30))
classificadorA.add(Dropout(0.2))
classificadorA.add(Dense(units = 8,
                         activation = 'relu',
                         kernel_initializer = 'normal',))
classificadorA.add(Dropout(0.2))
classificadorA.add(Dense(units = 1,
                         activation = 'sigmoid'))
classificadorA.compile(optimizer = 'adam',
                       loss = 'binary_crossentropy',
                       metrics = ['binary_accuracy'])
classificadorA.fit(entradas,
                   saidas,
                   batch_size = 10,
                   epochs = 100)
```

Criamos como usualmente fizemos anteriormente a estrutura de nossa rede neural, agora já parametrizada com os melhores parâmetros encontrados no processo de tuning.

```
Epoch 96/100
569/569 [==============================] - 0s 128us/step - loss: 0.3394 -
binary_accuracy: 0.8541
Epoch 97/100
569/569 [==============================] - 0s 126us/step - loss: 0.3236 -
binary_accuracy: 0.8594
Epoch 98/100
569/569 [==============================] - 0s 130us/step - loss: 0.3346 -
binary_accuracy: 0.8401
Epoch 99/100
569/569 [==============================] - 0s 135us/step - loss: 0.3218 -
binary_accuracy: 0.8699
Epoch 100/100
569/569 [==============================] - 0s 125us/step - loss: 0.3339 -
binary_accuracy: 0.8506
Out[2]: <keras.callbacks.History at 0x1bedba79cf8>
```

Selecionando e executando todo bloco de código de nossa rede neural acompanhamos via console o processo de execução da mesma assim como se houve algum erro no processo. Lembrando que a esta altura o mais comum é se houver erros, estes ser erros de sintaxe, pois da estrutura lógica da rede até aqui você já deve compreender.

```
objeto = np.array([[17.99,10.38,122.8,1001,0.1184,0.2776,0.3001,
                    0.1471,0.2419,0.07871,1095,0.9053,8589,153.4,
                    0.006399,0.04904,0.05373,0.01587,0.03003,0.006193,
                    25.38,17.33,184.6,2019,0.1622,0.6656,0.7119,0.2654,
                    0.4601,0.1189]])
```

Dando sequência, vamos ao que realmente interessa. Uma vez feito todo processo de importação das bibliotecas, módulos, base de dados e executada a rede neural uma vez para seu respectivo treino, hora de aplicar teste sobre uma amostra. Para isso, simplesmente criamos uma variável de nome objeto que recebe como atributo uma array do tipo numpy onde selecionamos, aqui nesse exemplo, uma linha aleatória em nossa base de dados com seus respectivos 30 atributos previsores.

```
previsorA = classificadorA.predict(objeto)
previsorB = (previsorA > 0.5)
```

Por fim criamos duas variáveis, uma de nome previsorA que recebe como atributo a função .predict( ) parametrizada com nosso objeto criado anteriormente, executando tudo sobre nossa rede neural. Da mesma forma, criamos uma variável de nome previsorB que simplesmente irá conferir se previsorA for maior do que 0.5, automaticamente essa expressão nos retornará um dado True ou False.

Nome	Tipo	Tamanho	Valor
entradas	DataFrame	(569, 30)	Column names: radius_mean, texture_mean, perimeter_mean, area_mean ...
objeto	float64	(1, 30)	[[1.799e+01 1.038e+01 1.228e+02 ... 2.654e-01 4.601e-01 1.189e-01]]
previsorA	float32	(1, 1)	[[3.970654e-06]]
previsorB	bool	(1, 1)	[[False]]
saidas	DataFrame	(569, 1)	Column names: 0

De fato, executando todo último bloco de código são criadas as respectivas variáveis. A variável previsorA nos deu como retorno um número extremamente pequeno, porém, como mencionado anteriormente, o ideal é expormos nossas informações de forma clara.

O previsorB por sua vez, com base em previsorA nos retorna False, que em outras palavras significa que para aquela amostra o resultado é negativo para câncer.

Salvando o Modelo Para Reutilização

Como dito anteriormente, uma prática comum é a reutilização de um modelo de rede uma vez que esse está pronto, sem erros de código, treinado já com os melhores parâmetros, para que ele possa ser ferramenta para solucionar certos problemas computacionais de mesmo tipo. Isto pode ser feito de forma bastante simples, salvando o código da rede neural (obviamente) assim como sua estrutura funcional e pesos por meio de arquivos do tipo .py, .json e .h5. Raciocine que em bases de dados pequenas, como esta que estamos usando para testes, o processamento dela pela rede se dá em poucos segundos, porém, em determinadas situações reais é interessante você ter uma rede pronta sem a necessidade de ser treinada novamente (ainda mais com casos onde são usadas bases com milhões de dados como em big data).

```
import pandas as pd
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, Dropout

entradas = pd.read_csv('entradas-breast.csv')
saidas = pd.read_csv('saidas-breast.csv')

classificadorB = Sequential()
classificadorB.add(Dense(units = 8,
                         activation = 'relu',
                         kernel_initializer = 'normal',
                         input_dim = 30))
classificadorB.add(Dropout(0.2))
classificadorB.add(Dense(units = 8,
                         activation = 'relu',
                         kernel_initializer = 'normal',))
classificadorB.add(Dropout(0.2))
classificadorB.add(Dense(units = 1,
                         activation = 'sigmoid'))
classificadorB.compile(optimizer = 'adam',
                       loss = 'binary_crossentropy',
                       metrics = ['binary_accuracy'])
classificadorB.fit(entradas,
                   saidas,
                   batch_size = 10,
                   epochs = 100)
```

Repare que esta é simplesmente a mesma estrutura criada anteriormente, já com as devidas importações, já com toda estrutura da rede neural, etc... Simplesmente você pode reaproveitar todo esse bloco de código em um novo arquivo para poupar seu tempo.

```
classificador_json = classificadorB.to_json()

with open('classificador_binario.json', 'w') as json_file:
    json_file.write(classificador_json)

classificadorB.save_weights('classificador_binario_pesos.h5')
```

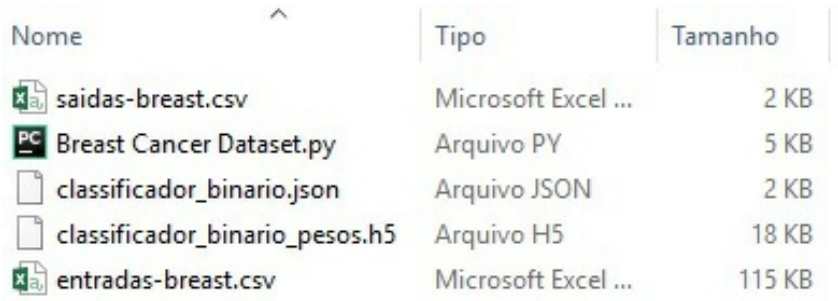
O processo em si, de salvar nosso modelo para reutilização basicamente de dá da seguinte forma. Inicialmente criamos uma variável de nome classificador_json que recebe toda estrutura de nossa rede neural classificadorB com a função .to_json( ), essa função permite salvar esse modelo neste formado de arquivo nativo para as IDEs Python.

Em seguida entramos com o método with open( ) que por sua vez recebe como parâmetro inicialmente ‘classificador_binario.json’, ‘w’, isto

nada mais é do que o comando que estamos dando para que se crie um arquivo com o nome classificador_binario.json na mesma pasta onde estão seus arquivos de código, na sequência existe a condição as json_file: que permite definirmos algum atributo interno.

Dentro de json_file simplesmente aplicamos sobre si a função .write( ) passando como parâmetro todo o conteúdo de classificador_json.

Por fim, fazemos um processo parecido para salvar os pesos, mas agora, usando uma função interna da biblioteca pandas mesmo, então, alocando sobre classificadorB a função .save_weights( ) e definindo como parâmetro também o nome do arquivo que guardará esses pesos, no formato h5. Selecionando e rodando esse bloco de código serão criados os respectivos arquivos na pasta onde está o seu código.



Carregando Uma Rede Neural Artificial Pronta Para Uso

```
1 import numpy as np
2 from keras.models import model_from_json
```

Como sempre, começando pelas importações, note que há a importação de uma ferramenta ainda não mencionada nos exemplos anteriores. A ferramenta model_from_json, como o próprio nome indica, serve para importar modelos prontos, desde que estejam no formado json, esta ferramenta faz parte do módulo models da biblioteca keras.

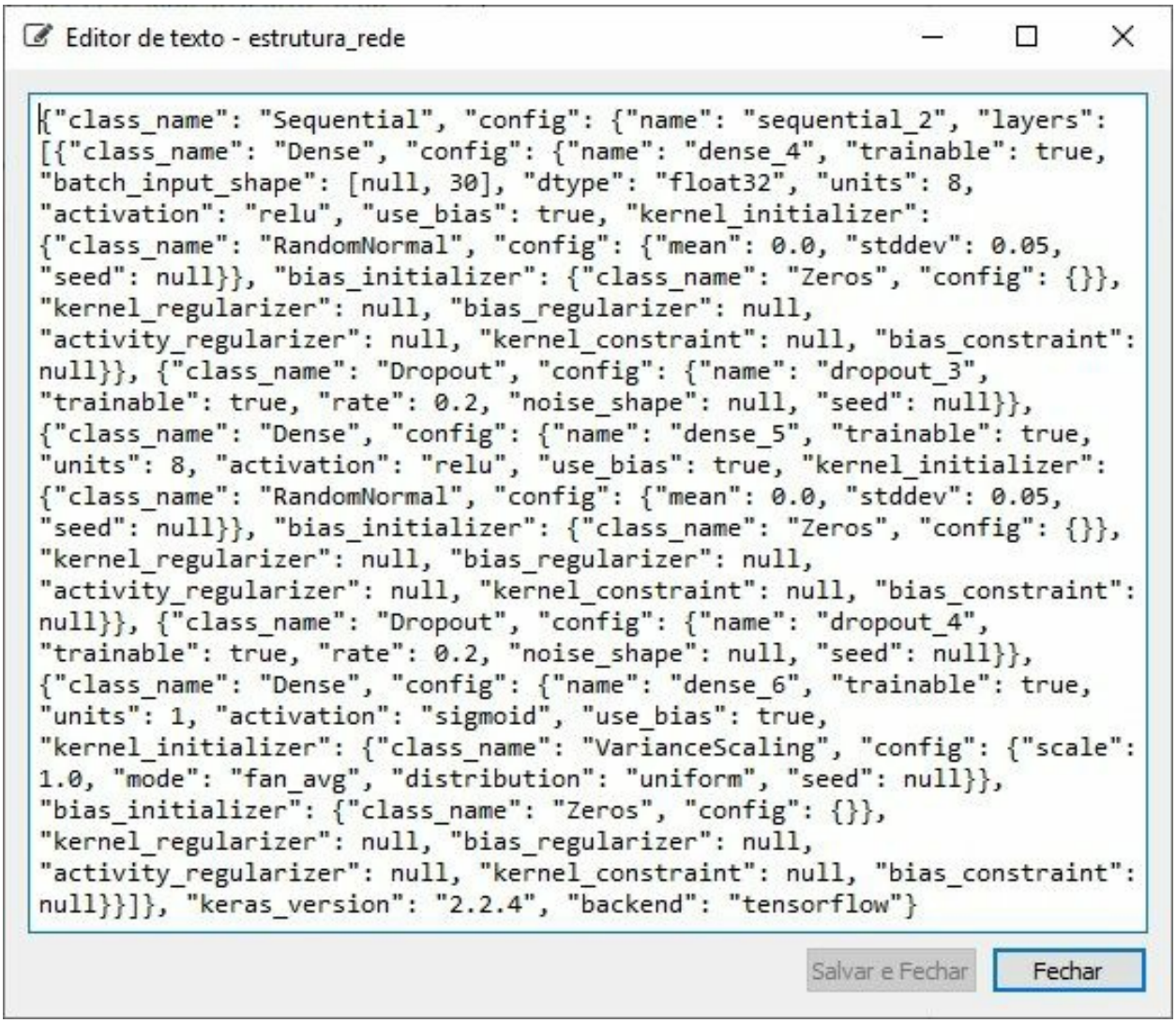
```
4 arquivo = open('classificador_binario.json', 'r')
5 estrutura_rede = arquivo.read()
6 arquivo.close()
```

Em seguida criamos uma variável de nome arquivo, que recebe como atributo ao método open, parametrizado com o nome de nosso arquivo salvo anteriormente em forma de string, assim como o segundo parâmetro ‘r’, uma vez que agora estamos fazendo a leitura do arquivo.

Na sequência criamos uma variável de nome estrutura_rede que recebe os dados de arquivo pela função .read( ). Por fim, podemos fechar o arquivo carregado por meio da função .close( ). Pode ficar tranquilo que uma vez feita a leitura ela ficará pré-alocada em memória para nosso interpretador, o arquivo em si pode ser fechado normalmente.

Nome	Tipo	Tamanho	Valor
estrutura_rede	str	1	{"class_name": "Sequential", "config": {"name": "sequential_2", "layer ...

Se o processo de leitura a partir do arquivo json foi feita corretamente, a respectiva variável será criada.



Abrindo a variável estrutura_rede é possível ver que de fato todos os parâmetros de nossa rede neural estão salvos de forma que nosso

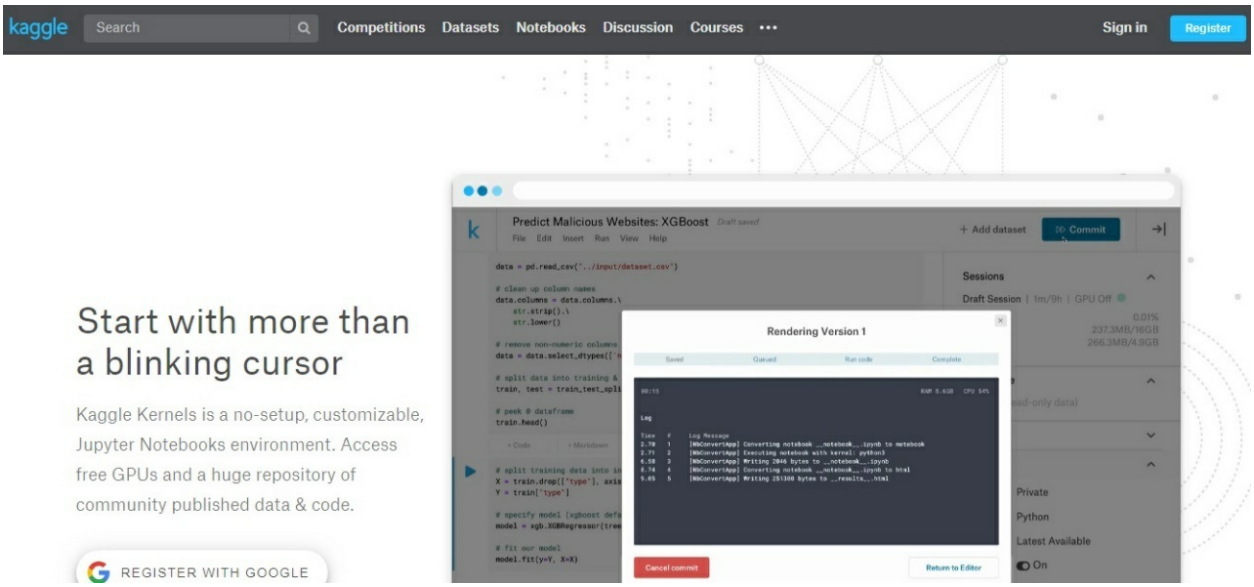
interpretador conseguirá fazer a leitura e uso dos mesmos. A partir daqui, é possível criar qualquer modelo previsor, instanciando nosso classificador.

Regressão de Dados de Planilhas Excel – Autos Dataset

Uma vez entendidos os conceitos básicos do que é uma rede neural artificial e sua estrutura lógica e de código quando o problema computacional proposto é uma classificação de dados, hora de gradualmente avançarmos para outros modelos, não necessariamente de maior complexidade, mas modelos que elucidarão as propostas de outros tipos de redes neurais desenvolvidas para outros fins.

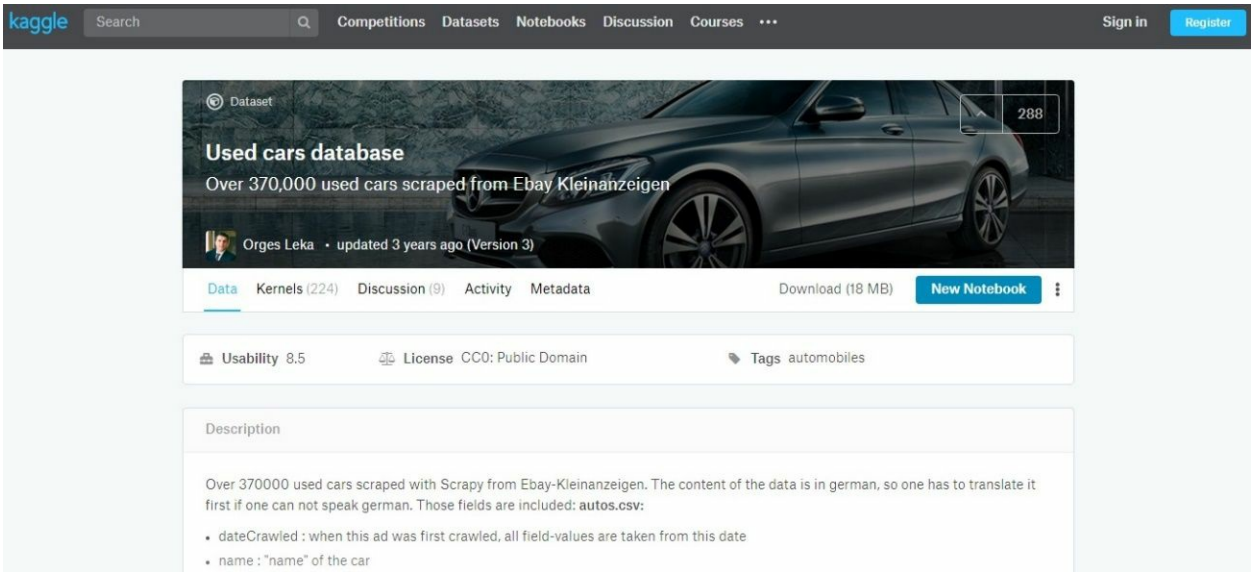
Um dos tipos de redes neurais artificiais mais comumente utilizados são as redes que aplicam métodos de regressão. Em outras palavras, com base em uma grande base de dados, treinaremos uma rede para reconhecer padrões que permitirão encontrar por exemplo, o valor médio de um automóvel usado se comparado a outros de mesmo tipo.

Aqui como exemplo usaremos o Autos Dataset, uma base de dados com milhares de carros usados catalogados como em uma revenda. Aproveitaremos esse modelo para ver como se dá o processo de polimento dos dados para criarmos uma base com valores mais íntegros, capazes de gerar resultados mais precisos.

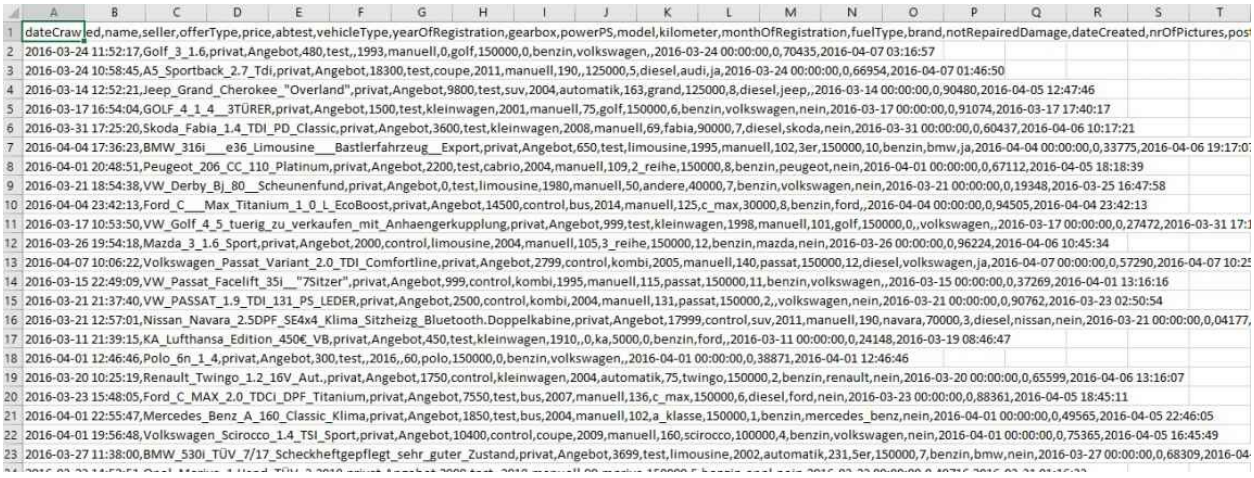


A base de dados que iremos usar neste exemplo pode ser baixada gratuitamente no Kaggle, um repositório de machine learning parecido com o

ACI, porém mais diversificado e com suporte de toda uma comunidade de desenvolvedores.



Usando a ferramenta de busca do site você pode procurar por Used cars database. O dataset em si, como exibido na imagem acima, trata-se de uma base de dados de aproximadamente 370 mil registros de anúncios de carros usados retirado dos classificados do Ebay. Tudo o que precisamos fazer é o download da base de dados no formato .csv.



Abrindo o arquivo pelo próprio Excel pode-se ver que é uma daquelas típicas bases de dados encontradas na internet, no sentido de haver muita informação faltando, muitos dados inconsistentes, muito a ser pré-processado antes de realmente começar a trabalhar em cima dessa base.

Autos Dataset.py

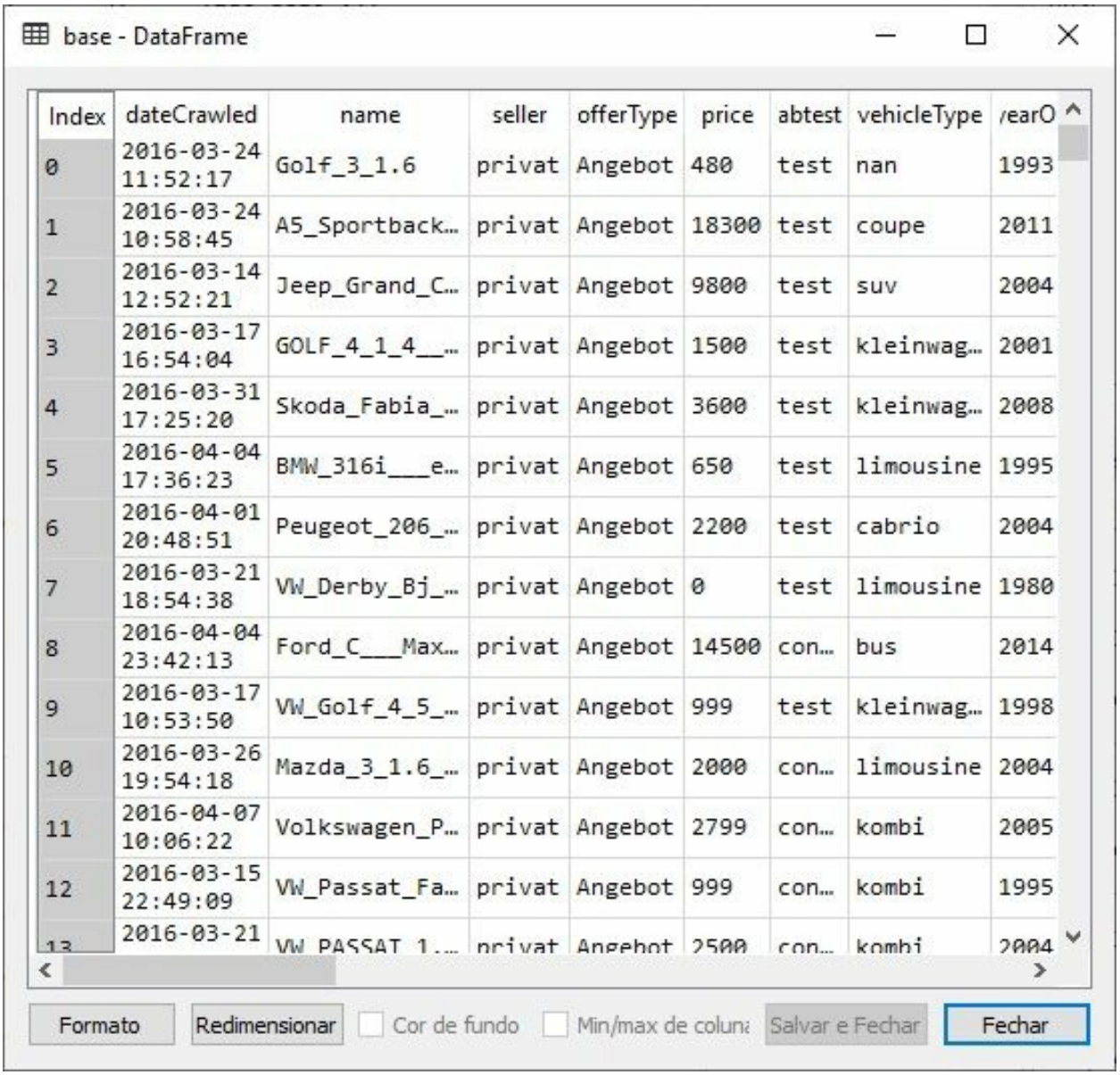
```
1 import pandas as pd
2
3 base = pd.read_csv('autos.csv', encoding = 'ISO-8859-1')
```

Dando início ao que interessa, como sempre, todo código começa com as devidas importações das bibliotecas e módulos necessários, por hora, simplesmente importamos a biblioteca pandas para que possamos trabalhar com arrays.

Em seguida já criamos uma variável de nome base que pela função .read_csv( ) realiza a leitura e importação dos dados de nosso arquivo autos.csv, note que aqui há um parâmetro adicional encoding = ‘ISO-8859-1’, este parâmetro nesse caso se faz necessário em função de ser uma base de dados codificada em alguns padrões europeus, para não haver nenhum erro de leitura é necessário especificar a codificação do arquivo para o interpretador.

Nome	Tipo	Tamanho	Valor
base	DataFrame	(371528, 20)	Column names: dateCrawled, name, seller, offerTyp price, abtest, veh ...

Feita a importação da base de dados, podemos conferir em nosso explorador de variáveis que o objeto foi importado e reconhecido como um DataFrame, contendo 371528 amostras com 20 atributos, respectivamente.



Index	dateCrawled	name	seller	offerType	price	abtest	vehicleType	yearO
0	2016-03-24 11:52:17	Golf_3_1.6	privat	Angebot	480	test	nan	1993
1	2016-03-24 10:58:45	A5_Sportback…	privat	Angebot	18300	test	coupe	2011
2	2016-03-14 12:52:21	Jeep_Grand_C…	privat	Angebot	9800	test	suv	2004
3	2016-03-17 16:54:04	GOLF_4_1_4__…	privat	Angebot	1500	test	kleinwag…	2001
4	2016-03-31 17:25:20	Skoda_Fabia_…	privat	Angebot	3600	test	kleinwag…	2008
5	2016-04-04 17:36:23	BMW_316i___e…	privat	Angebot	650	test	limousine	1995
6	2016-04-01 20:48:51	Peugeot_206_…	privat	Angebot	2200	test	cabrio	2004
7	2016-03-21 18:54:38	VW_Derby_Bj_…	privat	Angebot	0	test	limousine	1980
8	2016-04-04 23:42:13	Ford_C___Max…	privat	Angebot	14500	con…	bus	2014
9	2016-03-17 10:53:50	VW_Golf_4_5_…	privat	Angebot	999	test	kleinwag…	1998
10	2016-03-26 19:54:18	Mazda_3_1.6_…	privat	Angebot	2000	con…	limousine	2004
11	2016-04-07 10:06:22	Volkswagen_P…	privat	Angebot	2799	con…	kombi	2005
12	2016-03-15 22:49:09	VW_Passat_Fa…	privat	Angebot	999	con…	kombi	1995
13	2016-03-21	VW_PASSAT_1…	privat	Angebot	2500	con…	kombi	2004

Abrindo a variável para uma rápida análise visual é possível ver que existem muitos dados faltando, ou inconsistentes, que literalmente só atrapalharão nosso processamento. Sendo assim, nessa fase inicial de polimento desses dados vamos remover todos os atributos (todas as colunas) de dados que não interessam à rede.

```
5 base = base.drop('dateCrawled', axis=1)
6
```

Aplicando diretamente sobre nossa variável base a função .drop( ) passando como parâmetro o nome da coluna e seu eixo, podemos eliminá-la sem maiores complicações. Lembrando apenas que o parâmetro axis quando

definido com valor 0 irá aplicar a função sobre uma linha, enquanto valor 1 aplicará a função sobre uma coluna.

Nome	Tipo	Tamanho	Valor
base	DataFrame	(371528, 20)	Column names: dateCrawled, name, seller, offerTyp price, abtest, veh ...

Base original.

Nome	Tipo	Tamanho	Valor
base	DataFrame	(371528, 20)	Column names: dateCrawled, name, seller, offerTyp price, abtest, veh ...

Após execução da linha de código anterior. Note que eliminamos a coluna ‘dateCrawled’, e como aplicamos a função sobre nossa variável, eliminamos essa coluna de forma permanente.

```
5 base = base.drop('dateCrawled', axis=1)
6 base = base.drop('dateCreated', axis=1)
7 base = base.drop('nrOfPictures', axis=1)
8 base = base.drop('postalCode', axis=1)
9 base = base.drop('lastSeen', axis=1)
```

Sendo assim, por meio da função .drop( ) podemos remover de imediato as colunas que não fazem sentido manter em nossa base de dados, em seguida, podemos verificar se uma ou mais colunas pode ser eliminada em relação a inconsistência de seus dados.

```
11 base['name'].value_counts()
12
```

Para tal feito selecionamos manualmente o nome de uma coluna seguido da função .value_counts( ) que irá apresentar esses dados agrupados quanto a igualdade.

```
In [37]: base['name'].value_counts()
Out[37]:
Ford_Fiesta                                                    657
BMW_318i                                                       627
Opel_Corsa                                                     622
Volkswagen_Golf_1.4                                            603
BMW_316i                                                       523
BMW_320i                                                       492
Volkswagen_Polo                                                475
Renault_Twingo                                                 447
Volkswagen_Golf                                                428
Volkswagen_Golf_1.6                                            413
Volkswagen_Polo_1.2                                            412
BMW_116i                                                       394
Opel_Corsa_1.2_16V                                             373
Opel_Corsa_B                                                   369
Opel_Astra                                                     366
```

Acompanhando via console, podemos notar que, por exemplo Ford_Fiesta aparece 657 vezes em nossa base de dados, o problema é que o interpretador está fazendo esse levantamento de forma literal, existem 657 “Ford_Fiesta” assim como devem existir outras dezenas ou centenas de “Ford Fiesta”.

Esse tipo de inconsistência é muito comum ocorrer quando importamos de plataformas onde é o usuário que alimenta com os dados, em função de cada usuário cadastrar seu objeto/item sem uma padronização de sintaxe.

```
Audi_A4_b8_8K_Avant_s_line_Leder_/_Navi_/_reduziert!    1
Ford_Mondeo_2.0_Kombi_Ghia_*Xenon_Klima*                1
Grand_Cherokee_2.7_CRD_GÜRNE_4_PM1!_2017/05TÜV          1
Volvo_V40___Tuev_neu                                    1
Mazda_121__Mit_TÜV                                      1
Seat_Altea_1.4_TSI_XL_Reference_Comfort                 1
Schicker_Gelaendewagen                                  1
Verkaufe_Audi_a4_b5_Facelift                            1
Audi__TT_Tuev_Neu_Service_Neu!!!!                       1
Auto__zu_verkaufen                                      1
Ford_Focus_2.0_TDCi_DPF_Style__Navi                     1
Opel_Rekord_Olympia_P2_Coupe_Oldtimer_/_Echter_Scheunenfund    1
Chrysler_Status_2_0_lx_Bastler_Schlachten_Leder         1
BMW_320i__MP3__PDC__Regensensor__SCHECKHEFT             1
Mercedes_Benz_A_140__Facelift___Md_04__nur_100tkm__TÜV__2.Hd.!    1
Mercedes_C_Coupe_51tkm_aus_1ter_Hand                    1
Name: name, Length: 233531, dtype: int64

In [38]:
```

Da mesma forma repare, por exemplo, que existe apenas uma amostra de Audi__TT_Tuev_Neu_Service_Neu!!!!, isso se dá pela forma da escrita do nome do objeto, de fato, ninguém mais iria anunciar seu Audi TT escrevendo aquele nome exatamente daquela forma. Sendo assim, a coluna ‘names’ pode ser eliminada em função de sua inconsistência.

```
13 base = base.drop('name', axis=1)
14 base = base.drop('seller', axis=1)
15 base = base.drop('offerType', axis=1)
```

Nesta etapa, eliminamos três colunas em função de suas inconsistências em seus dados.

```
17 varTeste1 = base.loc[base.price <= 10]
18
```

Outro processo comumente realizado é tentarmos filtrar e eliminar de nossa base de dados amostras/registros com valores absurdos. Aqui como exemplo inicial criamos uma variável de nome varTeste1 que recebe como atributo base com o método .loc onde queremos descobrir de nossa base, no atributo price, todas amostras que tem valor igual ou menor que 10.

Lembre-se que estamos trabalhando em uma base de dados de carros usados, é simplesmente impossível haver carros com preço igual ou menor a 10 Euros, com certeza isso é inconsistência de dados também.

Nome	Tipo	Tamanho	Valor
base	DataFrame	(371528, 15)	Column names: name, seller, offerType, price abtest, vehicleType, yea ...
varTeste1	DataFrame	(12118, 15)	Column names: name, seller, offerType, price abtest, vehicleType, yea ...

Selecionando a linha de código anterior e a executando note que é criada a variável varTeste1, repare que existem 12118 amostras com valor igual ou menor que 10, tantas amostras com valores tão errados certamente iriam atrapalhar o processamento de nossa rede neural posteriormente.

```
19 base = base[base.price > 10]
20
```

Aplicando este método base.price > 10 diretamente em nossa base, fazemos com que a mesma seja atualizada mantendo apenas os valores de registros com o atributo ‘price’ maior que 10.

```
19 varTeste2 = base.loc[base.price > 350000]
20 base = base[base.price < 350000]
```

Da mesma forma, filtramos e eliminamos de nossa base de dados todos registros com respectivo preço maior que 350000. Totalizando 115 amostras eliminadas da base.

```
22 base.loc[pd.isnull(base['vehicleType'])]
23 base['vehicleType'].value_counts()
```

Para finalizar nossa fase de polimento dos dados, outro processo bastante comum é corrigir os valores que estiverem literalmente faltando em nossa base de dados. Por meio do método .loc e da função .isnull( ) sobre nossa base, passando como parâmetro ‘vehicleType’, estaremos fazendo este tipo de verificação. Assim como visto anteriormente que pela função .value_counts( ) iremos ver esses dados agrupados quanto a sua nomenclatura.

```
In [  ]: base.loc[pd.isnull(base['vehicleType'])]
Out[  ]:
                                                  name  . notRepairedDamage
0                                           Golf_3_1.6  .               NaN
16                                          Polo_6n_1_4  .               NaN
22                          Opel_Meriva_1.Hand_TÜV_3.2018  .              nein
26                              Citroen_C4_Grand_Picasso.  .              nein
31                           Renault_clio_1.2_TÜV_07/2016  .               NaN
35                                             VW_Golf_3  .              nein
37                              Renault_Kangoo_1.9_Diesel  .              nein
48                VW_Golf_6___Klima___Alu___Scheckheft_!!!  .               NaN
51                             Fiat_punto_5_tuerer__6_gang  .                ja
52           Verkaufe_meinen_kleinen_wegen_neu_Anschaffung  .               NaN
58              Seat_inca_1.9SDI___LKW_Zulassung___TÜV_NEU  .              nein
66                    Opel_Astra_1.4_mit_vielen_Extras!!!!  .              nein
72                                          BMW_520i_E39  .              nein
78                             Golf_4_TDi_1_9.__3_tuerig  .               NaN
81                                   Opel_Astra_F_Cabrio  .                ja
```

Via console podemos acompanhar o retorno da função .isnull( ). Muito cuidado para não confundir a nomenclatura, neste caso, ja e nein significam sim e não, respectivamente, enquanto NaN é uma abreviação gerada pelo nosso interpretador quando realmente nesta posição da lista/dicionário existem dados faltando.

Em outras palavras, ja e nein são dados normais, presentes na classificação, NaN nos mostra que ali não há dado nenhum.

```
In [  ]: base['vehicleType'].value_counts()
Out[  ]:
limousine     93614
kleinwagen    78014
kombi         65921
bus           29699
cabrio        22509
coupe         18386
suv           14477
andere         3125
Name: vehicleType, dtype: int64

In [  ]:
```

Acompanhando o retorno da função .value_counts( ) temos os números quanto aos agrupamentos. Esse dado nos será importante porque, podemos pegar o veículo que contem mais amostras, o mais comum deles, e usar como média para substituir os valores faltantes em outras categorias. A ideia é essa mesma, para não eliminarmos esses dados pode inconsistência, podemos preencher os espaços vazios com um valor médio.

```
22 base.loc[pd.isnull(base['vehicleType'])]
23 base['vehicleType'].value_counts()
24 base.loc[pd.isnull(base['gearbox'])]
25 base['gearbox'].value_counts()
26 base.loc[pd.isnull(base['model'])]
27 base['model'].value_counts()
28 base.loc[pd.isnull(base['fuelType'])]
29 base['fuelType'].value_counts()
30 base.loc[pd.isnull(base['notRepairedDamage'])]
31 base['notRepairedDamage'].value_counts()
```

Da mesma forma repetimos o processo para todas colunas onde houverem dados faltando (NaN), descobrindo os dados/valores médios para preencher tais espaços vazios.

```
33 valores_medios = {'vehicleType': 'limousine',
34                   'gearbox': 'manuell',
35                   'model': 'golf',
36                   'fuelType': 'benzin',
37                   'notRepairedDamage': 'nein'}
```

Em seguida criamos uma variável de nome valores_medios que recebe em forma de dicionário os parâmetros e seus respectivos dados/valores médios encontrados anteriormente.

```
39 base = base.fillna(value = valores_medios)
40
```

Uma vez identificados os dados faltantes em nossa base de dados, e descobertos os dados/valores médios para cada atributo previsor, hora de preencher esses espaços vários com dados. Aplicando sobre nossa variável base a função .fillna( ), passando como parâmetro os dados do dicionário valores_medios, finalmente fazemos a devida alimentação e temos uma base de dados onde possamos operar sem problemas.

```
41 entradas = base.iloc[:, 1:13].values
42 saidas = base.iloc[:, 0].values
```

Com nossa base de dados íntegra, sem colunas desnecessárias, sem dados inconsistentes ou faltantes, podemos finalmente dividir tais dados em dados de entrada e de saída. Para isso criamos uma variável de nome entradas que recebe todos os valores de todas as linhas e todos valores das colunas 1 até a 13 por meio dos métodos .iloc[ ] e .values. Da mesma forma, criamos uma variável de nome saidas que recebe os valores de todas as linhas da coluna 0.

Nome	Tipo	Tamanho	Valor
base	DataFrame	(359291, 15)	Column names: name, seller, offerType, price, abtest, vehicleType, yea ...
entradas	object	(359291, 12)	ndarray object of numpy module
saidas	object	(359291,)	ndarray object of numpy module
valores_medios	dict	5	{'vehicleType':'limousine', 'gearbox':
varTeste1	DataFrame	(12118, 15)	Column names: name, seller, offerType, price, abtest, vehicleType, yea ...
varTeste2	DataFrame	(115, 15)	Column names: name, seller, offerType, price, abtest, vehicleType, yea ...

Se o processo ocorreu como esperado, podemos ver que de fato foram criadas as variáveis entradas e saidas. Entradas por sua vez tem um total de 12 colunas de atributos previsores e 359291 linhas de amostras a serem processadas.

Note que estes números são bem inferiores aos da base em sua forma original, bruta. É absolutamente normal no processo de polimento eliminarmos boa parte da base de dados, em ciência de dados na verdade estamos mais preocupados com a integridade do que do número de dados a serem processados.

```
44 from sklearn.preprocessing import LabelEncoder
45
```

Como até o momento, mesmo após o polimento inicial, temos uma base de dados mista, contendo dados tanto em formato int (números) quanto string (texto), será necessário fazer a conversão desses dados todos para um tipo de dado de mesmo tipo.

Para isso, da mesma forma como já feito no exemplo anterior, usaremos da ferramenta LabelEncoder para converter tudo em números a serem processados pela rede neural. O processo de importação se dá como de costume.

```
46 labelencoder = LabelEncoder()
47 entradas[:,0] = labelencoder.fit_transform(entradas[:,0])
48 entradas[:,1] = labelencoder.fit_transform(entradas[:,1])
49 entradas[:,3] = labelencoder.fit_transform(entradas[:,3])
50 entradas[:,5] = labelencoder.fit_transform(entradas[:,5])
51 entradas[:,8] = labelencoder.fit_transform(entradas[:,8])
52 entradas[:,9] = labelencoder.fit_transform(entradas[:,9])
53 entradas[:,10] = labelencoder.fit_transform(entradas[:,10])
```

Como primeira parte desse processo de conversão, criamos nossa variável labelencoder que inicializa a ferramenta LabelEncoder, sem parâmetros mesmo. Na sequência sobre a coluna 0 de nossa base de dados aplicamos a função .fit_transform( ) que converte e atualiza os dados da coluna 0.

O processo é feito para todas as colunas onde teremos de converter texto em número, ou seja, é replicado o processo também nas colunas 1, 2, 5, 8, 9 e 10.

```
55 from sklearn.preprocessing import OneHotEncoder
56
```

Como última parte desse processo de conversão, precisamos transformar esses dados em algo que seja indexado e interpretado pelo interpretador de nossa IDE corretamente. Tal processo é feito pela ferramenta OneHotEncoder. Importada como de costume do módulo preprocessing da biblioteca sklearn.

```
57 onehotencoder = OneHotEncoder(categorical_features = [0,1,3,5,8,9,10])
58 entradas = onehotencoder.fit_transform(entradas).toarray()
```

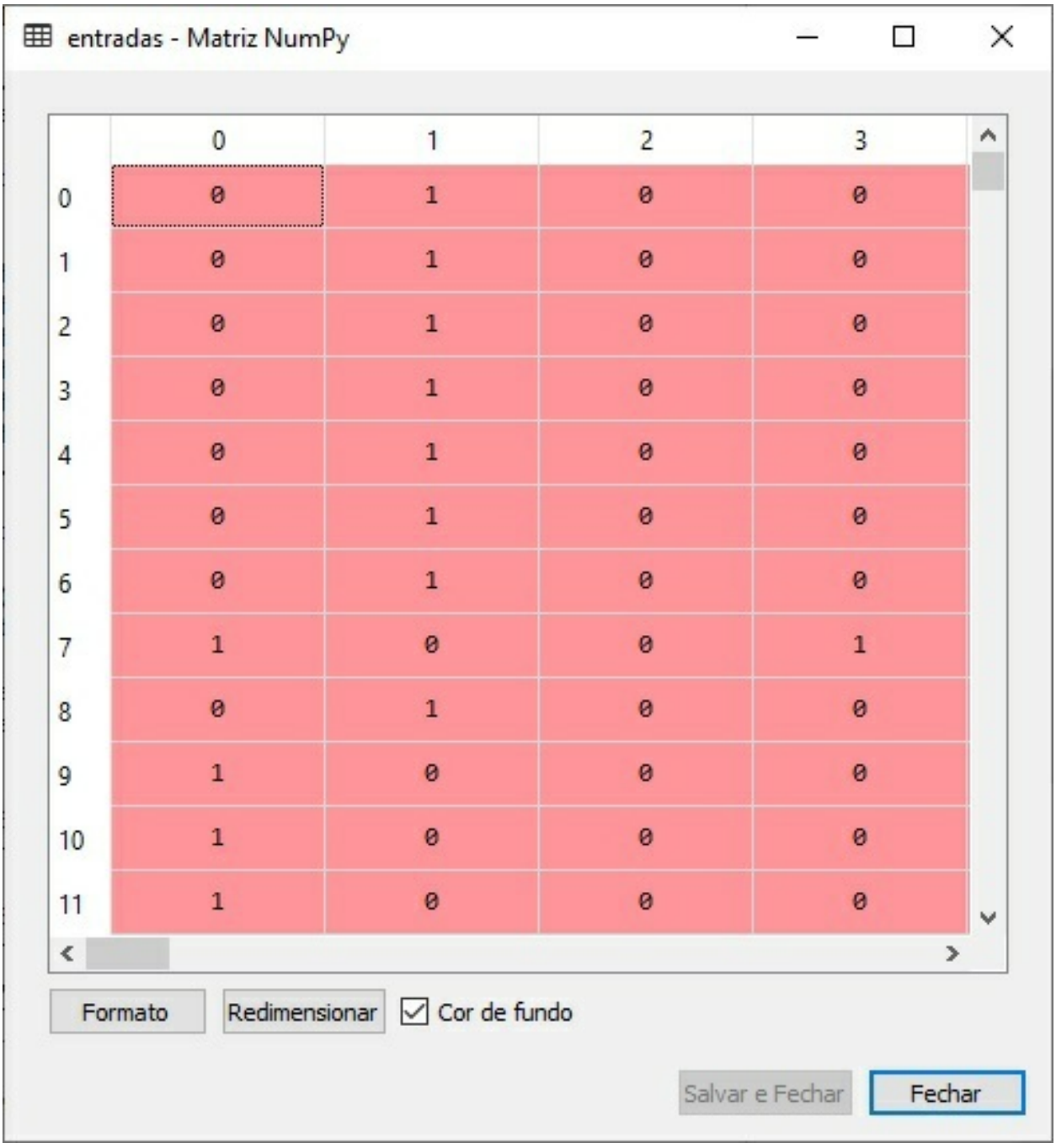
Em seguida criamos nossa variável de nome onehotencoder que

recebe como atributo a ferramenta OneHotEncoder, que por sua vez recebe como parâmetro categorical_features = [0,1,3,5,8,9,10], em outras palavras, agora é feito a transformação de números brutos para valores categóricos indexados internamente.

Logo após aplicamos sobre nossa variável entradas a função .fit_transform da ferramenta onehotencoder, convertendo tudo para uma array pela função .toarray( ).

Nome	Tipo	Tamanho	Valor
base	DataFrame	(359291, 12)	Column names: price, abtest, vehicleTy yearOfRegistration, gearbox, ...
entradas	float64	(359291, 316)	[[0.00e+00 1.00e+00 0.00e+00 ... 0.00e [0.00e+0 ...
saidas	int64	(359291,)	[480 18300 9800 ... 9200 3400 289
valores_medios	dict	5	{'vehicleType':'limousine', 'gearbox':
varTeste1	DataFrame	(12118, 12)	Column names: price, abtest, vehicleTy yearOfRegistration, gearbox, ...
varTeste2	DataFrame	(115, 12)	Column names: price, abtest, vehicleTy yearOfRegistration, gearbox, ...

Note que após feitas as devidas conversões houve inclusive uma mudança estrutural na forma de apresentação dos dados. Os dados de entradas agora por sua vez possuem 359291 amostras categorizadas em 316 colunas. Não se preocupe, pois, essa conversão é internamente indexada para que se possam ser aplicadas funções sobre essa matriz.



Também é possível agora abrir a variável e visualmente reconhecer seu novo padrão.

Finalmente encerrada toda a fase de polimento de nossos dados, podemos dar início a próxima etapa que é a criação da estrutura de nossa rede neural artificial.

```
60 from keras.models import Sequential
61 from keras.layers import Dense
```

Como de praxe, todo processo se inicia com as importações das bibliotecas e módulos que serão usados em nosso código e que nativamente não são carregados nem pré-alocados em memória. Para este exemplo também estaremos criando uma rede neural densa (multicamada interconectada) e sequencial (onde cada camada é ligada à sua camada subsequente.

```
63 regressor = Sequential()
64 regressor.add(Dense(units = 158,
65                     activation = 'relu',
66                     input_dim = 316))
67 regressor.add(Dense(units = 158,
68                     activation = 'relu'))
69 regressor.add(Dense(units = 1,
70                     activation = 'linear'))
71 regressor.compile(loss = 'mean_absolute_error',
72                   optimizer = 'adam',
73                   metrics = ['mean_absolute_error'])
```

A estrutura se mantém muito parecida com as que construímos anteriormente, primeira grande diferença é que no lugar de um classificador estamos criando um regressor. Note que na primeira camada existem 316 neurônios, na segunda 158 e na última apenas 1.

Outro ponto importante de salientar é que o processo de ativação da camada de saída agora é ‘linear’, este tipo de parâmetro de ativação é o mais comum de todos, uma vez que ele na verdade não aplica nenhuma função ao neurônio de saída, apenas pega o dado que consta nele e replica como valor final.

No processo de compilação repare que tanto a função de perda quanto a métrica é definida como ‘mean_absolute_error’, em outras palavras, os pesos a serem corrigidos serão considerados de forma normal, absoluta, consultando a documentação do Keras você verá que outra função muito utilizada é a ‘squared_absolute_error’, onde o valor de erro é elevado ao quadrado e reaplicado na função como forma de penalizar mais os dados incorretos no processamento, ambos os métodos são válidos em um sistema de regressão, inclusive é interessante você realizar seus testes fazendo o uso dos dois.

```
74 regressor.fit(entradas,
75                saidas,
76                batch_size = 300,
77                epochs = 100)
```

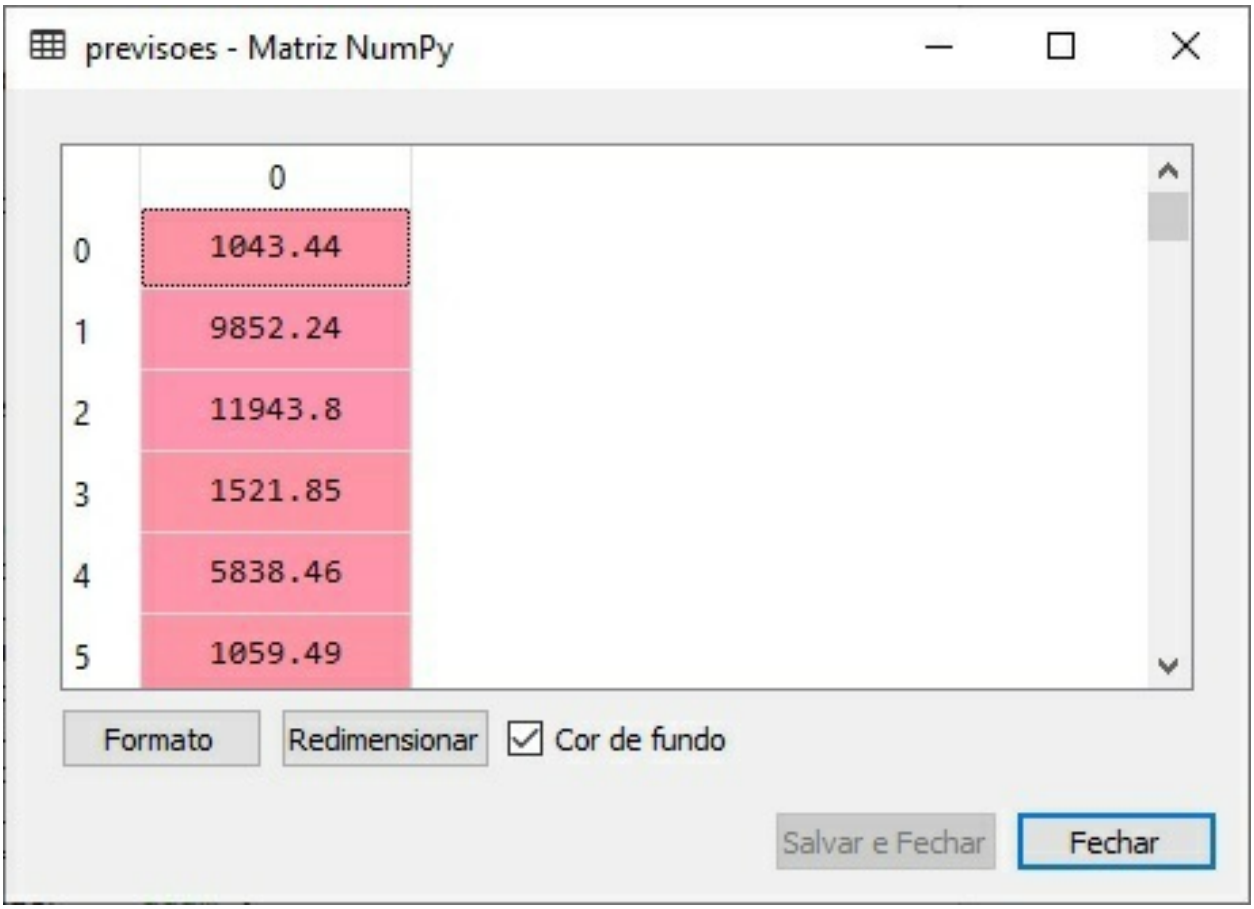
Estrutura criada, por meio da função .fit( ) alimentamos nossa rede com os dados de entradas, saidas, definimos que os pesos serão atualizados a cada 300 amostras e que a rede por sua vez será executada 100 vezes. Como sempre, é interessante testar taxas de atualização diferentes assim como executar a rede mais vezes a fim de que ela por aprendizado de reforço encontre melhores resultados.

```
Epoch 96/100
359291/359291 [==============================] - 7s 18us/step - loss: 2238.8938
mean_absolute_error: 2238.8938
Epoch 97/100
359291/359291 [==============================] - 7s 18us/step - loss: 2248.9770
mean_absolute_error: 2248.9770
Epoch 98/100
359291/359291 [==============================] - 7s 18us/step - loss: 2237.1569
mean_absolute_error: 2237.1569
Epoch 99/100
359291/359291 [==============================] - 7s 18us/step - loss: 2238.6745
mean_absolute_error: 2238.6745
Epoch 100/100
359291/359291 [==============================] - 7s 18us/step - loss: 2242.1481
mean_absolute_error: 2242.1481
Out[ ]:
```

Terminado o processamento da rede note que o valor retornado foi de 2242.14, o que em outras palavras significa que a variação de valor desses carros em geral varia em 2242,14 euros para mais ou para menos.

```
79 previsoes = regressor.predict(entradas)
80 saidas.mean()
81 previsoes.mean()
```

Como de costume, sobre esses dados iniciais podemos aplicar nossos métodos preditivos. Para isso criamos uma variável de nome previsoes que aplica a função .predict( ) em nosso regressor passando os dados contidos em entradas como parâmetro.



Explorando de forma visual a variável previsoes podemos ver que de fato é feito um processamento interno usando outras funções aritméticas que resultam em diferentes valores para cada uma das amostras.

```
mean_absolute_error: 2238.6745
Epoch 100/100
359291/359291 [==============================] - 7s 18us/step - loss: 2242.1481
mean_absolute_error: 2242.1481
Out[ ]: <keras.callbacks.History at 0x2534d0f8da0>

In [ ]: previsoes = regressor.predict(entradas)
   ...: saidas.mean()
   ...: previsoes.mean()
Out[ ]: 2357.3423

In [ ]: saidas.mean()
Out[ ]: 2916.833945186492
```

Via console podemos notar a proximidade dos resultados obtidos, o que nos indica que a rede está configurada corretamente. Uma grande disparidade entre esses valores pode indicar erros de processamento da rede.

```
83 from sklearn.model_selection import cross_val_score
84 from keras.wrappers.scikit_learn import KerasRegressor
```

Dando sequência podemos realizar sobre este modelo a mesma técnica de validação cruzada aplicada no modelo anterior. Como sempre, importamos o que é necessário, nesse caso a ferramenta cross_val_score do módulo model_selection da biblioteca sklearn.

Também estaremos aplicando e comparando resultados por meio do regressor que a biblioteca keras já tem dentro de si pré-configurado. Para isso importamos KerasRegressor do módulo wrappers.scikit_learn da biblioteca keras.

```
86 def regressorValCruzada():
87     regressorV = Sequential()
88     regressorV.add(Dense(units = 158,
89                          activation = 'relu',
90                          input_dim = 316))
91     regressorV.add(Dense(units = 158,
92                          activation = 'relu'))
93     regressorV.add(Dense(units = 1,
94                          activation = 'linear'))
95     regressorV.compile(loss = 'mean_absolute_error',
96                        optimizer = 'adam',
97                        metrics = ['mean_absolute_error'])
98     return regressorV
```

Prosseguindo, criamos nosso regressorValCruzada( ) em forma de função. Internamente repare que a estrutura de rede neural é a mesma anterior. Usaremos essa função/rede para realizar novos testes.

```
100 regValCruzada = KerasRegressor(build_fn = regressorValCruzada,
101                                epochs = 100,
102                                batch_size = 300)
103 resValCruzada = cross_val_score(estimator = regValCruzada,
104                                  X = entradas,
105                                  y = saidas,
106                                  cv = 10,
107                                  scoring = 'neg_mean_absolute_error')
```

Para isso criamos nossa variável regValCruzada que recebe como atributo o KerasRegressor, que por sua vez tem como parâmetro build_fn = regressorValCruzada, o que em outras palavras ele inicializa nossa função criada antes internamente, executando a rede 100 vezes, atualizando os pesos a cada 300 amostras.

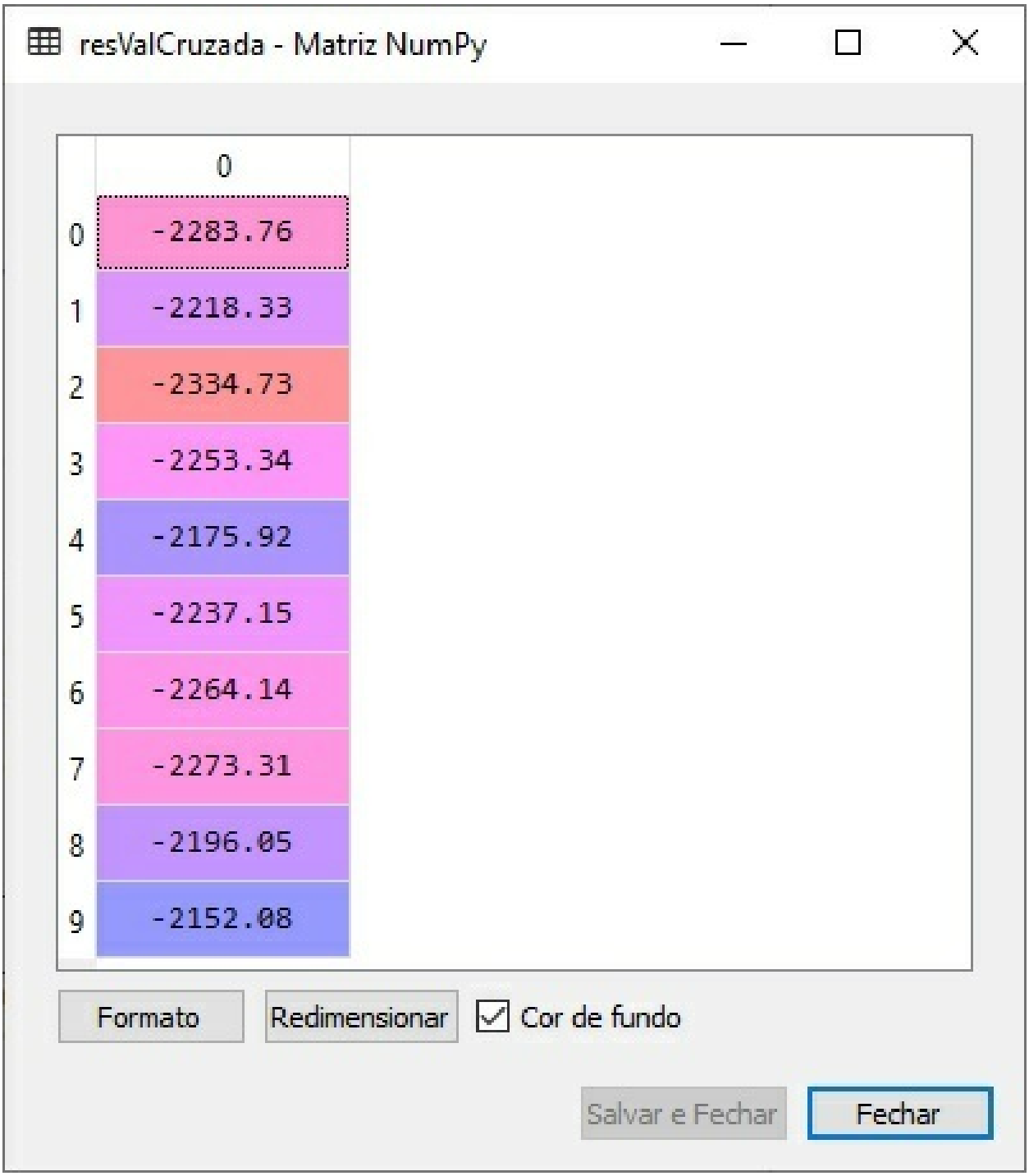
Na sequência, criamos nossa variável de nome resValCruzada, que

irá executar a validação cruzada propriamente dita, então, por sua vez, resValCruzada recebe como atributo cross_val_score que tem como parâmetros estimator = regValCruzada, ou seja, aplica seu processamento sobre os dados obtidos de nossa função regressorValCruzada( ), usando X e y como dados de parâmetro, dados a serem usados diretamente de nossa base para comparação.

Na sequência é definido que cv = 10, em outras palavras, toda base será dividida em 10 partes iguais e testadas individualmente, por fim scoring = ‘neg_mean_absolute_error’ fará a apresentação da média dos resultados obtidos, desconsiderando o sinal dos mesmos, apenas os números em seu estado bruto.

```
Epoch 96/100
323361/323361 [==============================] - 7s 21us/step - loss: 2266.0997
mean_absolute_error: 2266.0997
Epoch 97/100
323361/323361 [==============================] - 8s 26us/step - loss: 2254.7366
mean_absolute_error: 2254.7366
Epoch 98/100
323361/323361 [==============================] - 8s 24us/step - loss: 2248.4811
mean_absolute_error: 2248.4811
Epoch 99/100
323361/323361 [==============================] - 8s 24us/step - loss: 2250.4997
mean_absolute_error: 2250.4997
Epoch 100/100
323361/323361 [==============================] - 6s 19us/step - loss: 2253.2591
mean_absolute_error: 2253.2591
```

Selecionando e executando todo bloco de código é possível acompanhar via console a rede neural sendo executada, lembrando que esta etapa costuma ser bastante demorada uma vez que toda rede será executada uma vez para treino e mais 10 vezes para testar cada parte da base.



Terminado o processo de validação cruzada é possível ver os valores obtidos para cada uma das partes da base de dados testada individualmente.

```
108 media = resValCruzada.mean()
109 desvioPadrao = resValCruzada.std()
```

Finalizando nosso exemplo, criamos as variáveis dedicadas a apresentar os resultados finais de nossa validação cruzada para as devidas comparações com os dados de nossa rede neural artificial.

Nome	Tipo	Tamanho	Valor
base	DataFrame	(359291, 12)	Column names: price, abtest, vehicleTy
desvioPadrao	float64	1	51.95240597776519
entradas	float64	(359291, 316)	[[0.00e+00 1.00e+00 0.00e+00 ... 0.00e [0.00e+0 ...
media	float64	1	-2238.8809286392097
previsoes	float32	(359291, 1)	[[1043.4368] [9852.238]
resValCruzada	float64	(10,)	[-2283.76042768 -2218.33065824 -2334.7 -2175.92 ...
saidas	int64	(359291,)	[480 18300 9800 ... 9200 3400 28990]

E o importante a destacar é que, como nos exemplos anteriores, estes testes são realizados como um viés de confirmação para termos certeza da integridade de nossos dados, aqui, realizados os devidos testes, podemos concluir que os resultados são próximos, confirmando que a execução de nossa rede se deu de forma correta.

Código Completo:

```
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
from keras.models import Sequential
from keras.layers import Dense
from sklearn.model_selection import cross_val_score
from keras.wrappers.scikit_learn import KerasRegressor

base = pd.read_csv('autos.csv', encoding = 'ISO-8859-1')

base = base.drop('dateCrawled', axis=1)
base = base.drop('dateCreated', axis=1)
base = base.drop('nrOfPictures', axis=1)
base = base.drop('postalCode', axis=1)
base = base.drop('lastSeen', axis=1)
base = base.drop('name', axis=1)
base = base.drop('seller', axis=1)
base = base.drop('offerType', axis=1)

varTeste1 = base.loc[base.price <= 10]
base = base[base.price > 10]
varTeste2 = base.loc[base.price > 350000]
base = base[base.price < 350000]
```

```
base.loc[pd.isnull(base['vehicleType'])]
base['vehicleType'].value_counts()
base.loc[pd.isnull(base['gearbox'])]
base['gearbox'].value_counts()
base.loc[pd.isnull(base['model'])]
base['model'].value_counts()
base.loc[pd.isnull(base['fuelType'])]
base['fuelType'].value_counts()
base.loc[pd.isnull(base['notRepairedDamage'])]
base['notRepairedDamage'].value_counts()

valores_medios = {'vehicleType': 'limousine',
                  'gearbox': 'manuell',
                  'model': 'golf',
                  'fuelType': 'benzin',
                  'notRepairedDamage': 'nein'}

base = base.fillna(value = valores_medios)
entradas = base.iloc[:, 1:13].values
saidas = base.iloc[:, 0].values

labelencoder = LabelEncoder()
entradas[:,0] = labelencoder.fit_transform(entradas[:,0])
entradas[:,1] = labelencoder.fit_transform(entradas[:,1])
entradas[:,3] = labelencoder.fit_transform(entradas[:,3])
entradas[:,5] = labelencoder.fit_transform(entradas[:,5])
entradas[:,8] = labelencoder.fit_transform(entradas[:,8])
entradas[:,9] = labelencoder.fit_transform(entradas[:,9])
entradas[:,10] = labelencoder.fit_transform(entradas[:,10])

onehotencoder = OneHotEncoder(categorical_features = [0,1,3,5,8,9,10])
entradas = onehotencoder.fit_transform(entradas).toarray()

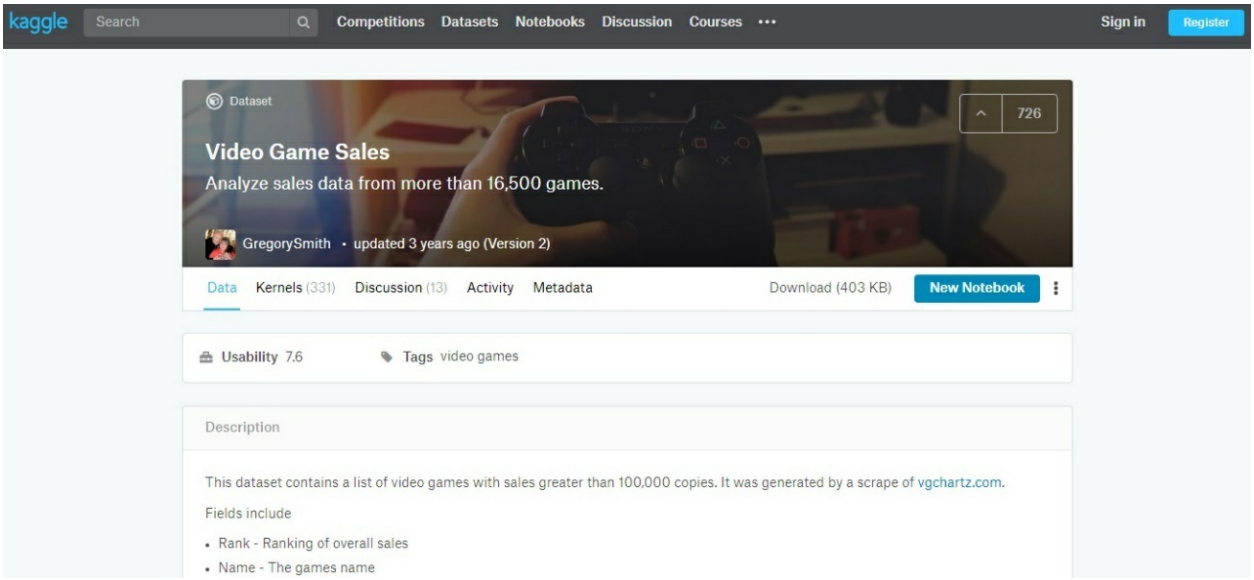
regressor = Sequential()
regressor.add(Dense(units = 158,
                    activation = 'relu',
                    input_dim = 316))
regressor.add(Dense(units = 158,
                    activation = 'relu'))
regressor.add(Dense(units = 1,
                    activation = 'linear'))
regressor.compile(loss = 'mean_absolute_error',
                  optimizer = 'adam',
                  metrics = ['mean_absolute_error'])
regressor.fit(entradas,
              saidas,
              batch_size = 300,
              epochs = 100)

previsoes = regressor.predict(entradas)
saidas.mean()
previsoes.mean()
```

```
78 def regressorValCruzada():
79     regressorV = Sequential()
80     regressorV.add(Dense(units = 158,
81                          activation = 'relu',
82                          input_dim = 316))
83     regressorV.add(Dense(units = 158,
84                          activation = 'relu'))
85     regressorV.add(Dense(units = 1,
86                          activation = 'linear'))
87     regressorV.compile(loss = 'mean_absolute_error',
88                        optimizer = 'adam',
89                        metrics = ['mean_absolute_error'])
90     return regressorV
91
92 regValCruzada = KerasRegressor(build_fn = regressorValCruzada,
93                                epochs = 100,
94                                batch_size = 300)
95 resValCruzada = cross_val_score(estimator = regValCruzada,
96                                 X = entradas,
97                                 y = saidas,
98                                 cv = 10,
99                                 scoring = 'neg_mean_absolute_error')
100 media = resValCruzada.mean()
101 desvioPadrao = resValCruzada.std()
```

Regressão com Múltiplas Saídas – VGDB Dataset

Uma das aplicações bastante comuns em ciência de dados é tentarmos estipular números de vendas de um determinado produto, para a partir disso gerar algum prospecto de mudanças em seu modelo de vendas. Aqui, para entendermos esse tipo de conceito de forma prática, usaremos uma base de dados de vendas de jogos de videogame onde temos as informações de suas vendas em três mercados diferentes. Temos dados brutos das vendas dos jogos na américa do norte, na europa e no japão, após o devido polimento e processamento dos dados dessa base, poderemos ver qual região obteve maior lucro em suas vendas para assim planejar estratégias de vendas nas outras de menor lucro. Para isto estaremos usando uma base de dados real disponível gratuitamente no repositório Kaggle.



Dentro do site do Kaggle, basta procurar por Video Game Sales e da página do dataset fazer o download do respectivo arquivo em formato .csv.

Partindo para o código:

VGDB.py

```
1 import pandas as pd
2 from keras.layers import Dense, Input
3 from keras.models import Model
```

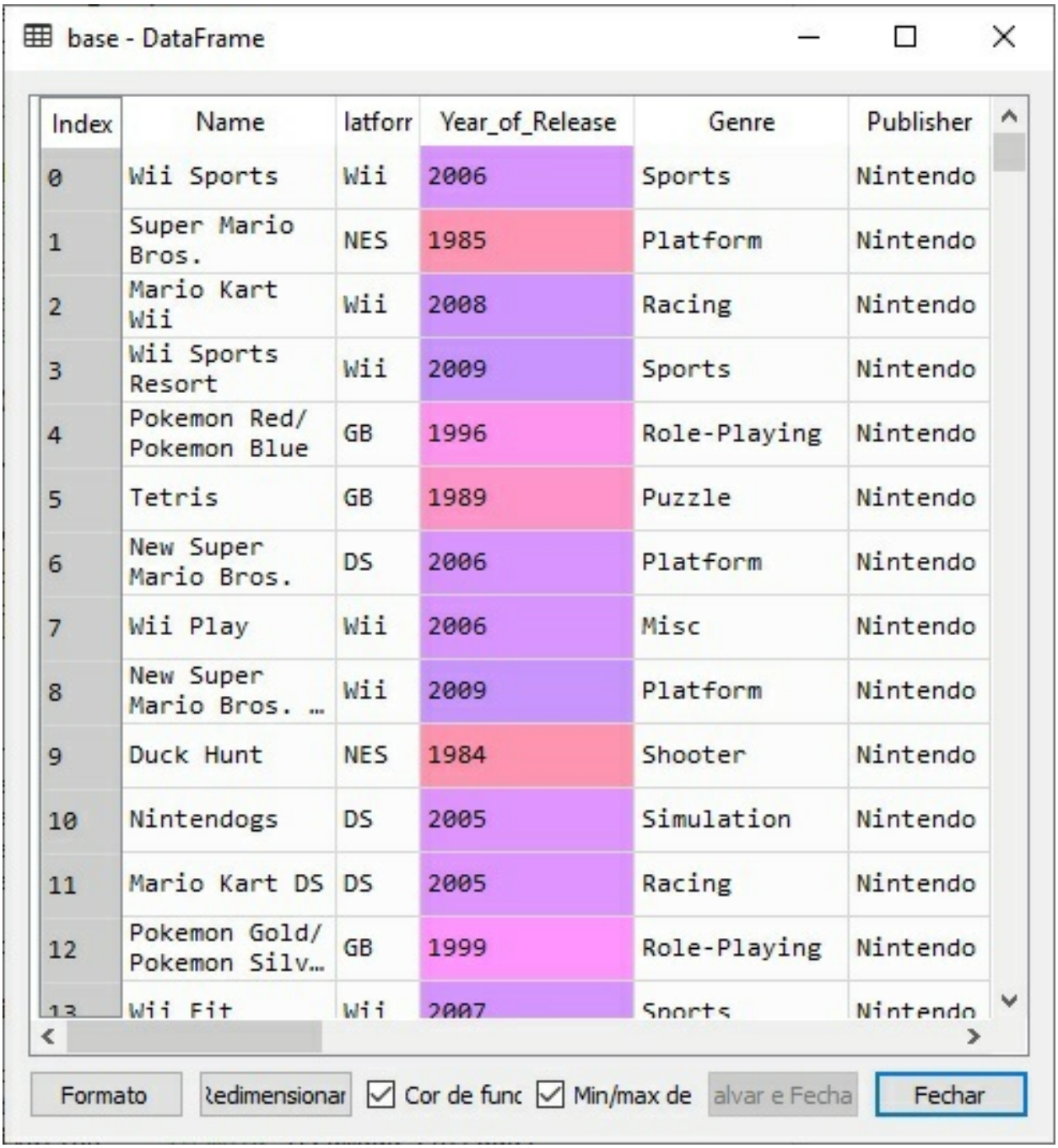
Inicialmente criamos um novo arquivo Python 3 de nome VGDB.py, em seguida, como sempre fazemos, realizamos as devidas importações das bibliotecas, módulos e ferramentas que nos auxiliarão ao longo do código. Repare que aqui temos uma pequena diferença em relação aos exemplos anteriores, não importamos Sequential (módulo que criava e linkava automaticamente as camadas de nossa rede neural) e importamos o módulo Model, que por sua vez nos permitirá criar uma rede neural densa com múltiplas saídas.

```
2 pszs = bq·ls9q¯csv(,8swsz·csv,)
```

Em seguida criamos uma variável de nome base que recebe por meio da função .read_csv( ) o conteúdo de nosso arquivo games.csv parametrizado aqui.

Nome	Tipo	Tamanho	Valor
base	DataFrame	(16719, 16)	Column names: Name, Platform, Year_of_Release, Ger Publisher, NA_Sa ...

Uma vez realizada a importação é possível via explorador de variáveis ver que de fato foi criada nossa variável base em forma de DataFrame, onde inicialmente temos 16719 objetos/amostras/registros com atributos divididos em 16 categorias diferentes.



Analisando o conteúdo de base, podemos ver que, como de costume, temos muitas inconsistências nos dados, muitos dados faltando e por fim, colunas de informação desnecessárias para nosso propósito. Sendo assim iniciamos a fase de polimento dos dados removendo as colunas desnecessárias.

```
base = base.drop('Other_Sales', axis = 1)
base = base.drop('Global_Sales', axis = 1)
base = base.drop('Developer', axis = 1)
base = base.dropna(axis = 0)
base = base.loc[base['NA_Sales'] > 1]
base = base.loc[base['EU_Sales'] > 1]
```

Diretamente sobre nossa variável base aplicaremos algumas funções. Primeiramente por meio da função .drop( ) eliminamos as colunas Other_Sales, Global_Sales e Developer, note que para eliminar colunas de nossa base devemos colocar como parâmetro também axis = 1.

Na sequência por meio da função .dropna( ) excluímos todas as linhas onde haviam valores faltando (não zerados, faltando mesmo), e agora como estamos aplicando função sobre linhas, agora axis = 0. Por fim, por meio da expressão .loc[base['NA_Sales'] > 1] mantemos dessa coluna apenas as linhas com valores maiores que 1, o mesmo é feito para EU_ Sales.

Nome	Tipo	Tamanho	Valor
base	DataFrame	(258, 13)	Column names: Name, Platform, Year_of_Release, Genre Publisher, NA_Sa ...

Repare que após a aplicação das funções anteriores, houve uma redução considerável no número de amostras que temos. O que indica que dentro dos parâmetros que definimos haviam muitas inconsistências de dados, e isto é perfeitamente normal quando pegarmos bases de dados geradas tanto por sistemas quanto por usuários, o importante é que dos poucos dados que sobrem, estes sejam íntegros para que possamos processá-los e extrair informações a partir dos mesmos.

```
base = base.rob(,Name,' axis = 1)
backupName = base.Name
base['Name'].value_counts()
```

Dando sequência realizaremos um procedimento de extrair os dados de uma coluna inteira de nossa base, porém não iremos descartar tais dados, apenas separá-los da base salvando-os em uma nova variável. Para isso aplicando sobre a coluna Name de nossa base a função .value_counts( ) fazemos a leitura de todos os dados/valores que houverem nessa coluna.

Em seguida criamos uma variável de nome backupBase que recebe os dados contidos em base.Names. Por fim, por meio da função .drop( ) eliminamos de nossa base a coluna inteira Name.

```
18 entradas = base.iloc[:, [0,1,2,3,7,8,9,10,11]].values
19 vendas_NA = base.iloc[:, 4].values
20 vendas_EU = base.iloc[:, 5].values
21 vendas_JP = base.iloc[:, 6].values
```

Eliminadas as inconsistências de nossa base de dados, hora de dividir a mesma em dados de entrada e de saída. Inicialmente criamos uma variável de nome entradas que recebe como atributo todos os dados das colunas 0, 1, 2, 3, 7, 8, 9, 10 e 11 por meio de .iloc[ ].values.

Na sequência criamos 3 variáveis de saída, uma vez que neste exemplo é isto que queremos. A variável vendas_NA recebe os dados de todas as linhas da coluna 4, vendas_EU recebe o conteúdo de todas as linhas da coluna 5 e por fim vendas_JP recebe todas as linhas da coluna 6.

Nome	Tipo	Tamanho	Valor
backupName	Series	(258,)	Series object of pandas.core.series module
base	DataFrame	(258, 12)	Column names: Platform, Year_of_Release, Genre Publisher, NA_Sales, E ...
entradas	object	(258, 9)	ndarray object of numpy module
vendas_EU	float64	(258,)	[28.96 12.76 10.93 ... 1.05 1.12 1.19]
vendas_JP	float64	(258,)	[3.77 3.79 3.28 ... 0.24 0.06 0.]
vendas_NA	float64	(258,)	[41.36 15.68 15.61 ... 1.05 1.13 1.07]

Se não houver nenhum erro de sintaxe nesse processo podemos ver que as devidas variáveis foram criadas. Note que temos dados numéricos e em forma de texto, sendo assim, próxima etapa da fase de polimento é converter todos dados para tipo numérico para que possam ser processados pela rede.

```
22
23 from sklearn.preprocessing import LabelEncoder, OneHotEncoder
```

Como sempre, realizamos as devidas importações das ferramentas que inicialmente não são pré-carregadas por nossa IDE.

```
25 labelencoder = LabelEncoder()
26 entradas[:,0] = labelencoder.fit_transform(entradas[:,0])
27 entradas[:,2] = labelencoder.fit_transform(entradas[:,2])
28 entradas[:,3] = labelencoder.fit_transform(entradas[:,3])
29 entradas[:,8] = labelencoder.fit_transform(entradas[:,8])
```

Dando sequência criamos nossa variável labelencoder que inicializa a ferramenta LabelEncoder, sem parâmetros mesmo. Na sequência é aplicada a função .fit_transform( ) sobre as colunas 0, 2, 3 e 8, transformando seus dados do tipo string para int.

```
34 entradas = onehotencoder.fit_transform(entradas).toarray()
35 onehotencoder = OneHotEncoder(categorical_features = [0,2,3,8])
```

Próximo passo é tornar esse conteúdo convertido indexado de forma que nosso interpretador consiga fazer a correta leitura dos dados e aplicar funções sobre os mesmos. Logo, criamos nossa variável de nome onehotencoder que executa a ferramenta OneHotEncoder e seu parâmetro categorical_features sobre as colunas 0, 2, 3 e 8. Por fim, nossa variável entradas recebe a conversão de tais dados para uma array do tipo numpy por meio das funções .fit_transform( ).toarray( ).

```
34 camada_entrada = Input(shape = (61, ))
35 camada_oculta1 = Dense(units = 32,
36                        activation = 'sigmoid')(camada_entrada)
37 camada_oculta2 = Dense(units = 32,
38                        activation = 'sigmoid')(camada_oculta1)
39 camada_saida1 = Dense(units = 1,
40                       activation = 'linear')(camada_oculta2)
41 camada_saida2 = Dense(units = 1,
42                       activation = 'linear')(camada_oculta2)
43 camada_saida3 = Dense(units = 1,
44                       activation = 'linear')(camada_oculta2)
```

Finalmente terminada a fase de polimento dos dados, damos início a criação da estrutura de nossa rede neural artificial. Nos modelos anteriores estivemos criando redes neurais sequenciais.

Aqui criaremos um modelo um pouco diferente, ainda sequencial mas iremos criar os laços entre as camadas de forma totalmente manual, por fim, a segunda grande diferença é que estaremos criando um modelo de rede que processará uma base de dados para gerar múltiplas saídas independentes, uma vez que aqui estamos trabalhando para descobrir preços de vendas em 3 locações diferentes.

Inicialmente criamos nossa variável camada_entrada que recebe como atributo Input que por sua vez recebe o formado da array gerada anteriormente, note que não estamos pegando as amostras e transformando em neurônios da camada de entrada, as amostras por sua vez foram convertidas e indexadas como parte de uma matriz.

Em seguida criamos uma variável de nome camada_oculta1 que recebe como atributo Dense que por sua vez tem como parâmetros units = 32 e activation = ‘sigmoid’, em outras palavras, nossa rede que tem 61 neurônios em sua camada de entrada, agora tem 32 em sua primeira camada oculta, e o processamento desses neurônios com seus respectivos pesos deve gerar uma probabilidade entre 0 e 1 por causa da função de ativação escolhida ser a sigmoide.

Por fim, para haver a comunicação entre uma camada e outra a referência da camada é passada como um segundo parâmetro independente. Para finalizar são criadas 3 camadas de saída, cada uma com 1 neurônio, activation = linear (ou seja, não é aplicada nenhuma função, apenas replicado o valor encontrado) e as devidas linkagens com camada_oculta2.

```
46 regressor = Model(inputs = camada_entrada,
47                   outputs = [camada_saida1,
48                              camada_saida2,
49                              camada_saida3])
50 regressor.compile(optimizer = 'adam',
51                   loss = 'mse')
52 regressor.fit(entradas,
53               [vendas_NA, vendas_EU, vendas_JP],
54               epochs = 5000,
55               batch_size = 100)
```

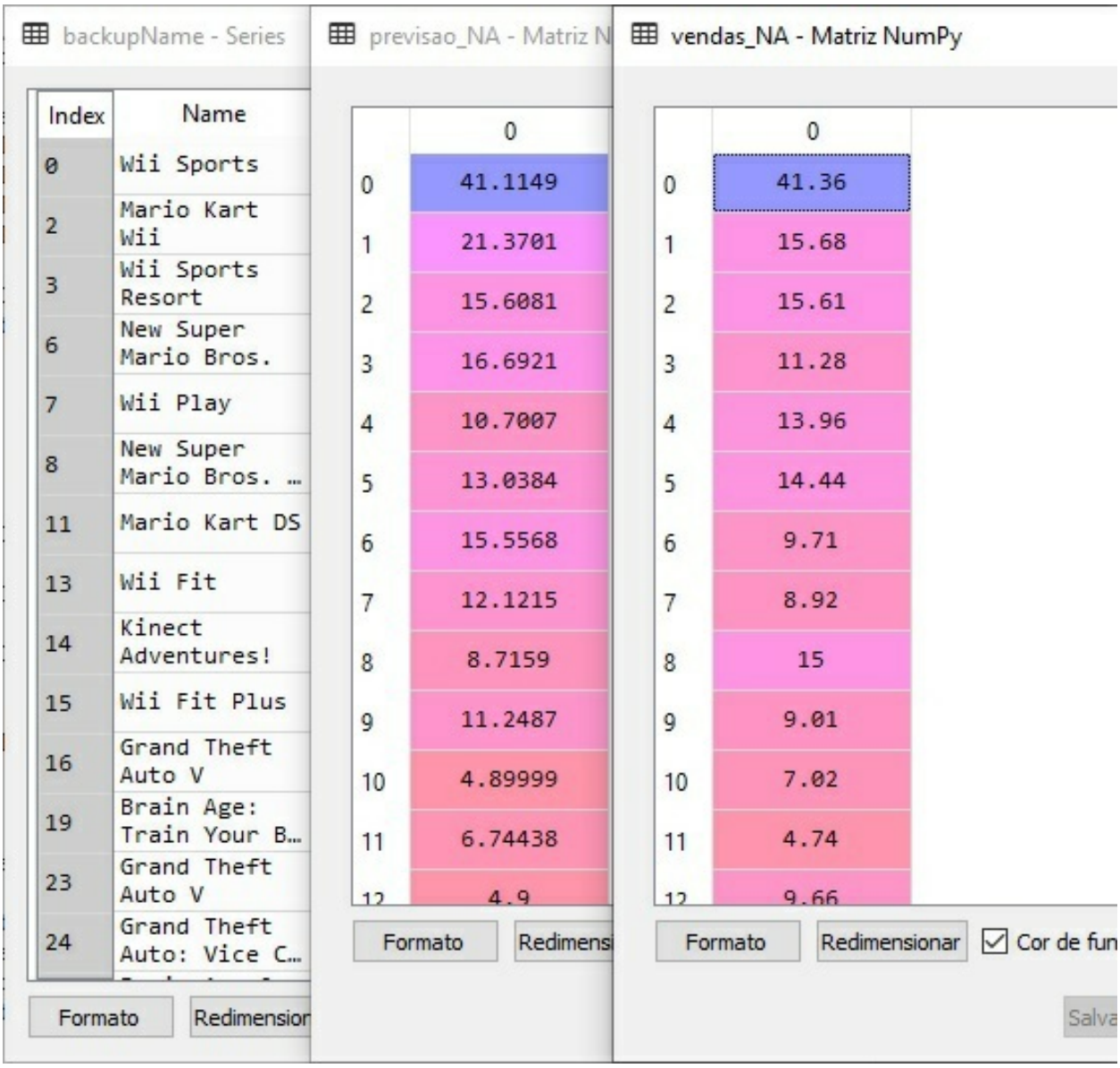
Em seguida é criado nosso regressor, por meio da variável homônima, que recebe como atributo Model que por sua vez tem como parâmetros inputs = camada_entrada e outputs = [camada_saida1, camada_saida2, camada_saida3].

Logo após é executado sobre nosso regressor a função .compile que define o modo de compilação do mesmo, optimizer = ‘adam’ e loss = ‘mse’. Adam já é um otimizador conhecido nosso de outros modelos, mas neste caso ‘mse’ significa mean squared error, é um modelo de função de perda onde o resultado encontrado é elevado ao quadrado e reaplicado na rede, é uma forma de dar mais peso aos erros e forçar a rede a encontrar melhores parâmetros de processamento.

Por fim alimentamos nosso regressor por meio da função .fit( ) com os respectivos dados de entrada e de saída, assim como definimos que a rede será executada 5000 vezes, atualizando os pesos a cada 100 amostras.

```
previsao_NA, previsao_EU, previsao_JP = regressor.predict(entradas)
#previsao
```

Como de costume, é interessante realizar previsões e as devidas comparações para testarmos a eficiência de nossa rede. Para isso criamos as variáveis previsao_NA, previsao_EU e previsao_JP que recebe nosso regressor, aplicando a função .predict( ) diretamente sobre nossos dados de entradas.



Por fim podemos equiparar e comparar as colunas dos respectivos nomes dos jogos, os valores reais extraídos de nossa base de dados e os valores encontrados por nossa rede neural artificial, note que de fato os dados

são próximos, ou pelo menos possuem uma margem de erro dentro do aceitável. Sendo assim, podemos salvar esse modelo para reutilizar quando for necessário resolver algum tipo de problema parecido.

Código Completo:

```
import pandas as pd
from keras.layers import Dense, Input
from keras.models import Model
from sklearn.preprocessing import LabelEncoder, OneHotEncoder

base = pd.read_csv('games.csv')
base = base.drop('Other_Sales', axis = 1)
base = base.drop('Global_Sales', axis = 1)
base = base.drop('Developer', axis = 1)
base = base.dropna(axis = 0)
base = base.loc[base['NA_Sales'] > 1]
base = base.loc[base['EU_Sales'] > 1]
base['Name'].value_counts()
backupName = base.Name
base = base.drop('Name', axis = 1)

entradas = base.iloc[:, [0,1,2,3,7,8,9,10,11]].values
vendas_NA = base.iloc[:, 4].values
vendas_EU = base.iloc[:, 5].values
vendas_JP = base.iloc[:, 6].values

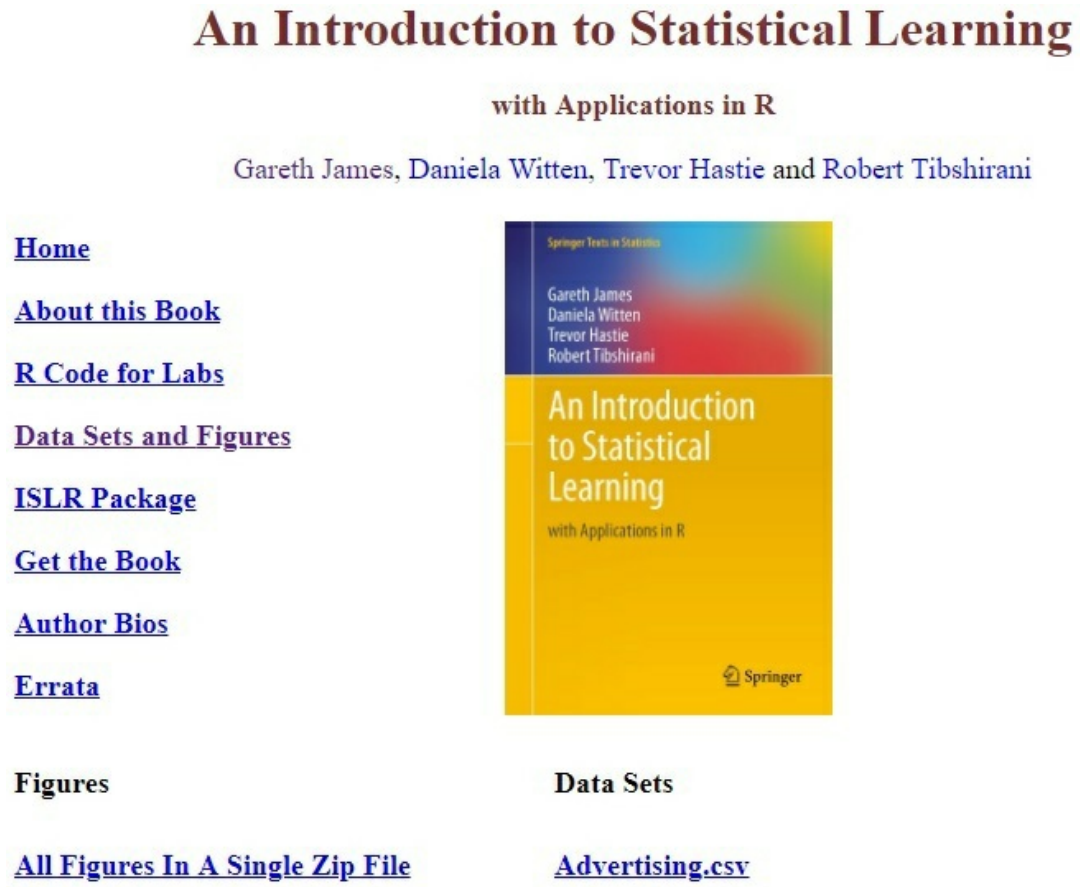
labelencoder = LabelEncoder()
entradas[:,0] = labelencoder.fit_transform(entradas[:,0])
entradas[:,2] = labelencoder.fit_transform(entradas[:,2])
entradas[:,3] = labelencoder.fit_transform(entradas[:,3])
entradas[:,8] = labelencoder.fit_transform(entradas[:,8])
onehotencoder = OneHotEncoder(categorical_features = [0,2,3,8])
entradas = onehotencoder.fit_transform(entradas).toarray()
```

```
camada_entrada = Input(shape = (61, ))
camada_oculta1 = Dense(units = 32,
                       activation = 'sigmoid')(camada_entrada)
camada_oculta2 = Dense(units = 32,
                       activation = 'sigmoid')(camada_oculta1)
camada_saida1 = Dense(units = 1,
                      activation = 'linear')(camada_oculta2)
camada_saida2 = Dense(units = 1,
                      activation = 'linear')(camada_oculta2)
camada_saida3 = Dense(units = 1,
                      activation = 'linear')(camada_oculta2)

regressor = Model(inputs = camada_entrada,
                  outputs = [camada_saida1,
                             camada_saida2,
                             camada_saida3])
regressor.compile(optimizer = 'adam',
                  loss = 'mse')
regressor.fit(entradas,
              [vendas_NA, vendas_EU, vendas_JP],
              epochs = 5000,
              batch_size = 100)
#previsor
previsao_NA, previsao_EU, previsao_JP = regressor.predict(entradas)
```

Previsão Quantitativa – Publi Dataset

Uma das aplicações de ciência de dados bastante recorrente é a de identificar algum retorno (em valores monetários) com base no investimento feito. Em outras palavras é possível identificar via processamento de aprendizado de máquina, se o investimento em um determinado nicho surtiu de fato algum resultado. Aqui estaremos usando uma database que com base no investimento em algumas categorias de publicidade de um determinado produto, possamos ver onde esse investimento surtiu melhor efeito, levando em consideração qual foi o retorno de acordo com cada unidade monetária investida.



A base de dados que usaremos dessa vez não está nem no UCI nem no Kaggle, porém você pode ter acesso a essa base de dados por meio do site acima. Clicando sobre Adversiting.csv você fará o download do referente arquivo.

Advertising Dataset.py

```
1 import pandas as pd
2
```

Como sempre, inicialmente criamos um arquivo Python 3 para nosso código e realizamos as devidas importações iniciais.

```
3 base = pd.read_csv('Advertising.csv')
4 print(base.head())
```

Em seguida criamos nossa variável de nome base que irá receber todo o conteúdo de Adversising.csv por meio da função .read_csv( ) do pandas. A partir desse momento se o processo de importação ocorreu sem erros, a variável base será devidamente criada. Uma das maneiras de visualizar rapidamente o seu conteúdo (principalmente se você estiver usando outra IDE diferente do Spyder) é executar o comando print( ) parametrizado com base.head( ), dessa forma será exibida em console as 5 primeiras linhas desse dataframe.

Console 1/A

```
In [ ]: import pandas as pd
   ...:
   ...: base = pd.read_csv('Advertising.csv')

In [ ]: print(base.head())
   Unnamed: 0     TV  radio  newspaper  sales
0           1  230.1   37.8       69.2   22.1
1           2   44.5   39.3       45.1   10.4
2           3   17.2   45.9       69.3    9.3
3           4  151.5   41.3       58.5   18.5
4           5  180.8   10.8       58.4   12.9
```

Note que existem 6 colunas, sendo duas delas indexadores, um da própria planilha Excel e outro gerado pelo pandas durante a importação. Essas colunas simplesmente não serão utilizadas como referência nesse modelo. Também temos 3 colunas com atributos previsores (TV, radio e newspaper) e uma coluna com dados de saída (sales).

O que iremos fazer nesse modelo é executar algumas funções interessantes fora de uma rede neural densa, apenas para também visualizar estas possibilidades, uma vez que dependendo muito da complexidade do problema o mesmo não precisa ser processado por uma rede neural.

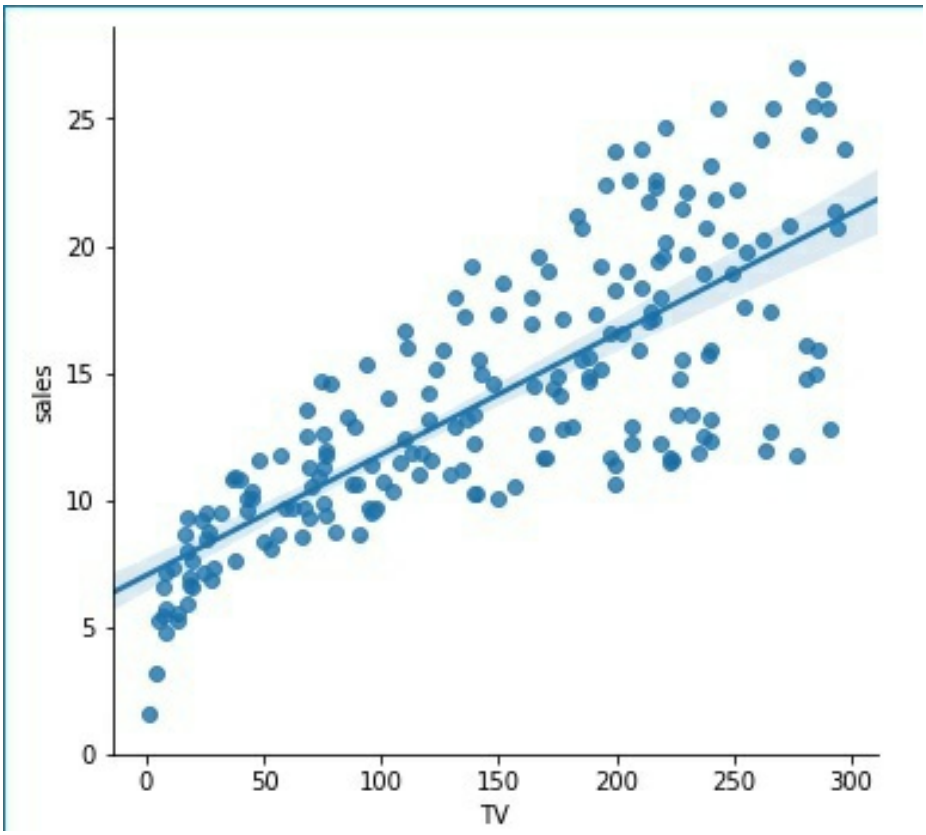
```
import seaborn as sns

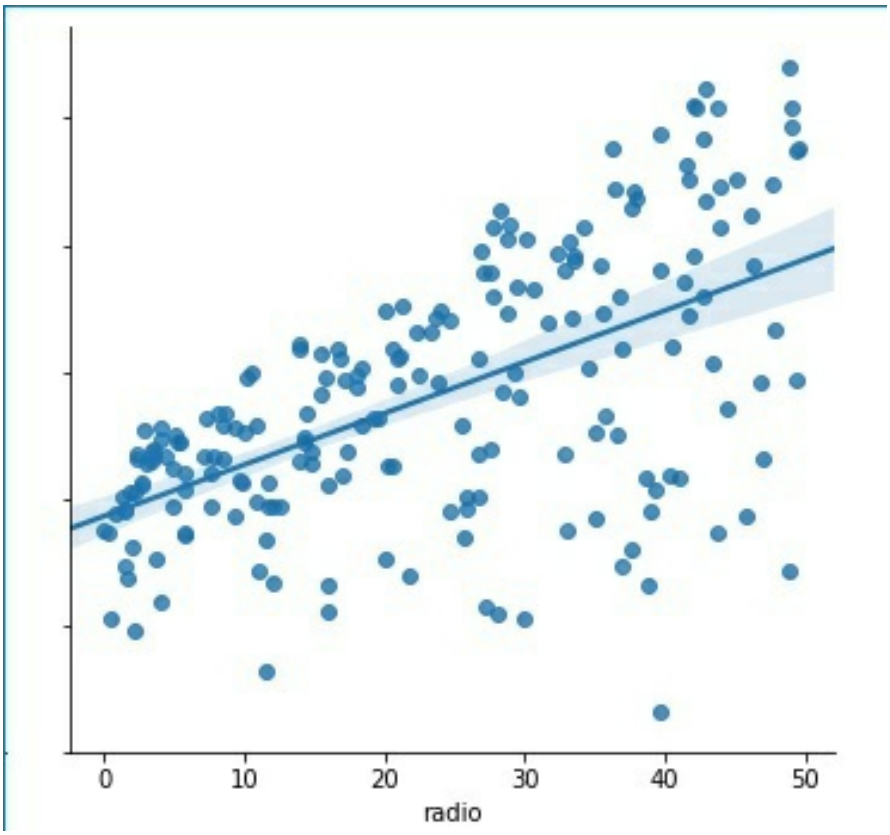
```

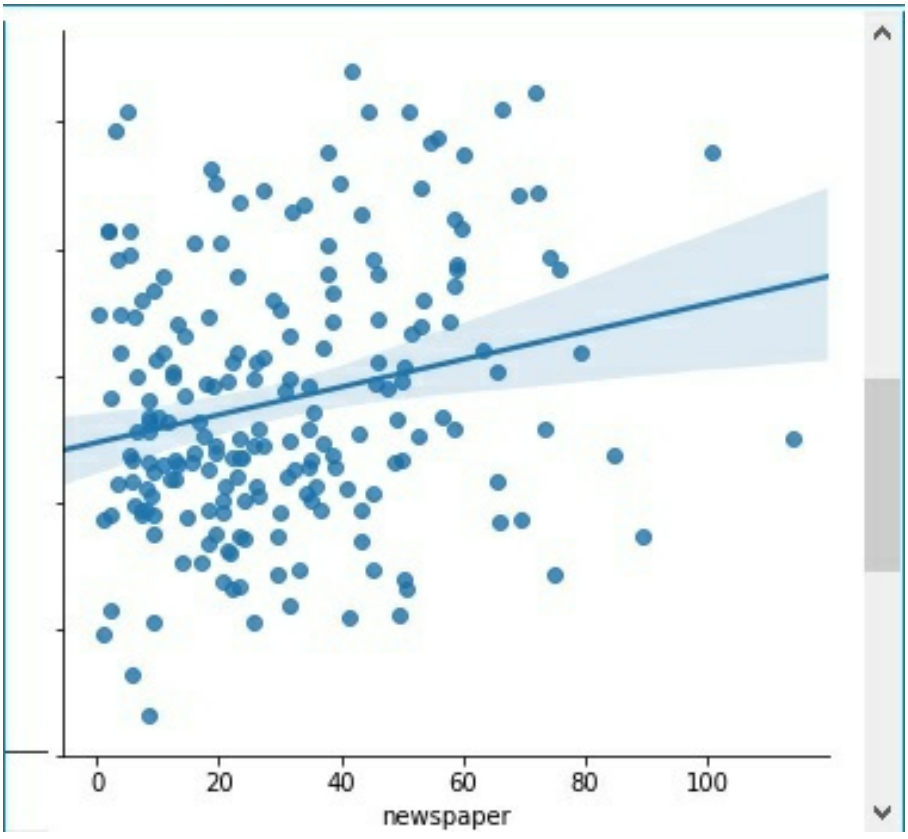
Dando sequência realizamos a importação da biblioteca Seaborn, muito utilizada em alternativa a MatPlotLib, de finalidade parecida, ou seja, exibir os dados de forma visual em forma de gráficos.

```
sns = sns.pairplot(base,
                   x_vars = ['TV', 'radio', 'newspaper'],
                   y_vars = 'sales',
                   size = 5,
                   kind = 'reg')
```

Em seguida criamos nossa variável sns, que por sua vez recebe como atributo a função sns.pairplot( ), onde passamos como parâmetros todo conteúdo de base, x_vars recebe os atributos previsores (os que serão plotados no eixo X), y_vars que recebe os dados de saída de sales, size = 5 simplesmente para visualização mais clara dos gráficos e por fim kind = ‘reg’, que define que o tipo de apresentação dos dados no gráfico é o tipo de dados de regressão.




0
10
20
30
40
50
radio


0
20
40
60
80
100
newspaper

Analisando rapidamente os gráficos podemos encontrar a relação entre cada atributo previsoes e o valor de vendas, importante salientar que aqui por hora, o que existe de mais relevante nesses gráficos é a angulação da linha gerada, uma linha ascendente confirma que houve retorno positivo de acordo com o investimento, pode acontecer de existir linhas descendentes em um ou mais gráficos de amostras, representando prejuízo em relação ao investimento. Nesse exemplo, ambos os 3 previsores (veículos de mídia) retornaram estimativas positivas, de lucro com base em seus dados.

```
17 from sklearn.model_selection import train_test_split
18
19 etreino, eteste, streino, steste = train_test_split(entradas,
20                                                     saidas,
21                                                     test_size = 0.3)
```

Tendo as primeiras estimativas de resultado, hora de aplicarmos alguns testes para avaliar a performance de nosso modelo. Como de costume, o processo se inicia importando o que for necessário, aqui, usaremos inicialmente a ferramenta train_test_split. Logo após criamos as variáveis dedicadas a treino e teste do modelo, bem como a definição de que 30% das amostras serão reservadas para teste, sobrando 70% para treino.

Nome	Tipo	Tamanho	Valor
base	DataFrame	(200, 5)	Column names: Unnamed: 0, TV, radio, newspaper, s
entradas	DataFrame	(200, 3)	Column names: TV, radio, newspaper
eteste	DataFrame	(60, 3)	Column names: TV, radio, newspaper
etreino	DataFrame	(140, 3)	Column names: TV, radio, newspaper
saidas	Series	(200,)	Series object of pandas.core.series module
steste	Series	(60,)	Series object of pandas.core.series module
streino	Series	(140,)	Series object of pandas.core.series module

Dando uma conferida no nosso explorador de variáveis podemos ver que de fato foram criadas tais variáveis.

```
23 from sklearn.linear_model import LinearRegression
24
```

Em seguida realizamos a importação da ferramenta LinearRegression, do módulo linear_model, da biblioteca sklearn.

```
25 reglinear = LinearRegression()
26 reglinear.fit(etreino,streino)
```

A seguir criamos nossa variável de nome reglinear que inicialmente apenas inicializa a ferramenta LinearRegression, sem parâmetros mesmo. Depois aplicamos sobre reglinear a função .fit( ) passando como parâmetro os dados de etreino e streino.

```
58 print(list(zip(['TV', 'radio', 'newspaper'], reglinear.coef_)))
59
```

Uma vez executado o modelo de regressão linear podemos agora aplicar alguns testes. Primeiramente instanciamos o conteúdo de nossos atributos previsores, seguido do argumento reglinear.coef_ que em outras palavras, retornará o valor de coeficiente (de investimento nesse caso).

```
In [8]: print(list(zip(['TV', 'radio', 'newspaper'], reglinear.coef_)))
[('TV', 0.0473674298724549), ('radio', 0.1853722490601593), ('newspaper',
-0.00238463121485491)]
```

Selecionando e executando o bloco de código anterior teremos um retorno padrão via console, onde podemos ver que existem coeficientes para cada um de nossos atributos previsores, estes coeficientes para entendimento mais fácil podem ser interpretados como valores de unidade monetária.

Por exemplo em ‘TV’ temos 0.04, o que significa que para cada 1 dólar investido, houve um aumento nas vendas de 4%, seguindo o mesmo raciocínio lógico, em ‘radio’ para cada dólar investido houve um aumento nas vendas de 18% e por fim em ‘newspaper’ para cada dólar investido houve um valor nulo, você até poderia em outras situações considerar 0.002 centavos de dólar de prejuízo, aqui nosso parâmetro é de apenas duas casas decimais mesmo, sendo que para cada 1 dólar investido não houve nem lucro nem prejuízo nas vendas.

```
30 print(reglinear.predict([[230.1,37.8,69.2]]))
31
```

Por meio da função .predict( ) podemos passar em forma de lista uma das linhas de nossa base de dados, contendo os três atributos previsores e realizar teste sobre essa amostra.

```
In [ ]: print(reglinear.predict([[230.1,37.8,69.2]]))
[20.64299359]
```

Usando esses parâmetros como exemplo, numa campanha onde foi

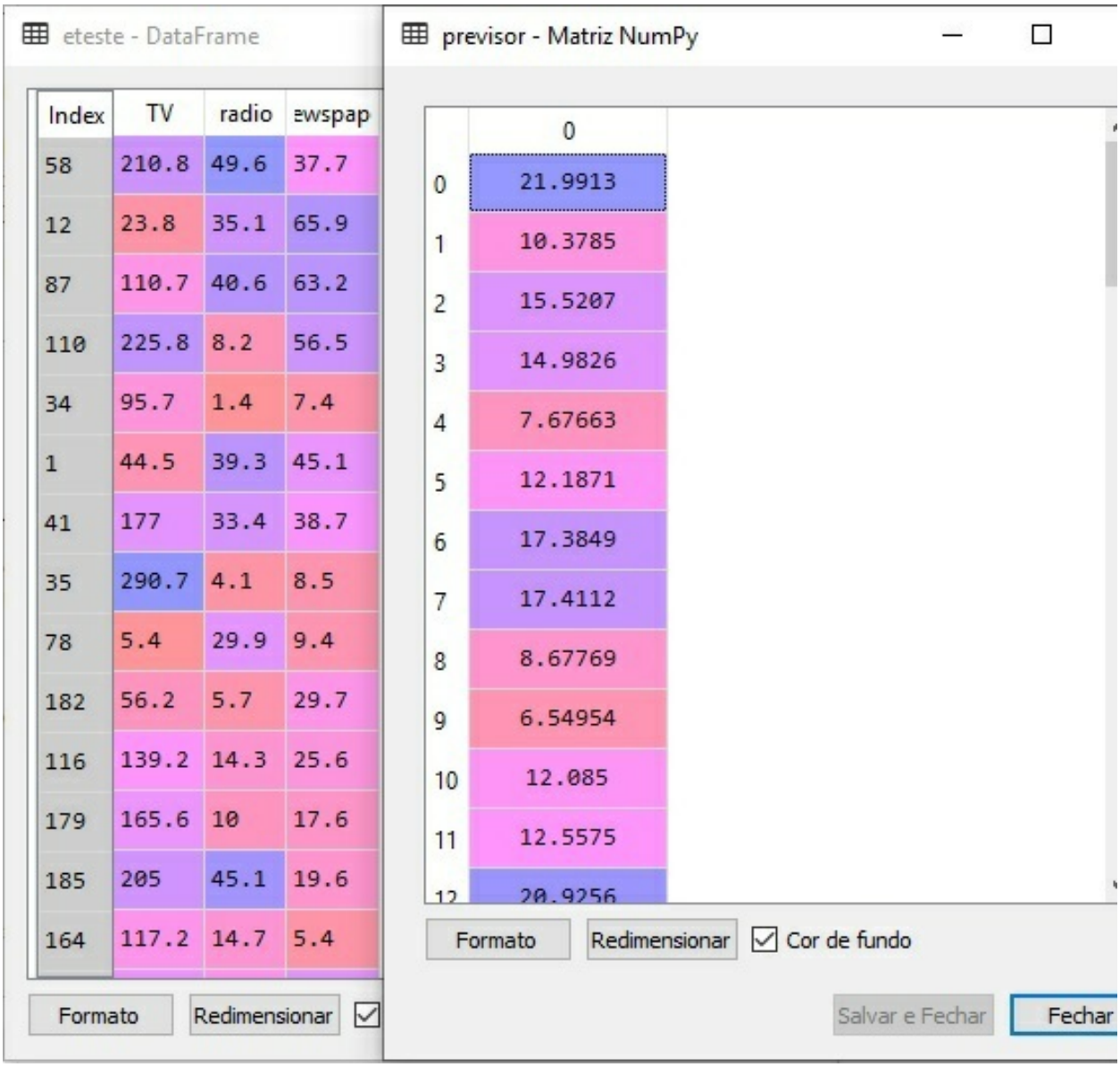
investido 230.1 dólares em publicidade via TV, 37.8 dólares em publicidade via rádio e 69.2 dólares em publicidade via jornal, houve um aumento nas vendas em geral de 20.6%. Repare que o interessante desse teste em particular é justamente equiparar quanto foi investido para cada mídia e qual foi o retorno, para justamente realizar ajustes e focar nas próximas campanhas na mídia que deu maior retorno.

```
32 previsor = reglinear.predict(eteste)
33 print(previsor)
```

Como já fizemos algumas vezes em outros modelos, podemos criar nossa variável de nome previsor que aplica a função .predict( ) sobre toda a nossa base separada em eteste.

```
In [  ]: previsor = reglinear.predict(eteste)
    ...: print(previsor)
[21.99131061 10.37845701 15.52067254 14.98257988  7.67663135 12.18712659
 17.38487641 17.41116216  8.67769227  6.54954127 12.08501628 12.55749281
 20.92556622 11.16525127 14.45488414  9.75249818 13.91969088  8.3366973
 10.0741478  16.3781017  10.36147754  9.49902928 11.32164028 15.50715257
 21.93928981  8.15167538 19.28286002 15.46912896  9.42837231 15.95689331
  4.40108511  6.05415038 17.91265432 16.92633817  9.87808944 21.33738647
 13.61495178  9.69074923  5.75556433 17.04488883 14.45251759 19.1892124
 15.89857622  8.94867862 14.30579785  8.83091711  7.71232706 14.07316039
 15.07185971 18.57963053 15.29149013 10.18037182  8.35771214 21.243932
  7.37650581 18.68617391 16.36841081 11.64848861  7.89186185 12.82134607]
```

Selecionado e executado o bloco de código anterior, os resultados são exibidos em console, assim como atribuídos a nossa variável previsor.



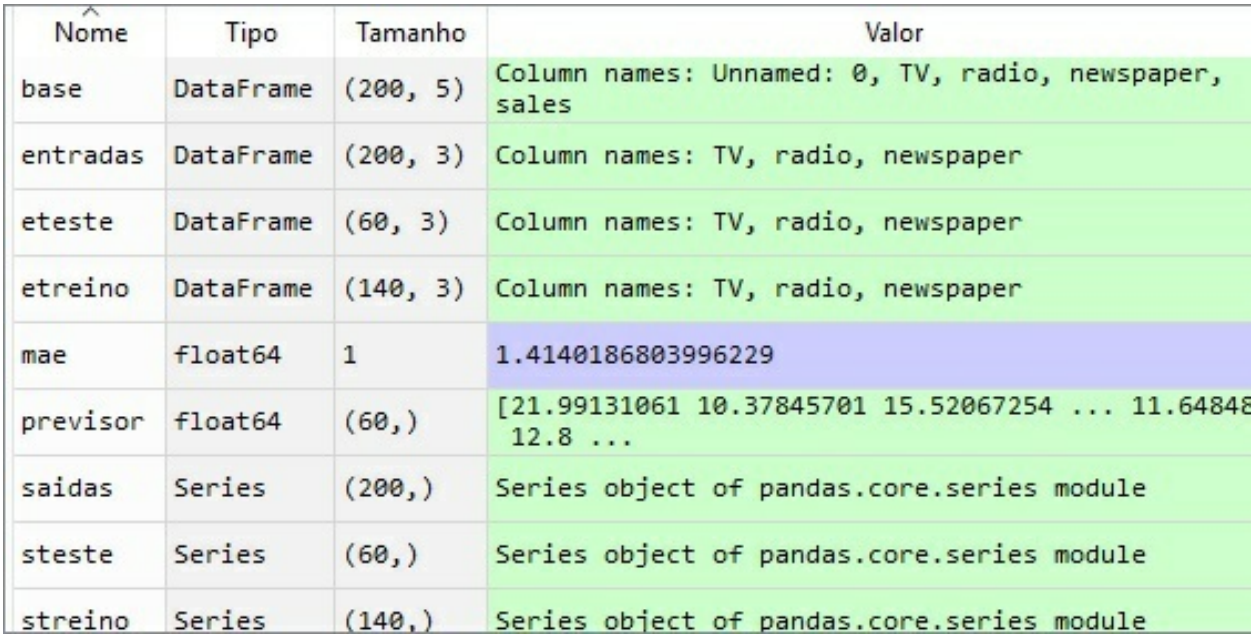
Abrindo as devidas variáveis via explorador de variáveis podemos fazer uma fácil relação, onde dentro de um período, os valores investidos em campanhas publicitárias surtiram um certo percentual de aumento nas vendas em geral.

```
35 from sklearn import metrics
36
```

Podemos ainda realizar alguns testes para confirmar a integridade de nossos dados. Aqui usaremos da ferramenta metrics da biblioteca sklearn para tal fim.

```
37 mae = metrics.mean_absolute_error(steste,previsor)
38
```

Primeiro teste será realizado por meio da função .mean_absolute_error( ) que com base nos dados de steste e dos obtidos em previsor, gerará um valor atribuído a nossa variável mae.



Nome	Tipo	Tamanho	Valor
base	DataFrame	(200, 5)	Column names: Unnamed: 0, TV, radio, newspaper, sales
entradas	DataFrame	(200, 3)	Column names: TV, radio, newspaper
eteste	DataFrame	(60, 3)	Column names: TV, radio, newspaper
etreino	DataFrame	(140, 3)	Column names: TV, radio, newspaper
mae	float64	1	1.4140186803996229
previsor	float64	(60,)	[21.99131061 10.37845701 15.52067254 ... 11.64848 12.8 ...
saidas	Series	(200,)	Series object of pandas.core.series module
steste	Series	(60,)	Series object of pandas.core.series module
streino	Series	(140,)	Series object of pandas.core.series module

Se não houve nenhum erro de sintaxe até o momento será criada a variável mae, note que ela possui o valor 1.41, que em outras palavras significa que para cada 1 dólar investido, houve um retorno de $1.41.

```
39 mse = metrics.mean_squared_error(steste,previsor)
40
```

Em seguida nos mesmos moldes aplicamos a função .mean_squared_error( ) que por sua vez fará o processamento dos dados sobre steste e previsor, porém com a particularidade que aqui os valores de erro encontrados durante o processamento são elevados ao quadrado, dando mais peso aos erros.

Raciocine que por exemplo um erro 2 elevado ao quadrado é apenas 4, enquanto um erro 5 ao quadrado é 25, o que gera grande impacto sobre as funções aplicadas.

Nome	Tipo	Tamanho	Valor
base	DataFrame	(200, 5)	Column names: Unnamed: 0, TV, radio, newspaper, sales
entradas	DataFrame	(200, 3)	Column names: TV, radio, newspaper
eteste	DataFrame	(60, 3)	Column names: TV, radio, newspaper
etreino	DataFrame	(140, 3)	Column names: TV, radio, newspaper
mae	float64	1	1.4140186803996229
mse	float64	1	3.2825652471496274
previsor	float64	(60,)	[21.99131061 10.37845701 15.52067254 ... 11.64848 12.8 ...
saidas	Series	(200,)	Series object of pandas.core.series module
steste	Series	(60,)	Series object of pandas.core.series module

Importante salientar que esse valor obtido inicialmente para mse não pode ser convertido diretamente para valor monetário. É necessário aplicar uma segunda função para este fim.

```
41 import numpy as np
42
43 rmse = np.sqrt(metrics.mean_squared_error(steste,previsor))
```

Para isso importamos a biblioteca numpy, na sequência criamos uma variável de nome rmse que aplica a raiz quadrada sobre mse, gerando agora um valor que pode ser levado em conta como unidade monetária.

Nome	Tipo	Tamanho	Valor
base	DataFrame	(200, 5)	Column names: Unnamed: 0, TV, radio, newspaper, sales
entradas	DataFrame	(200, 3)	Column names: TV, radio, newspaper
eteste	DataFrame	(60, 3)	Column names: TV, radio, newspaper
etreino	DataFrame	(140, 3)	Column names: TV, radio, newspaper
mae	float64	1	1.4140186803996229
mse	float64	1	3.2825652471496274
previsor	float64	(60,)	[21.99131061 10.37845701 15.52067254 ... 11.64848 12.8 ...
rmse	float64	1	1.811785099604704
saidas	Series	(200,)	Series object of pandas.core.series module

Note que agora podemos fazer as devidas comparações. Inicialmente havíamos descoberto que para cada 1 dólar investido, o retorno teria sido de $1.41, agora, dando mais peso aos erros de processamento, chegamos ao dado de que para cada 1 dólar investido, o retorno foi de $1.81, o que condiz perfeitamente com os dados plotados lá no início em nossos gráficos.

Código Completo:

```
import pandas as pd
import seaborn as sns
from sklearn.linear_model import LinearRegression
from sklearn import metrics
import numpy as np

base = pd.read_csv('Advertising.csv')
entradas = base[['TV', 'radio', 'newspaper']]
saidas = base['sales']

sns = sns.pairplot(base,
                   x_vars = ['TV', 'radio', 'newspaper'],
                   y_vars = 'sales',
                   size = 5,
                   kind = 'reg')

from sklearn.model_selection import train_test_split

etreino, eteste, streino, steste = train_test_split(entradas,
                                                    saidas,
                                                    test_size = 0.3)
reglinear = LinearRegression()
reglinear.fit(etreino,streino)

print(list(zip(['TV', 'radio', 'newspaper'], reglinear.coef_)))
print(reglinear.predict([[230.1,37.8,69.2]]))

previsor = reglinear.predict(eteste)
print(previsor)

mae = metrics.mean_absolute_error(steste,previsor)
mse = metrics.mean_squared_error(steste,previsor)
rmse = np.sqrt(metrics.mean_squared_error(steste,previsor))
```

REDES NEURAIS ARTIFICIAIS CONVOLUCIONAIS

Uma das aplicações mais comuns dentro de machine learning é o processamento de imagens. Na verdade, esta é uma área da computação que tem evoluído a passos largos justamente pelos avanços e inovações que redes neurais artificiais trouxeram para esse meio. O processamento de imagens que se deu início agrupando pixels em posições específicas em tela hoje evoluiu para modelos de visão computacional e até mesmo geração de imagens inteiramente novas a partir de dados brutos de características de imagens usadas para aprendizado.

O fato de haver um capítulo inteiro dedicado a este tipo de aprendizado de máquina se dá porque da mesma forma que anteriormente classificamos dados e extraímos informações a partir dos mesmos via redes neurais artificiais, aqui usaremos de modelos de redes neurais que serão adaptados desde o reconhecimento de dígitos e caracteres escritos à mão até modelos capazes de aprender características de um animal e gerar imagens completamente novas a partir de tais características “do que é um cachorro”.

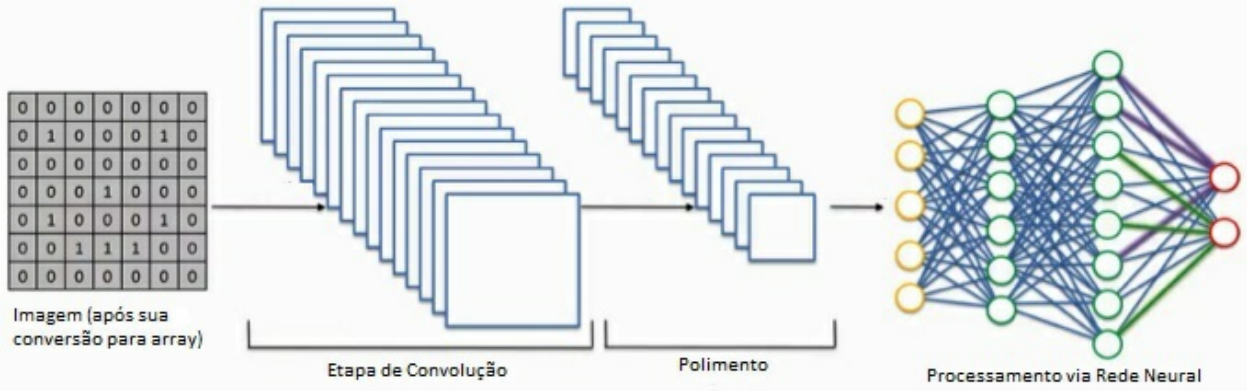
Assim como nos capítulos anteriores, estaremos aprendendo de forma procedural, nos primeiros exemplos contando com bastante referencial teórico para cada linha de código e à medida que você for criando sua bagagem de conhecimento estaremos mais focados a prática.

Como no capítulo anterior, iniciaremos nosso entendimento com base em um modelo que não necessariamente usa de uma rede neural artificial densa para seu processamento, inicialmente temos que fazer uma abordagem que elucide o processo de identificação de uma imagem assim como sua conversão para dados interpretados por um perceptron.

Raciocine que nos próximos exemplos sim estaremos trabalhando com modelos de redes neurais convolucionais, redes muito utilizadas na chamada visão computacional, onde se criam modelos capazes de detectar e reconhecer objetos. Uma das características importantes das redes ditas convolucionais é que apesar de poder haver ou não uma fase de treinamento manual da rede, elas são capazes de aprender sozinhas, reforçando padrões

encontrados e a partir desses realizar as conversões de imagem-matriz e matriz-imagem com base em mapeamento de pixels e seus respectivos valores e padrões.

Tenha em mente que aqui abordaremos de forma simples e prática uma das áreas de maior complexidade dentro da computação. Como de costume, abordaremos tudo o que for necessário para entendimento de nossos modelos, porém existe muito a se explorar com base na documentação das ferramentas que usaremos. De forma simples adaptaremos os conceitos aprendidos até então de uma rede neural para operadores de convolução, suas mecânicas de alimentação, conversão e processamento de dados assim como a prática da execução da rede neural convolucional em si.



O processo de processamento de dados oriundos de imagens se dá por uma série de etapas onde ocorrem diversas conversões entre os dados até o formato que finalmente possa ser processado via rede neural. Tenha em mente que a imagem que aparece para o usuário, internamente é uma espécie de mapa de pixels onde é guardado dados de informação como seu aspecto, coloração, intensidade de coloração, posição na grade de resolução, etc...

Sendo assim, ainda existe uma segunda etapa de processamento para que tais informações sejam transformadas em padrões e formatos interpretados dentro de uma rede neural para que seja feito o cruzamento desses dados e aplicação das funções necessárias.

De forma resumida, por hora, raciocine que uma imagem é lida a partir de algum dispositivo, de imediato serão feitas as conversões pixel a pixel para um mapa, como em um processo de indexação interna, para que tais dados possam ser convertidos para tipo e formato que possa ser processado via rede neural.

Reconhecimento de Caracteres – Digits Dataset

Para darmos início a esta parte dos estudos de redes neurais, iremos fazer o uso de uma base de dados numéricos com 1797 amostras divididas dentro do intervalo de 0 a 9. A ideia é que vamos treinar nosso modelo a aprender as características de cada número e posteriormente conseguir fazer o reconhecimento de um número escrito a mão para teste. Este é um exemplo bastante básico para começarmos a dar início do entendimento de como um computador interpreta imagens para seu processamento.

Nesse exemplo em particular vamos testar dois modelos de processamento para identificação de imagens, pois simularemos aqui um erro bastante comum nesse meio que se dá quando temos que fazer identificação a partir de uma base de dados muito pequena, muito imprecisa que é a execução do modelo confiando em uma margem de precisão relativamente baixa.



Como sempre, damos início ao processo criando um novo arquivo, nesse caso de nome Digits Dataset.py, em seguida realizamos as primeiras importações necessárias, por hora, importaremos o módulo datasets da biblioteca sklearn. Assim você pode deduzir que inicialmente estaremos usando uma biblioteca de exemplo para darmos os primeiros passos nessa área.

```
3 base = datasets.load_digits()
4 entradas = base.data
5 saidas = base.target
```

Logo após criamos nossa variável de nome base, que por sua vez recebe todos os dados do digits dataset por meio da função .load_digits( ) do módulo datasets. Em seguida criamos nossa variável entradas que recebe os atributos previsores por meio da instância base.data, da mesma forma criamos nossa variável saidas que recebe os dados alvo instanciando base.targets.

Nome	Tipo	Tamanho	Valor
base	utils.Bunch	5	Bunch object of sklearn.utils module
entradas	float64	(1797, 64)	[[0. 0. 5. ... 0. 0. 0.] [0. 0. 0. ... 10. 0. 0.]
saidas	int32	(1797,)	[0 1 2 ... 8 9 8]

Se o processo de importação ocorreu como esperado foram criadas as devidas variáveis, note que aqui temos a mesma quantidade de dados para nossas entradas e saídas, uma vez que estaremos fazendo o processo de aprendizagem de máquina para que nosso interpretador aprenda a identificar e classificar números escritos a mão. Importante salientar que este número de 1797 amostras é muito pouco quando o assunto é classificação de imagens, o que acarretará em problemas de imprecisão de nosso modelo quando formos executar alguns testes.

O valor 64, da segunda coluna de nossa variável entradas diz respeito ao formato em que tais dados estão dispostos, aqui nesse dataset temos 1797 imagens que estão em formato 8x8 pixels, resultando em 64 dados em linhas e colunas que são referências a uma escala de preto e branco.

```
7 print(base.data[0])
8
```

Para ficar mais claro, por meio da função print( ) passando como parâmetro a amostra de número 0 de nossa base de dados, poderemos ver via console tal formato.

```
In [ ]: print(base.data[0])
[ 0.  0.  5. 13.  9.  1.  0.  0.  0.  0. 13. 15. 10. 15.  5.  0.  0.  3.
 15.  2.  0. 11.  8.  0.  0.  4. 12.  0.  0.  8.  8.  0.  0.  5.  8.  0.
  0.  9.  8.  0.  0.  4. 11.  0.  1. 12.  7.  0.  0.  2. 14.  5. 10. 12.
  0.  0.  0.  0.  6. 13. 10.  0.  0.  0.]
```

Via console podemos finalmente visualizar tal formato. Cada valor é referente a uma posição dessa matriz 8x8, onde os valores estão dispostos em um intervalo de 0 até 16, onde 0 seria branco e 16 preto, com valores intermediários nessa escala para cada pixel que compõe essa imagem de um número escrito a mão.

```
 9 print(base.images[0])
10
```

Da mesma forma podemos visualizar o formato de imagem salvo na

própria base de dados. O método é muito parecido com o anterior, porém instanciamos a amostra número 0 de images.

```
In [ ]: print(base.images[0])
[[ 0.  0.  5. 13.  9.  1.  0.  0.]
 [ 0.  0. 13. 15. 10. 15.  5.  0.]
 [ 0.  3. 15.  2.  0. 11.  8.  0.]
 [ 0.  4. 12.  0.  0.  8.  8.  0.]
 [ 0.  5.  8.  0.  0.  9.  8.  0.]
 [ 0.  4. 11.  0.  1. 12.  7.  0.]
 [ 0.  2. 14.  5. 10. 12.  0.  0.]
 [ 0.  0.  6. 13. 10.  0.  0.  0.]]
```

Via console podemos ver uma matriz 8x8 onde inclusive com o olhar um pouco mais atento podemos identificar que número é esse, referente a primeira amostra da base de dados.

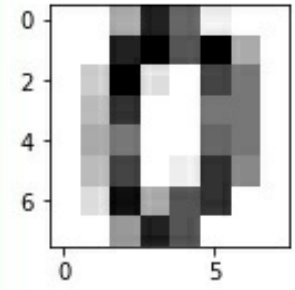
```
11 import matplotlib.pyplot as plt
12
```

Partindo para o que interessa, vamos fazer a plotagem deste número. Para isto, como de praxe, importamos a matplotlib.pyplot.

```
13 plt.figure(figsize = (2,2))
14 plt.imshow(base.images[0],
15            cmap = plt.cm.gray_r)
```

Em seguida, usando apenas o necessário para exibir nossa amostra, criamos essa pequena estrutura onde a função .figure( ) fará a leitura e plotagem com um tamanho definido em 2x2 (lembrando que isso não é pixels, apenas uma referência métrica interna a matplotlib). Na sequência a função .imshow( ) fica responsável por exibir nossa amostra parametrizada também em uma escala de cinza.

```
Out[5]: <matplotlib.image.AxesImage at 0x26db21af668>
```



Não havendo nenhum erro de sintaxe é finalmente exibido em nosso console o número 0, referente a primeira amostra de nossa base de

dados. Note a baixa qualidade da imagem, tanto a baixa qualidade quanto o número relativamente pequeno de amostras são um grande problema quando se diz respeito em classificação de imagens.

Revisando, toda imagem, internamente, para nosso sistema operacional e seus interpretadores, é um conjunto de referências a pixels e suas composições, aqui inicialmente estamos trabalhando com imagens de pequeno tamanho e em escala de cinza, mas independente disso, toda imagem possui um formato, a partir deste, uma matriz com dados de tonalidades de cinza (ou de vermelho, verde e azul no caso de imagens coloridas), que em algum momento será convertido para binário e código de máquina. A representação da imagem é uma abstração que o usuário vê em tela, desta disposição de pixels compondo uma figura qualquer.

```
17 from sklearn.model_selection import train_test_split
18 etreino,eteste,streino,steste = train_test_split(entradas,
19                                                  saidas,
20                                                  test_size = 0.1,
21                                                  random_state = 2)
```

Dando sequência, mesmo trabalhando sobre imagens podemos realizar aqueles testes de performance que já são velhos conhecidos nossos. Inicialmente importamos train_test_split, criamos suas respectivas variáveis, alimentamos a ferramenta com dados de entrada e de saída, assim como separamos as amostras em 90% para treino e 10% para teste.

Por fim, definimos que o método de separação das amostras seja randômico não pegando dados com proximidade igual ou menos a 2 números.

```
23 from sklearn import svm
24 classificador = svm.SVC()
25 classificador.fit(etreino,streino)
26 previsor = classificador.predict(eteste)
```

Separadas as amostras, hora de criar nosso modelo inicial de classificador, aqui neste exemplo inicialmente usaremos uma ferramenta muito usada quando o assunto é identificação e classificação de imagens chamada Suport Vector Machine, esta ferramenta está integrada na biblioteca sklearn e pode ser facilmente importada como importamos outras ferramentas em outros momentos.

Em seguida criamos nosso classificador que inicializa a ferramenta

svm.SVC( ), sem parâmetros mesmo, e codificada nessa sintaxe. Logo após por meio da função .fit( ) alimentamos nosso classificador com os dados de etreino e streino. Por fim, criamos nosso primeiro previsor, que com base nos dados encontrados em classificador fará as devidas comparações e previsões com os dados de eteste.

```
28 from sklearn import metrics
29 margem_acerto = metrics.accuracy_score(steste, previsor)
```

Criado o modelo, na sequência como de costume importamos metrics e criamos nossa variável responsável por fazer o cruzamento dos dados e o teste de performance do modelo. Então é criada a variável margem_acerto que por sua vez executa a ferramenta .accuracy_score( ) tendo como parâmetros os dados de steste e os encontrados em previsor.

Nome	Tipo	Tamanho	Valor
base	utils.Bunch	5	Bunch object of sklearn.utils module
entradas	float64	(1797, 64)	[[0. 0. 5. ... 0. 0. 0.] [0. 0. 0. ... 10. 0. 0.]
eteste	float64	(180, 64)	[[0. 0. 0. ... 0. 0. 0.] [0. 0. 1. ... 15. 1. 0.]
etreino	float64	(1617, 64)	[[0. 0. 10. ... 0. 0. 0.] [0. 0. 0. ... 16. 9. 0.]
margem_acerto	float64	1	0.5611111111111111
previsor	int32	(180,)	[4 5 9 ... 5 0 5]
saidas	int32	(1797,)	[0 1 2 ... 8 9 8]
steste	int32	(180,)	[4 0 9 ... 0 0 4]
streino	int32	(1617,)	[0 6 5 ... 1 1 5]

Selecionado e executado todo bloco de código anterior, via explorador de variáveis podemos nos ater aos dados apresentados nesses testes iniciais. O mais importante deles por hora, margem_acerto encontrou em seu processamento uma taxa de precisão de apenas 56%. Lembre-se que comentei em parágrafos anteriores, isto se dá por dois motivos, base de dados pequena e imagens de baixa qualidade.

De imediato o que isto significa? Nossos resultados não são nenhum pouco confiáveis nesse modelo, até podemos realizar as previsões, porém será interessante repetir os testes algumas vezes como prova real,

como viés de confirmação das respostas encontradas dentro dessa margem de acertos relativamente baixa.

Leitura e reconhecimento de uma imagem.

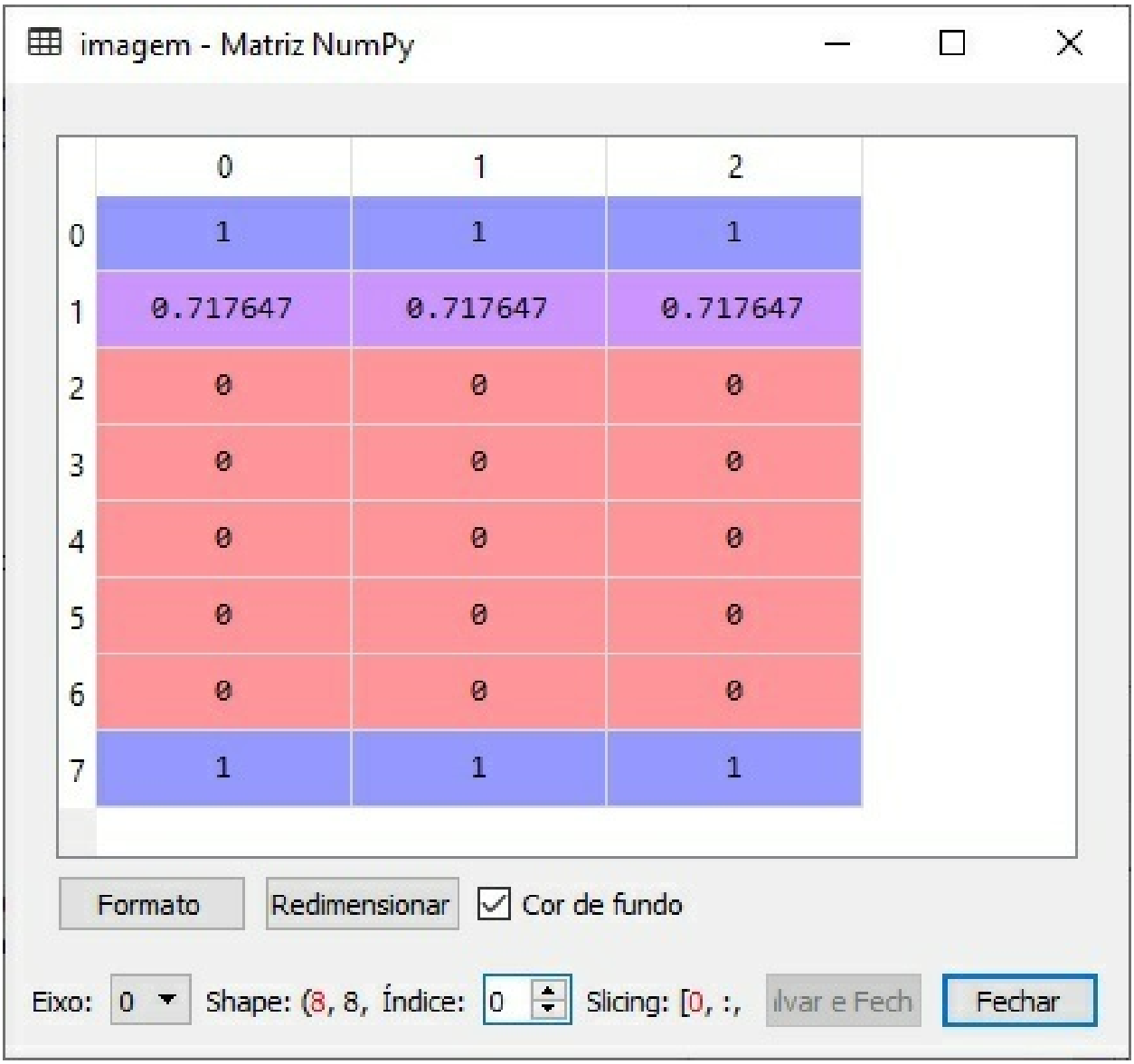
```
31 import numpy as np
32 import matplotlib.image as mimg
```

Uma vez criado o modelo, mesmo identificada sua baixa precisão, podemos realizar testes reais para o mesmo, fazendo a leitura de um número escrito a mão a partir de uma imagem importada para nosso código. Como sabemos o número em questão esse processo pode ser considerado válido como teste, até mesmo com a nossa atual margem de acerto.

Anteriormente havíamos importado a ferramenta pyplot da matplotlib, dessa vez a ferramenta que usaremos é a image, da própria matplotlib, o processo de importação é exatamente igual ao de costume, dessa vez referenciamos a ferramenta como mimg. Por fim, note que também importamos a biblioteca numpy.

```
34 imagem = mimg.imread('num2.png')
35 print(imagem)
```

Dando sequência, criamos nossa variável de nome imagem que recebe como atributo o conteúdo importado de num2.png por meio da função .imread( ). A seguir por meio do comando print( ) imprimimos imagem em nosso console.



Via explorador de variáveis podemos abrir a variável imagem e seu conteúdo, muita atenção ao rodapé da janela, uma vez que estamos visualizando aqui uma matriz, estes dados dispostos nessa grade 8x8 é apenas a informação de 1 pixel da imagem, por meio dos botões da parte inferior da janela podemos navegar pixel a pixel.

```
    ...: print(imagem)
[[[1.         1.         1.        ]
  [0.7176471  0.7176471  0.7176471 ]
  [0.         0.         0.        ]
  [0.         0.         0.        ]
  [0.         0.         0.        ]
  [0.         0.         0.        ]
  [0.         0.         0.        ]
  [1.         1.         1.        ]]

 [[1.         1.         1.        ]
  [0.         0.         0.        ]
  [1.         1.         1.        ]
  [1.         1.         1.        ]
```

Via console podemos ver em forma de lista todos dados também referentes a cada pixel.

Raciocine que por hora estes formatos de dados nos impedem de aplicar qualquer tipo de função sobre os mesmos, sendo assim, precisamos fazer a devida conversão desses dados referentes a cada pixel para uma matriz que por sua vez contenha dados em forma de uma matriz, mais especificamente uma array do tipo numpy.

```
37 def rgb2gray(rgb):
38     img_array = np.dot(rgb[...,:3],[0.299,0.587,0.114])
39     img_array = (16 - (img_array * 16)).astype(int)
40     img_array = img_array.flatten()
41     return img_array
```

Poderíamos criar uma função do zero que fizesse tal processo, porém vale lembrar que estamos trabalhando com Python, uma linguagem com uma enorme comunidade de desenvolvedores, em função disso com apenas uma rápida pesquisa no Google podemos encontrar uma função já pronta que atende nossos requisitos.

Analisando a função em si, note que inicialmente é criada a função rgb2gray( ) que recebe como parâmetro a variável temporária rgb, uma vez que essa será substituída pelos arquivos que faremos a leitura e a devida identificação. Internamente temos uma variável de nome img_array, inicialmente ela recebe o produto escalar de todas as linhas e das 3 colunas do formato anterior (matriz de pixel) sobre os valores 0.299, 0.587 e 0.114, uma convenção quando o assunto é conversão para rgb (escala colorida red grey blue).

Em seguida é feita a conversão desse valor resultado para tipo inteiro por meio da função .astype( ) parametrizada com int. Logo após é aplicada a função .flatten( ) que por sua vez irá converter a matriz atual por uma nova indexada no formato que desejamos, 64 valores em um intervalo entre 0 e 16 (escala de cinza) para que possamos aplicar o devido processamento que faremos a seguir.

```
43 rgb2gray(imagem)
44
```

Para ficar mais claro o que essa função faz, chamando-a e passando como parâmetro nossa variável imagem (que contém o arquivo lido anteriormente num2.png) podemos ver o resultado de todas conversões em nosso console.

```
In [  ]: rgb2gray(imagem)
Out[  ]:
array([ 0,  4, 16, 16, 16, 16, 16,  0,  0, 16,  0,  0,  0,  0,  0,  0,  0,
       16,  0,  0,  0,  0,  0,  0,  0,  7, 16, 16, 16,  6,  0,  0,  0,  0,
        0,  0,  0, 16,  0,  0,  0, 16,  0,  0,  0, 16, 16,  0,  0,  0, 16,
       16, 16, 16,  0,  0,  0,  0,  0, 16, 16,  0,  0,  0])
```

Aplicadas as conversões sobre num2.png que estava instanciada em nossa variável imagem, podemos notar que temos uma array com 64 referências de tonalidades de cinza em uma posição organizada para o devido reconhecimento.

```
45 identificador = svm.SVC()
46 identificador.fit(entradas,saidas)
47 previsor_id = identificador.predict([rgb2gray(imagem)])
48 print(previsor_id)
```

Criada toda estrutura inicial de nosso modelo, responsável por fazer até então a leitura e a conversão do arquivo de imagem para que seja feito o reconhecimento de caractere, hora de finalmente realizar os primeiros testes de reconhecimento e identificação. Aqui, inicialmente usaremos uma ferramenta chamada Suport Vector Machine.

Esta ferramenta por sua vez usa uma série de métricas de processamento vetorial sobre os dados nela alimentados. Uma SVM possui a característica de por meio de aprendizado supervisionado (com base em treinamento via uma base de dados), aprender a reconhecer padrões e a partir destes realizar classificações.

Inicialmente criamos nossa variável de nome identificador, que por sua vez inicia o módulo SVC da ferramenta svm. Em seguida por meio da função .fit( ) é feita a alimentação da mesma com os dados de entrada e saída, lembrando que nesse modelo estamos tratando de atributos previsores e das devidas saídas, já que estamos treinando nossa máquina para reconhecimento de imagens.

Logo após criamos nossa variável de nome previsor_id, que por sua vez via função .predict( ) recebe nossa função rgb2gray( ) alimentada com nosso arquivo num2.png. Por fim, simplesmente imprimimos em nosso console o resultado obtido em previsor_id.

```
In [  ]: identificador = svm.SVC()
    ...: identificador.fit(entradas,saidas)
    ...: previsor_id = identificador.predict([rgb2gray(imagem)])
    ...: print(previsor_id)
C:\Users\Fernando\Anaconda3\lib\site-packages\sklearn\svm\base.py:193:
FutureWarning: The default value of gamma will change from 'auto' to 'scale' in
version 0.22 to account better for unscaled features. Set gamma explicitly to
'auto' or 'scale' to avoid this warning.
  "avoid this warning.", FutureWarning)
[5]
```

Selecionando e executando o bloco de código anterior podemos acompanhar via console o seu processamento e que, apesar de nossa margem de acerto relativamente baixa no processo de aprendizado, ele identificou corretamente que o número passado para identificação (nosso arquivo num2.png) era o número 5 escrito a mão.

Da mesma forma podemos realizar mais testes de leitura, desde que alimentando nosso código com imagens no mesmo padrão. Porém, para não nos estendermos muito nesse capítulo inicial, vamos realizar uma segunda verificação independente da SVM, para termos um segundo parâmetro de confirmação. Já sabemos que a leitura e identificação de nosso arquivo foi feita corretamente, mas caso houvesse ocorrido um erro de identificação esse segundo teste poderia nos ajudar de alguma forma.

Em outro momento já usamos do modelo de regressão logística para classificar dados quanto seu agrupamento e cruzamento de dados via matriz, aqui nossas conversões vão de um arquivo de imagem para uma array numpy, sendo possível aplicar o mesmo tipo de processamento.

```
50 from sklearn.linear_model import LogisticRegression
51 logr = LogisticRegression()
52 logr.fit(etreino,streino)
53 previsor_logr = logr.predict(eteste)
54 acerto_logr = metrics.accuracy_score(steste, previsor_logr)
55 print(acerto_logr)
```

Como não havíamos usado nada deste modelo em nosso código atual, realizamos a importação da ferramenta LogisticRegression do módulo linear_model da biblioteca sklearn.

Em seguida criamos uma variável logr que inicializa a ferramenta, na sequência alimentamos a ferramenta com os dados de etreino e streino, por fim criamos uma variável previsor_logr que faz as previsões sobre os dados de eteste.

Também criamos uma variável de nome acerto_logr que mostra a taxa de precisão cruzando os dados de steste e previsor_logr, por fim, exibindo no console este resultado.

Nome	Tipo	Tamanho	Valor
acerto_logr	float64	1	0.9277777777777778
base	utils.Bunch	5	Bunch object of sklearn.utils module
entradas	float64	(1797, 64)	[[0. 0. 5. ... 0. 0. 0.] [0. 0. 0. ... 10. 0. 0.]
eteste	float64	(180, 64)	[[0. 0. 0. ... 0. 0. 0.] [0. 0. 1. ... 15. 1. 0.]
etreino	float64	(1617, 64)	[[0. 0. 10. ... 0. 0. 0.] [0. 0. 0. ... 16. 9. 0.]
imagem	float32	(8, 8, 3)	[[[1. 1. 1.] [0.7176471 0.7176471 0.7176471] .. ..
margem_acerto	float64	1	0.5611111111111111
previsor	int32	(180,)	[4 5 9 ... 5 0 5]
previsor_id	int32	(1,)	[5]

Via explorador de variáveis podemos ver que o modelo de regressão logística, aplicado em nossa base de dados, obteve uma margem de acertos de 92%, muito maior aos 56% da SVM que em teoria deveria retornar melhores resultados por ser especializada em reconhecimento e identificação de padrões para imagens.

Novamente, vale lembrar que isso se dá porque temos uma base de

dados muito pequena e com imagens de baixa qualidade, usadas apenas para fins de exemplo, em aplicações reais, tais margens de acerto tendem a ser o oposto do aqui apresentado.

```
57 regressor = LogisticRegression()
58 regressor.fit(entradas,saidas)
59 previsor_regl = regressor.predict([rgb2gray(imagem)])
60 print(previsor_regl)
```

Para finalizar, criamos nosso regressor por meio de uma variável homônima que inicializa a ferramenta, na sequência é alimentada com os dados contidos em entradas e saidas. Também criamos a variável previsor_regl que realiza as devidas previsões com base na nossa função rgb2gray( ) alimentada com nosso arquivo instanciado num2.png. Concluindo o processo imprimimos em nosso console o resultado obtido para previsor_regl.

```
In [  ]: regressor = LogisticRegression()
    ...: regressor.fit(entradas,saidas)
    ...: previsor_regl = regressor.predict([rgb2gray(imagem)])
    ...: print(previsor_regl)
C:\Users\Fernando\Anaconda3\lib\site-packages\sklearn\linear_model\logistic.py:
432: FutureWarning: Default solver will be changed to 'lbfgs' in 0.22. Specify a
solver to silence this warning.
  FutureWarning)
C:\Users\Fernando\Anaconda3\lib\site-packages\sklearn\linear_model\logistic.py:
469: FutureWarning: Default multi_class will be changed to 'auto' in 0.22. Spec:
the multi_class option to silence this warning.
  "this warning.", FutureWarning)
[5]
```

Via console podemos acompanhar o processamento e como esperado, a previsão da identificação do arquivo, reconhecido como número 5 escrito a mão.

Código Completo:

```python
from sklearn import datasets
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn import svm
from sklearn import metrics
import numpy as np
import matplotlib.image as mimg
from sklearn.linear_model import LogisticRegression

base = datasets.load_digits()
entradas = base.data
saidas = base.target

plt.figure(figsize = (2,2))
plt.imshow(base.images[0],
           cmap = plt.cm.gray_r)

etreino,eteste,streino,steste = train_test_split(entradas,
                                                 saidas,
                                                 test_size = 0.1,
                                                 random_state = 2)
classificador = svm.SVC()
classificador.fit(etreino,streino)
previsor = classificador.predict(eteste)
margem_acerto = metrics.accuracy_score(steste, previsor)

imagem = mimg.imread('num2.png')

def rgb2gray(rgb):
    img_array = np.dot(rgb[...,:3],[0.299,0.587,0.114])
    img_array = (16 - (img_array * 16)).astype(int)
    img_array = img_array.flatten()
    return img_array

rgb2gray(imagem)

identificador = svm.SVC()
identificador.fit(entradas,saidas)
previsor_id = identificador.predict([rgb2gray(imagem)])
print(previsor_id)

logr = LogisticRegression()
logr.fit(etreino,streino)
previsor_logr = logr.predict(eteste)
acerto_logr = metrics.accuracy_score(steste, previsor_logr)
print(acerto_logr)

regressor = LogisticRegression()
regressor.fit(entradas,saidas)
previsor_regl = regressor.predict([rgb2gray(imagem)])
print(previsor_regl)
```

Classificação de Dígitos Manuscritos – MNIST Dataset

Dados os nossos passos iniciais nesta parte de identificação e reconhecimento de dígitos manuscritos, hora de fazer o mesmo tipo de processamento agora de forma mais robusta, via rede neural artificial. Desta vez estaremos usando o dataset MNIST, base de dados integrante da biblioteca Keras, com maiores amostras de imagem e imagens também armazenadas em uma resolução maior.

Lembre-se que no exemplo anterior nosso grande problema foi trabalhar sobre uma baixa margem de precisão devido os dados de nossa base que não colaboravam para um melhor processamento.

MNIST.py

```
1 import matplotlib.pyplot as plt
2 from keras.datasets import mnist
3 from keras.models import Sequential
4 from keras.layers import Dense, Flatten, Dropout
5 from keras.utils import np_utils
6 from keras.layers import Conv2D, MaxPooling2D
7 from keras.layers.normalization import BatchNormalization
```

Sem mais delongas vamos direto ao ponto, direto ao código. Como sempre, inicialmente criamos um novo arquivo de nome MNIST.py que armazenará nossos códigos, dentro de si começamos realizando as devidas importações.

Não muito diferente do que estamos habituados, realizamos a importação da matplotlib.pyplot (para plotagem/exibição das imagens via console), da base de dados mnist, das ferramentas Sequential, Dense, Flatten e Dropout de seus referidos módulos.

Também importamos np_utils (para nossas transformações de formato de matriz), em seguida importamos Conv2D (ferramenta pré-configurada para criação de camadas de rede neural adaptada a processamento de dados de imagem), MaxPooling2d (ferramenta que criará micromatrizes indexadas para identificação de padrões pixel a pixel) e por fim BatchNormalization (ferramenta que realiza uma série de processos internos para diminuir a margem de erro quando feita a leitura pixel a pixel

da imagem).

```
(etreino,streino),(eteste,steste) = mnist.load_data()
```

Na sequência criamos as variáveis que armazenarão os dados de entrada e de saída importados da base mnist.

Nome	Tipo	Tamanho	Valor
eteste	float32	(10000, 28, 28, 1)	[[[[0.] [0.]
etreino	float32	(60000, 28, 28, 1)	[[[[0.] [0.]
steste	float32	(10000, 10)	[[0. 0. 0. ... 1. 0. 0.] [0. 0. 1. ... 0. 0. 0.]
streino	float32	(60000, 10)	[[0. 0. 0. ... 0. 0. 0.] [1. 0. 0. ... 0. 0. 0.]

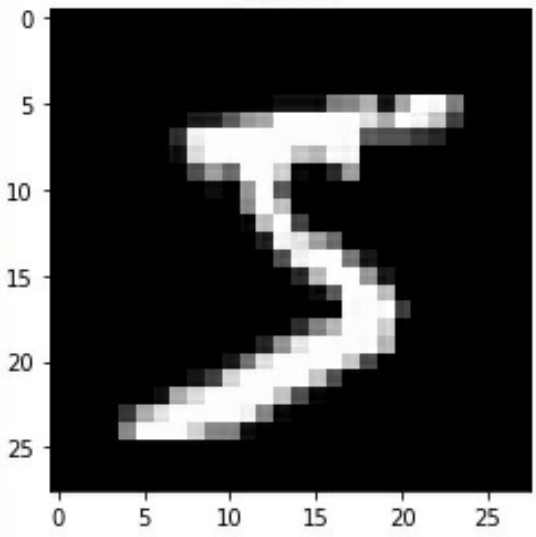
Fazendo uma rápida leitura via explorador de variáveis podemos ver que de fato nossas variáveis foram criadas. Note que imediatamente podemos deduzir o tamanho dessa base de dados, uma vez que temos 60000 amostras separadas para treino e 10000 separadas para teste. Um número consideravelmente maior do que a base usada no exemplo anterior (1797 amostras apenas).

Repare nos dados de entrada que eles possuem uma configuração adicional (28,28,1), que significa que cada amostra está numa configuração de matrix 28x28 de uma camada de profundidade (pode existir processamento sobre voxels, onde haverão mais camadas de profundidade/volume).

```
plt.imshow(etreino[0], cmap = 'gray')
plt.title('Classe' + str(streino[0]))
```

Em seguida fazendo uma simples plotagem, nos mesmos moldes do modelo anterior, pegamos a amostra de índice número 0 de nossa base de entradas para treino, apenas por conformidade instanciamos a nomenclatura da amostra de índice número 0 de nossa base de saída de treino.

```
Out[2]: Text(0.5, 1.0, 'Classe5')
```



Selecionado e executado o bloco de código anterior, podemos ver via console que das amostras de saída categorizadas como “número 5”, a imagem apresentada é um número 5 que podemos deduzir pelo seu formato ter sido escrito à mão. Repare a qualidade da imagem, muito superior em relação a do exemplo anterior.

```
14 etreino = etreino.reshape(etreino.shape[0],28,28,1)
15 eteste = eteste.reshape(eteste.shape[0],28,28,1)
```

Iniciando a fase de polimento de nossos dados, o que faremos em cima de uma base de dados de imagens de números é converter essas informações de mapeamento de pixels para um formato array onde se possa ser aplicado funções. Sendo assim, criamos nossa variável de nome etreino que recebe sobre si a função .reshape( ), redimensionando todas amostras para o formato 28x28 de 1 camada. O mesmo é feito para eteste.

Nome	Tipo	Tamanho	Valor
eteste	uint8	(10000, 28, 28, 1)	[[[[0] [0]
etreino	uint8	(60000, 28, 28, 1)	[[[[0] [0]
steste	uint8	(10000,)	[7 2 1 ... 4 5 6]
streino	uint8	(60000,)	[5 0 4 ... 5 6 8]

Repare que as variáveis etreino e eteste foram atualizadas para o

novo formato.

```
17 etreino = etreino.astype('float32')
18 eteste = eteste.astype('float32')
```

Em seguida realizamos uma nova alteração e atualização sobre etreino e eteste, dessa vez por meio da função .astype( ) convertemos o seu conteúdo para o tipo float32 (números com casas decimais).

```
20 etreino /= 255
21 eteste /= 255
```

Prosseguindo alteramos e atualizamos etreino e eteste mais uma vez, dessa vez dividindo o valor de cada amostra por 255.

```
23 streino = np_utils.to_categorical(streino, 10)
24 steste = np_utils.to_categorical(steste, 10)
```

Finalizada a formatação de nossos dados de entrada, hora de realizar o polimento sobre os dados de saída. Aqui simplesmente aplicamos a função .to_categorical( ) que altera e atualiza os dados em 10 categorias diferentes (números de 0 a 9).

Nome	Tipo	Tamanho	Valor
eteste	float32	(10000, 28, 28, 1)	[[[[0.] [0.]
etreino	float32	(60000, 28, 28, 1)	[[[[0.] [0.]
steste	float32	(10000, 10)	[[0. 0. 0. ... 1. 0. 0.] [0. 0. 1. ... 0. 0. 0.]
streino	float32	(60000, 10)	[[0. 0. 0. ... 0. 0. 0.] [1. 0. 0. ... 0. 0. 0.]

Feito o reajuste de nossos dados de saída temos agora dados matriciais compatíveis tanto para o cruzamento entre eles quanto para aplicação de funções. Note que terminamos com 60000 amostras que servirão para treinar nossa rede neural, realizaremos testes sobre 10000 amostras dessas, as categorizando em números de 0 a 9.

```
26 classificador = Sequential()
27 classificador.add(Conv2D(32,
28                         (3,3),
29                         input_shape = (28,28,1),
30                         activation = 'relu'))
31 classificador.add(BatchNormalization())
32 classificador.add(MaxPooling2D(pool_size = (2,2)))
33 classificador.add(Conv2D(32,
34                         (3,3),
35                         activation = 'relu'))
```

Agora damos início a criação da estrutura de nossa rede neural artificial convolucional, você verá que o modelo em si segue um certo padrão que você já está familiarizado, apenas acrescentando ou alterando algumas funções que serão responsáveis pelo processamento e detecção de padrões sobre esses dados providos de imagens.

Dessa forma iniciamos o modelo criando nosso classificador, que inicialmente só executa a ferramenta Sequential, em seguida criamos a camada de entrada e nossa primeira camada oculta, note que dessa vez não estamos usando Dense para isso, mas Conv2D que recebe como parâmetro input_shape = (28,28,1), enquanto anteriormente definíamos um número de neurônios para camada de entrada, agora estamos definindo que a camada de entrada é uma matrix 28x28 de uma camada de profundidade.

O método de ativação da mesma é ‘relu’ e os parâmetros referentes a primeira camada oculta, que nesse caso é o chamado operador de convolução que irá pegar os dados brutos da matriz de entrada, realizar a detecção de suas características, parear com uma nova matriz aplicando funções pixel a pixel e por fim gerando uma série de mapas de características.

Esta é uma forma de redimensionar a imagem, diminuindo seu tamanho e convertendo as informações relevantes de padrões de pixels que compõe a imagem para que seu processamento seja facilitado.

Em seguida é realizado o processo de normalização desses dados, você já deve ter notado que em imagens de baixa qualidade, em torno do traço existem pixels borrados, imprecisos, aqui o operador simplesmente verificará qual a relevância desses pixels e, sempre que possível, irá eliminar do mapa os desnecessários para reduzir ainda mais a quantidade de dados a serem processados.

Repare que aqui via MaxPooling é feito um novo

redimensionamento para uma matrix 2x2, contendo um mapa com apenas as características relevantes a imagem. O que deve ficar bem entendido aqui é que não estamos simplesmente redimensionando uma imagem reduzindo seu tamanho de largura e altura, estamos criando um sistema de indexação interno que pegará a informação de cada pixel e sua relação com seus vizinhos, gerando um mapa de características e reduzindo o número de linhas e colunas de sua matriz, no final do processo temos um mapeamento que consegue reconstruir a imagem original (ou algo muito próximo a ela) mas que para processamento via rede neural possui um formato matricial pequeno e simples.

Como sempre, a segunda camada oculta agora não conta mais com o parâmetro dos dados de entrada.

```
36 classificador.add(BatchNormalization())
37 classificador.add(MaxPooling2D(pool_size = (2,2)))
38 classificador.add(Flatten())
39 classificador.add(Dense(units = 128,
40                         activation = 'relu'))
41 classificador.add(Dropout(0.2))
42 classificador.add(Dense(units = 128,
43                         activation = 'relu'))
44 classificador.add(Dropout(0.2))
45 classificador.add(Dense(units = 10,
46                         activation = 'softmax'))
```

Dando sequência após algumas camadas de processamento, redimensionando os dados de entrada, é executada a ferramenta Flatten que por sua vez irá pegar essas matrizes e seus referentes mapas de características e irá transformar isto em uma lista, onde cada valor (de cada linha) atuará como um neurônio nesta camada intermediária, finalizada essa etapa agora criamos camadas densas, aplicamos dropout em 20% (ou seja, de cada camada o operador pegará 20% dos neurônios e simplesmente irá subtraí-los da camada, a fim de também diminuir seu tamanho) e assim chegamos a camada de saída, com 10 neurônios (já que estamos classificando dados em números de 0 a 9) que por sua vez tem ativação ‘softmax’, função de ativação essa muito parecida com a nossa velha conhecida ‘sigmoid’, porém capaz de categorizar amostras em múltiplas saídas.

```
47 classificador.compile(loss = 'categorical_crossentropy',
48                       optimizer = 'adam',
49                       metrics = ['accuracy'])
50 classificador.fit(etreino,
51                   streino,
52                   batch_size = 128,
53                   epochs = 10,
54                   validation_data = (eteste,steste))
```

Por fim, criamos a estrutura que compila a rede, definindo sua função de perda como ‘categorical_crossentropy’ que como já visto anteriormente, verifica quais dados não foram possíveis de classificar quanto sua proximidade aos vizinhos, otimizador ‘adam’ e como métrica foco em precisão por meio do parâmetro ‘accuracy’. Última parte, por meio da função .fit( ) alimentamos a rede com os dados de etreino e streino, definimos que os valores dos pesos serão atualizados a cada 128 amostras, que executaremos a rede 10 vezes e que haverá um teste de validação comparando os dados encontrados com os de eteste e steste.

Selecionado todo bloco de código, se não houver nenhum erro de sintaxe a rede começará a ser executada. Você irá reparar que, mesmo com todas reduções que fizemos, e mesmo executando nosso modelo apenas 10 vezes, o processamento dessa rede se dará por um bom tempo, dependendo de seu hardware, até mesmo mais de uma hora.

```
60000/60000 [==============================] - 257s 4ms/step - loss: 0.0259 - ac
0.9921 - val_loss: 0.0305 - val_acc: 0.9904
Epoch 7/10
60000/60000 [==============================] - 249s 4ms/step - loss: 0.0230 - ac
0.9928 - val_loss: 0.0364 - val_acc: 0.9894
Epoch 8/10
60000/60000 [==============================] - 294s 5ms/step - loss: 0.0214 - ac
0.9932 - val_loss: 0.0287 - val_acc: 0.9910
Epoch 9/10
60000/60000 [==============================] - 278s 5ms/step - loss: 0.0165 - ac
0.9948 - val_loss: 0.0358 - val_acc: 0.9912
Epoch 10/10
60000/60000 [==============================] - 255s 4ms/step - loss: 0.0162 - ac
0.9953 - val_loss: 0.0364 - val_acc: 0.9902
```

Terminada a execução da rede note que, feito todo processamento de todas camadas e parâmetros que definimos chegamos a excelentes resultados, onde a função de perda nos mostra que apenas 1% dos dados não puderam ser classificados corretamente, enquanto val_acc nos retorna uma margem de acertos/precisão de 99% neste tipo de classificação.

Código Completo:

```
import matplotlib.pyplot as plt
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Flatten, Dropout
from keras.utils import np_utils
from keras.layers import Conv2D, MaxPooling2D
from keras.layers.normalization import BatchNormalization

(etreino,streino),(eteste,steste) = mnist.load_data()

plt.imshow(etreino[0], cmap = 'gray')
plt.title('Classe' + str(streino[0]))

etreino = etreino.reshape(etreino.shape[0],28,28,1)
eteste = eteste.reshape(eteste.shape[0],28,28,1)

etreino = etreino.astype('float32')
eteste = eteste.astype('float32')

etreino /= 255
eteste /= 255

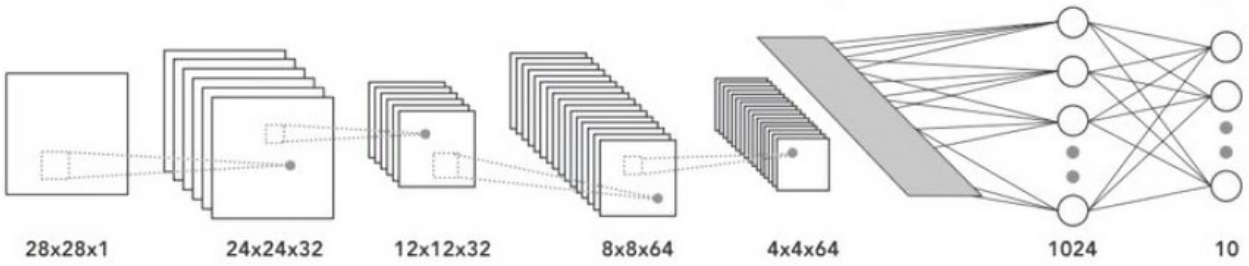
streino = np_utils.to_categorical(streino, 10)
steste = np_utils.to_categorical(steste, 10)
```

```
classificador = Sequential()
classificador.add(Conv2D(32,
                         (3,3),
                         input_shape = (28,28,1),
                         activation = 'relu'))
classificador.add(BatchNormalization())
classificador.add(MaxPooling2D(pool_size = (2,2)))
classificador.add(Conv2D(32,
                         (3,3),
                         activation = 'relu'))
classificador.add(BatchNormalization())
classificador.add(MaxPooling2D(pool_size = (2,2)))
classificador.add(Flatten())
classificador.add(Dense(units = 128,
                        activation = 'relu'))
classificador.add(Dropout(0.2))
classificador.add(Dense(units = 128,
                         activation = 'relu'))
classificador.add(Dropout(0.2))
classificador.add(Dense(units = 10,
                         activation = 'softmax'))
classificador.compile(loss = 'categorical_crossentropy',
                      optimizer = 'adam',
                      metrics = ['accuracy'])
classificador.fit(etreino,
                  streino,
                  batch_size = 128,
                  epochs = 10,
                  validation_data = (eteste,steste))
```

Aqui quero apenas fazer um breve resumo porque acho necessário para melhor entendimento. Sem entrar em detalhes da estrutura de nosso modelo, raciocine que o processo de classificação de imagens é bastante complexo, demanda muito processamento, uma vez que a grosso modo, uma imagem que temos em tela é simplesmente um conjunto de pixels em suas devidas posições com suas respectivas cores.

Internamente, para que possamos realizar processamento via rede neural artificial, devemos fazer uma série de transformações/conversões para que essa imagem se transforme em uma série de mapas matriciais onde possam se aplicar funções.

Para isso, temos que fazer o redimensionamento de tal imagem, sem perda de informação, de uma matriz grande para pequena, que será convertida em dados de camada para a rede neural, por fim, a rede irá aprender a reconhecer as características e padrões de tais matrizes e reconstruir toda essa informação novamente em imagem.



Infográfico mostrando desde a leitura da imagem em sua forma matricial, geração de mapas e polimento, conversão em neurônios e interação com camada subsequente.

Lembrando que este processo é apenas a estrutura da rede, assim como nos modelos anteriores, uma vez que nossa rede esteja funcionando podemos aplicar testes e fazer previsões em cima do mesmo.

```
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Flatten, Conv2D, MaxPooling2D
from keras.utils import np_utils
import numpy as np
from sklearn.model_selection import StratifiedKFold
```

Muito bem, realizado o processamento convencional de nossa rede neural convolucional, hora de aplicar algum teste para verificar a integridade de nossos resultados. Neste tipo de rede neural um dos testes mais eficazes é o nosso velho conhecido teste de validação cruzada, onde podemos separar nossa base de dados em partes e testá-las individualmente.

Aqui, lembrando que temos um modelo que está processando dados a partir de imagens, usaremos uma ferramenta um pouco diferente para o mesmo fim, a StratifiedKFold da biblioteca sklearn. O processo das importações é o mesmo de sempre.

```
seed = 5
np.random.seed(seed)
```

Na sequência criamos uma variável de nome seed com valor inicial 5, que em sua função subsequente fará a alimentação de nossa ferramenta StratifiedKFold com valores randômicos dentro desse intervalo.

```
(etreino,streino), (eteste,steste) = mnist.load_data()
entradas = etreino.reshape(etreino.shape[0], 28, 28, 1)
entradas = entradas.astype('float32')
entradas /= 255
saidas = np_utils.to_categorical(streino, 10)
```

Logo após realizamos aquele procedimento de importação dos dados de nossa base de dados assim como a distribuição desses dados em variáveis de treino, teste, entrada e de saída. Assim como as devidas conversões de formato da mesma forma que realizamos anteriormente na criação de nosso modelo convolucional.

```
kfold = StratifiedKFold(n_splits = 5, shuffle = True, random_state = seed)
resultados = []
```

Em seguida criamos uma variável de nome kfold que inicializa a ferramenta StratifiedKFold, note que definimos os parâmetros n_splits = 5, que significa que a base será dividida em 5 partes iguais, shuffle = True define que as amostras sejam pegas de forma aleatória, para evitar testes "viciados" e random_state que recebe o valor 5 que havíamos definido anteriormente em seed.

```
a = np.zeros(5)
b = np.zeros(shape = (saidas.shape[0], 1))
```

Na etapa seguinte criamos duas variáveis a e b que por meio da função .zeros( ) cria arrays preenchidas com números 0 a serem substituídos durante a aplicação de alguma função.

```
for evalcruzada,svalcruzada in kfold.split(entradas,
                                           np.zeros(shape=(saidas.shape[0],1)
    classificador = Sequential()
    classificador.add(Conv2D(32, (3,3), input_shape=(28,28,1), activation='rel
    classificador.add(MaxPooling2D(pool_size = (2,2)))
    classificador.add(Flatten())
    classificador.add(Dense(units = 128, activation = 'relu'))
    classificador.add(Dense(units = 10, activation = 'softmax'))
    classificador.compile(loss = 'categorical_crossentropy', optimizer='adam',
                          metrics = ['accuracy'])
    classificador.fit(entradas[evalcruzada], saidas[evalcruzada],
                      batch_size = 128, epochs = 5)
    precisao = classificador.evaluate(entradas[svalcruzada], saidas[svalcruzac
    resultados.append(precisao[1])
```

Dando sequência criamos a estrutura de nosso teste de validação cruzada dentro de um laço de repetição, onde são criadas as variáveis temporárias evalcruzada e svalcruzada referentes aos dados para entrada e saída. Por meio da função kfold.split( ) então é feita a divisão e separação de acordo com o formato que havíamos parametrizado anteriormente.

Note que dentro existe uma estrutura de rede neural assim como as anteriores, apenas agora condensada para poupar linhas aqui do livro. Esta

rede por sua vez é alimentada com os dados que forem passados vindos de entradas e saídas sobre evalcruzada assim como seu teste de performance é realizado com os dados atribuídos para entradas e saidas sobre svalcruzada.

Por fim o retorno de todo esse processamento (lembrando que aqui nessa fase a rede será executada novamente) irá alimentar resultados por meio da função .append( ) (uma vez que resultados está em formato de lista).

```
Epoch 2/5
48000/48000 [==============================] - 32s 665us/step - loss: 0.0769 -
acc: 0.9781
Epoch 3/5
48000/48000 [==============================] - 36s 754us/step - loss: 0.0511 -
acc: 0.9844
Epoch 4/5
48000/48000 [==============================] - 33s 685us/step - loss: 0.0366 -
acc: 0.9890
Epoch 5/5
48000/48000 [==============================] - 33s 677us/step - loss: 0.0264 -
acc: 0.9923
12000/12000 [==============================] - 3s 244us/step

In [ ]:
```

Via console podemos ver que após as novas execuções da rede, agora cruzando dados de amostras aleatórias, houve uma margem de acerto de 99%, compatível com o resultado obtido na rede neural do modelo.

```
38 media = sum(resultados) / len(resultados)
```

Por fim é criada a variável media que realiza o cruzamento e divisão dos dados obtidos nas linhas e colunas de resultados.

Nome	Tipo	Tamanho	Valor
a	float64	(5,)	[0. 0. 0. 0. 0.]
b	float64	(60000, 1)	[[0.] [0.]
entradas	float32	(60000, 28, 28, 1)	[[[[0.] [0.]
eteste	uint8	(10000, 28, 28)	[[[0 0 0 ... 0 0 0] [0 0 0 ... 0 0 0]
etreino	uint8	(60000, 28, 28)	[[[0 0 0 ... 0 0 0] [0 0 0 ... 0 0 0]
evalcruzada	int32	(48000,)	[0 1 3 ... 59997 59998 59999]
media	float64	1	0.98415
precisao	list	2	[0.05134425956034101, 0.98475]

Via explorador de variáveis é possível visualizar um resultado quase idêntico ao da execução da validação cruzada, isso é perfeitamente normal desde que os valores sejam muito aproximados (isto se dá porque em certas métricas de visualização de dados pode ocorrer “arredondamentos” que criam uma variação para mais ou para menos, mas sem impacto nos resultados), 98% de margem de acerto no processo de classificação.

Código Completo (Validação Cruzada):

```
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Flatten, Conv2D, MaxPooling2D
from keras.utils import np_utils
import numpy as np
from sklearn.model_selection import StratifiedKFold

seed = 5
np.random.seed(seed)

(etreino,streino), (eteste,steste) = mnist.load_data()
entradas = etreino.reshape(etreino.shape[0], 28, 28, 1)
entradas = entradas.astype('float32')
entradas /= 255
saidas = np_utils.to_categorical(streino, 10)

kfold = StratifiedKFold(n_splits = 5, shuffle = True, random_state = seed)
resultados = []

a = np.zeros(5)
b = np.zeros(shape = (saidas.shape[0], 1))

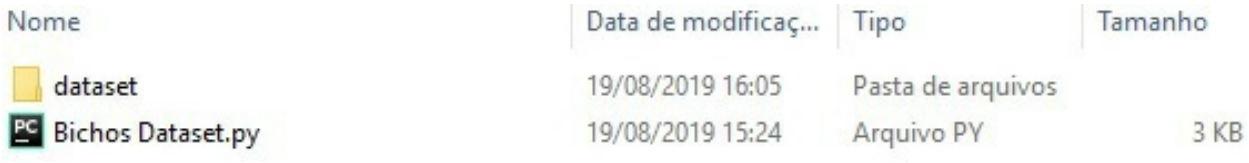
for evalcruzada,svalcruzada in kfold.split(entradas,
                                           np.zeros(shape=(saidas.shape[0],1)
    classificador = Sequential()
    classificador.add(Conv2D(32, (3,3), input_shape=(28,28,1), activation='rel
    classificador.add(MaxPooling2D(pool_size = (2,2)))
    classificador.add(Flatten())
    classificador.add(Dense(units = 128, activation = 'relu'))
    classificador.add(Dense(units = 10, activation = 'softmax'))
    classificador.compile(loss = 'categorical_crossentropy', optimizer='adam',
                          metrics = ['accuracy'])
    classificador.fit(entradas[evalcruzada], saidas[evalcruzada],
                      batch_size = 128, epochs = 5)
    precisao = classificador.evaluate(entradas[svalcruzada], saidas[svalcruzac
    resultados.append(precisao[1])

media = sum(resultados) / len(resultados)
```

Classificação de Imagens de Animais – Bichos Dataset

Avançando um pouco mais com nossos estudos sobre redes neurais convolucionais, foi entendido o processo de transformação de uma imagem para formatos de arquivos suportados pelas ferramentas de redes neurais. Também fiemos um breve entendimento sobre a forma como podemos treinar um modelo para que este aprenda a identificar características e a partir desse aprendizado, realizar previsões, porém até o momento estávamos nos atendo a modelos de exemplo das bibliotecas em questão.

Na prática, quando você precisar criar um identificador e classificador de imagem este será de uma base de objetos muito diferente dos exemplos anteriores. Em função disso neste tópico vamos abordar o processo de aprendizado de máquina com base em um conjunto de imagens de gatos e cachorros, posteriormente nossa rede será treinada para a partir de seu modelo, identificar novas imagens que fornecermos classificando-as como gato ou cachorro. Note que agora para tentarmos trazer um modelo próximo a realidade, dificultaremos de propósito esse processo, uma vez que estamos classificando 2 tipos de animais esteticamente bastante parecidos.



O primeiro grande diferencial deste modelo que estudaremos agora é que o mesmo parte do princípio de classificar gatos e cachorros a partir de imagens separadas de gatos e cachorros obviamente, mas completamente aleatórias, de diferentes tamanhos com diferentes características, e não padronizadas como nos exemplos anteriores, dessa forma, essa base de dados que criaremos se aproxima mais das aplicações reais desse tipo de rede neural. No mesmo diretório onde criamos nosso arquivo de código extraímos a base de dados.

Nome	Data de modificaç...	Tipo	Tamanho
test_set	19/08/2019 16:05	Pasta de arquivos	
training_set	19/08/2019 16:05	Pasta de arquivos	
.DS_Store	23/06/2018 13:15	Arquivo DS_STORE	7 KB

Dentro de dataset podemos verificar que temos duas pastas referentes a base de treino e de teste.

Nome	Data de modificaç...	Tipo	Tamanho
cachorro	19/08/2019 16:05	Pasta de arquivos	
gato	19/08/2019 16:05	Pasta de arquivos	
.DS_Store	21/06/2018 19:31	Arquivo DS_STORE	9 KB

Abrindo training_set podemos verificar que existem as categorias cachorro e gato.



Abrindo cachorros note que temos um conjunto de dados de imagens de cachorros diversos (2000 imagens para ser mais exato) com características diversas (tamanho, formato, pelagem, etc...) e em ações diversas (posição na foto).

O interessante desse tipo de dataset é que existe muita coisa

desnecessária em cada foto, objetos, pessoas, legenda, etc... e nossa rede neural terá de identificar o que é característica apenas do cachorro na foto, para dessa forma realizar previsões posteriormente.

Outro ponto interessante é que esse modelo é facilmente aplicável para classificação de qualquer tipo de objeto, e para classificação de diferentes tipos de objeto (mais de 2), a nível de base de dados a única alteração seria a criação de mais pastas (aqui temos duas, gatos e cachorros) referentes a cada tipo de objeto a ser classificado.

Bichos Dataset.py

```
1 from keras.models import Sequential
2 from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout
3 from keras.layers.normalization import BatchNormalization
4 from keras.preprocessing.image import ImageDataGenerator
```

Como sempre, o processo se dá iniciando com as importações das bibliotecas, módulos e ferramentas que estaremos usando ao longo do código. Dessa forma criamos um novo arquivo de código de nome Bichos Dataset e damos início as importações das bibliotecas já conhecidas, única diferença é que para este modelo importamos do módulo preprocessing da biblioteca keras a ferramenta ImageDataGenerator. Esta ferramenta por sua vez será usada para gerar dados a partir de padrões reconhecidos.

Em outras palavras, estaremos usando uma base de dados relativamente pequena, a medida que a rede for encontrando padrões que ela tem certeza ser de gato ou certeza ser de cachorro, ela irá gerar uma nova imagem com indexamento próprio com tais características, facilitando o processo de reforço de aprendizado.

A ideia desta ferramenta é compensar a alta complexidade encontrada em processamento de imagens, identificar caracteres como no modelo anterior é muito fácil, identificar características minúsculas, a partir de imagens tão variadas de animais (que por sua vez possuem muitas características variáveis) é um processo bastante complexo até mesmo para redes neurais, sendo assim, nas etapas onde são aplicados métodos de aprendizagem por reforço, a rede usará de características identificadas e separadas para ganhar eficiência em seu processamento.

Na fase de executar a rede propriamente dita você verá que este modelo de rede neural sempre começa com uma margem de acertos bastante

baixa e época a época ela aumenta e muito sua precisão, isto se dá por funções internas de ferramentas como a ImageDataGenerator.

```
gerador_treino = ImageDataGenerator(rescale = 1.0/255,
                                    rotation_range = 7,
                                    horizontal_flip = True,
                                    shear_range = 0.2,
                                    height_shift_range = 0.07,
                                    zoom_range = 0.2)
gerador_teste = ImageDataGenerator(rescale = 1.0/255)
```

Feitas as importações de forma correta podemos dar início diretamente na criação de nosso gerador, em algumas literaturas essa fase é chamada Augmentation, pois estamos justamente junto ao nosso polimento inicial dos dados, realizando o aumento de nossa base para que se tenha melhor precisão no processamento das imagens.

Para isso criamos nossa variável gerador_treino que inicializa a ferramenta ImageDataGenerator que por sua vez recebe uma série de parâmetros. O primeiro deles rescale = 1.0/255 nada mais é do que o escalonamento da imagem, rotation_range diz respeito ao fator de rotação (no sentido de angular a imagem nos dois sentidos por um ângulo pré-determinado e salvar também essas informações, horizontal_flip = True, que irá espelhar a imagem e guardar as informações da mesma espelhada, height_shift_range irá encontrar e salvar as informações referentes a possível distorção vertical da imagem e zoom_range por sua vez irá aplicar um zoom de 2% e salvar essa informação.

Note que dessa forma, não são apenas convertidos os dados brutos referentes a cada mapeamento de pixel normal, mas essas informações adicionais, quando salvas em blocos, ajudarão o interpretador e eliminar distorções e encontrar padrões mais precisos. Já para nossa variável gerador_teste é feito simplesmente a conversão de escala.

```
base_treino = gerador_treino.flow_from_directory('dataset/training_set',
                                                 target_size = (64,64),
                                                 batch_size = 32,
                                                 class_mode = 'binary')
base_teste = gerador_teste.flow_from_directory('dataset/test_set',
                                               target_size = (64,64),
                                               batch_size = 32,
                                               class_mode = 'binary')
```

Em seguida são criadas as variáveis referentes as nossas variáveis

de entradas e saídas com base nas importações das imagens. Inicialmente criamos a variável base_treino que instancia e executa nosso gerador_treino executa sua função .flow_from_directory( ) que por sua vez é capaz de ler os arquivos que estão dentro de um diretório específico, lembre-se que nossa base de dados está dividida em imagens de gatos e cachorros divididas para treino e teste dentro de pastas separadas.

Como parâmetros, inicialmente é passado o caminho da pasta onde se encontram os arquivos, em nosso caso, ‘dataset/training_set’, na sequência target_size realiza a primeira conversão para uma matriz de mapeamento 64x64, onde batch_size define a criação de blocos de 32 em 32 pixels, convertendo para binário via class_mode. Exatamente o mesmo processo é feito para nossa base_teste, apenas modificando o caminho do diretório onde se encontram as imagens separadas para teste.

```
classificador = Sequential()
classificador.add(Conv2D(32,
                         (3,3),
                         input_shape = (64,64,3),
                         activation = 'relu'))
classificador.add(BatchNormalization())
classificador.add(MaxPooling2D(pool_size = (2,2)))
classificador.add(Conv2D(32,
                         (3,3),
                         activation = 'relu'))
classificador.add(BatchNormalization())
classificador.add(MaxPooling2D(pool_size = (2,2)))
classificador.add(Flatten())
classificador.add(Dense(units = 128,
                        activation = 'relu'))
classificador.add(Dropout(0.2))
classificador.add(Dense(units = 128,
                        activation = 'relu'))
classificador.add(Dropout(0.2))
classificador.add(Dense(units = 1,
                        activation = 'sigmoid'))
```

Prosseguindo com nosso código hora de criar a estrutura de rede neural, onde codificamos todo aquele referencial teórico para um formato lógico a ser processado. Inicialmente criamos nosso classificador que inicializa Sequential sem parâmetros.

Em seguida é criada a primeira parte de nossa etapa de convolução, onde adicionamos uma camada onde parametrizamos que como neurônios da camada de entrada, temos uma estrutura matricial de 54 por 64 pixels em 3

camadas, que será dividida em blocos de 3x3 resultando em uma primeira camada oculta de 32 neurônios, passando pela ativação relu. Em seguida é adicionada uma camada de normalização por meio da função BatchNormalization( ), sem parâmetros mesmo.

Logo após é adicionada uma camada de polimento onde a matrix anterior 3x3 passa a ser um mapa 2x2. A seguir é repetida esta primeira etapa até o momento da dição da camada Flatten, que por sua vez irá pegar todo resultado da fase de convolução e converter para uma indexação em forma de lista onde cada valor da lista se tornará um neurônio da camada subsequente.

Na sequência é criada a fase de camadas densas da rede, onde a primeira camada delas possui em sua estrutura 128 neurônios e é aplicada a função de ativação relu.

Subsequente é feito o processo de seleção de uma amostra desses neurônios aleatoriamente para serem descartados do processo, apenas a fim de filtrar essa camada e poupar processamento, isto é feito por meio da ferramenta Dropout, parametrizada para descartar 20% dos neurônios da camada.

Finalmente é criada a camada de saída, onde haverá apenas 1 neurônio, passando pela função de ativação sigmoid em função de estarmos classificando de forma binária (ou gato ou cachorro), se fossem usadas 3 ou mais amostras lembre-se que a função de ativação seria a softmax.

```
44 classificador.compile(optimizer = 'adam',
45                       loss = 'binary_crossentropy',
46                       metrics = ['accuracy'])
47 classificador.fit_generator(base_treino,
48                             steps_per_epoch = 4000,
49                             epochs = 5,
50                             validation_data = base_teste,
51                             validation_steps = 1000)
```

Então é criada a estrutura de compilação da rede, nada diferente do usual usado em outros modelos e por fim é criada a camada que alimenta a rede e a coloca em execução, note que agora, em um modelo de rede neural convolucional, este processo se dá pela função .fit_generator( ) onde passamos como parâmetros os dados contidos em base_treino, steps_per_epoch define o número de execuções sobre cada amostra (se não houver 4000 amostras a rede irá realizar testes sobre amostras repetidas ou

geradas anteriormente), epochs como de costume define que a toda a rede neural será executada 5 vezes (reforçando seu aprendizado), validation_data irá, como o próprio nome dá a ideia, fazer a verificação para ver se os dados encontrados pela rede conferem perfeitamente com os da base de teste em estrutura e por fim é definido que essa validação será feita sobre 1000 amostras aleatórias.

Selecionando e executando todo esse bloco de código, se não houve nenhum erro de sintaxe você verá a rede em execução via console.

```
Use tf.cast instead.
Epoch 1/5
4000/4000 [==============================] - 3844s 961ms/step - loss: 0.4491 -
acc: 0.7805 - val_loss: 0.5055 - val_acc: 0.7933
Epoch 2/5
4000/4000 [==============================] - 6296s 2s/step - loss: 0.2024 - acc
0.9180 - val_loss: 0.7441 - val_acc: 0.7602
Epoch 3/5
4000/4000 [==============================] - 3686s 921ms/step - loss: 0.1213 -
acc: 0.9537 - val_loss: 0.7576 - val_acc: 0.7867
Epoch 4/5
4000/4000 [==============================] - 65106s 16s/step - loss: 0.0889 - a
0.9668 - val_loss: 0.7722 - val_acc: 0.7861
```

Você irá notar duas coisas a partir desse modelo, primeira delas é que mesmo realizadas todas as conversões, o processamento da mesma é relativamente lento, uma vez que internamente existe um enorme volume de dados a ser processado.

Segundo, que este modelo de rede neural se inicia com uma margem de acertos bastante baixa (78% na primeira época de execução) e à medida que vai avançando em épocas essa margem sobe consideravelmente (na quarta época a margem de acertos era de 96%).

```
Epoch 2/5
4000/4000 [==============================] - 6296s 2s/step - loss: 0.2024 - acc
0.9180 - val_loss: 0.7441 - val_acc: 0.7602
Epoch 3/5
4000/4000 [==============================] - 3686s 921ms/step - loss: 0.1213 -
acc: 0.9537 - val_loss: 0.7576 - val_acc: 0.7867
Epoch 4/5
4000/4000 [==============================] - 65106s 16s/step - loss: 0.0889 - a
0.9668 - val_loss: 0.7722 - val_acc: 0.7861
Epoch 5/5
4000/4000 [==============================] - 4074s 1s/step - loss: 0.0713 - acc
0.9741 - val_loss: 0.8693 - val_acc: 0.7931
Traceback (most recent call last):
```

Por fim na última época a margem de acertos final foi de 97%, lembrando que aqui, apenas como exemplo, executamos este modelo apenas 5 vezes, em aplicações reais é possível aumentar e estabilizar essa margem ainda mais, executando a rede por mais épocas, fazendo valer mais seu aprendizado por reforço.

```
53 import numpy as np
54 from keras.preprocessing import image
55 imagem_teste = image.load_img('dataset/test_set/gato/cat.3538.jpg',
56                               target_size = (64,64))
57 imagem_teste = image.img_to_array(imagem_teste)
58 imagem_teste = np.expand_dims(imagem_teste,
59                               axis = 0)
60 previsor = classificador.predict(imagem_teste)
```

Uma vez realizada a execução de nossa rede neural artificial convolucional, temos em mãos um modelo pronto para ser usado para testes. Como de costume, para não nos estendermos muito aqui no livro, podemos realizar um simples teste sobre uma amostra, algo próximo da aplicação real justamente por uma rede dessa ser usada para classificar um animal a partir de uma imagem fornecida.

Para isso realizamos a importação da numpy que até o momento não havíamos utilizado nesse modelo, também realizamos a importação da ferramenta image do módulo preprocessing da biblioteca keras. Em seguida é criada a variável imagem_teste que por meio da função image.load_img( ) faz a leitura de um arquivo escolhido definido pelo usuário, apenas complementando, é parametrizado que esta imagem no processo de leitura, independentemente de seu tamanho original, será convertida para 64x64.



A imagem em questão é essa, escolhida aleatoriamente (e lembrando que apesar de sua nomenclatura “cat...” o interpretador fará a leitura, as conversões e a identificação com base nas características da imagem. Na sequência é feita a conversão da imagem propriamente dita para uma array do tipo numpy por meio da função img_to_array( ).

Em seguida é parametrizado que expand_dims permite que tais dados sejam empilhados em forma de lista por meio do parâmetro axis = 0, lembre-se que na fase onde os dados passam pela camada Flatten, eles serão convertidos para dados sequenciais que se tornarão neurônios de uma camada da rede.

Por fim, por meio da função .predict( ) é passada esta imagem pelo processamento de nossa rede para que seja identificada como tal.

Nome	Tipo	Tamanho	Valor
imagem_teste	float32	(1, 64, 64, 3)	[[[[37. 45. 48.] [34. 46. 62.]
previsor	float32	(1, 1)	[[1.]]
resultado_final	dict	2	{'cachorro':0, 'gato':1}

O retorno neste caso foi, como esperado, que a imagem fornecida foi de fato classificada como uma imagem de um gato.

Código Completo:

```
import numpy as np
from keras.preprocessing import image
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout
from keras.layers.normalization import BatchNormalization
from keras.preprocessing.image import ImageDataGenerator

gerador_treino = ImageDataGenerator(rescale = 1.0/255,
                                    rotation_range = 7,
                                    horizontal_flip = True,
                                    shear_range = 0.2,
                                    height_shift_range = 0.07,
                                    zoom_range = 0.2)
gerador_teste = ImageDataGenerator(rescale = 1.0/255)

base_treino = gerador_treino.flow_from_directory('dataset/training_set',
                                                 target_size = (64,64),
                                                 batch_size = 32,
                                                 class_mode = 'binary')
base_teste = gerador_teste.flow_from_directory('dataset/test_set',
                                               target_size = (64,64),
                                               batch_size = 32,
                                               class_mode = 'binary')
```

```
classificador = Sequential()
classificador.add(Conv2D(32,
                         (3,3),
                         input_shape = (64,64,3),
                         activation = 'relu'))
classificador.add(BatchNormalization())
classificador.add(MaxPooling2D(pool_size = (2,2)))
classificador.add(Conv2D(32,
                         (3,3),
                         activation = 'relu'))
classificador.add(BatchNormalization())
classificador.add(MaxPooling2D(pool_size = (2,2)))
classificador.add(Flatten())
classificador.add(Dense(units = 128,
                        activation = 'relu'))
classificador.add(Dropout(0.2))
classificador.add(Dense(units = 128,
                        activation = 'relu'))
classificador.add(Dropout(0.2))
classificador.add(Dense(units = 1,
                        activation = 'sigmoid'))
classificador.compile(optimizer = 'adam',
                      loss = 'binary_crossentropy',
                      metrics = ['accuracy'])
classificador.fit_generator(base_treino,
                            steps_per_epoch = 4000,
                            epochs = 5,
                            validation_data = base_teste,
                            validation_steps = 1000)
```

```
imagem_teste = image.load_img('dataset/test_set/gato/cat.3538.jpg',
                              target_size = (64,64))
imagem_teste = image.img_to_array(imagem_teste)
imagem_teste = np.expand_dims(imagem_teste,
                              axis = 0)
previsor = classificador.predict(imagem_teste)
base_treino.class_indices
```

Classificação a Partir de Imagens – TensorFlow – Fashion Dataset

Para finalizar este capítulo, vamos dar uma breve explanada sobre como podemos criar modelos de redes neurais artificiais convolucionais a partir do TensorFlow. Esta biblioteca, desenvolvida pelo Google e liberada para uso no modelo open-source, possui um modelo lógico e estrutural que por sua vez vem sendo aprimorado pela comunidade para que tal ferramenta seja de fácil acesso e o principal, ofereça melhor performance se comparado com as outras bibliotecas científicas desenvolvidas para Python.

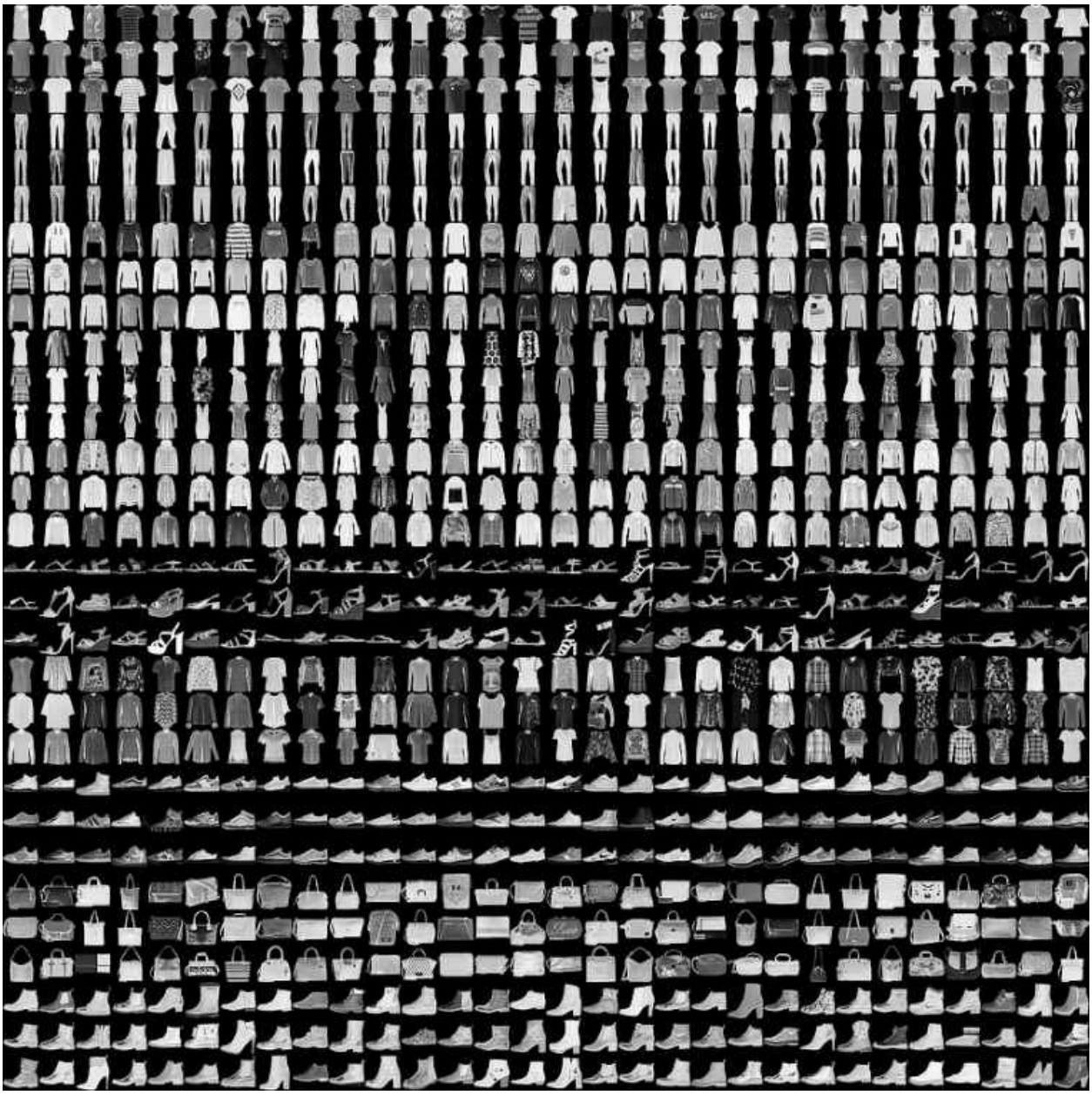
Internamente, o modelo de funcionamento de uma rede neural via TensorFlow se difere do convencionam uma vez que sua estrutura lógica segue um padrão de fluxo, onde as camadas de processamento realizam processamento em blocos e dessa forma se obtém melhor performance.

Raciocine que em um modelo de rede neural como viemos trabalhando até então criamos a estrutura de camadas “engessadas” entre si, uma pequena alteração que impeça a comunicação correta entre camadas resulta em uma rede que não consegue ser executada.

O principal do TensorFlow veio justamente nesse quesito, existe obviamente a comunicação entre camadas da rede, mas estas por sua vez podem ser desmembradas e processada a parte.

De forma geral, para fácil entendimento, via TensorFlow é possível pegar um problema computacional de grande complexidade e o dividir para processamento em pequenos blocos (até mesmo realizado em máquinas diferentes) para que se obtenha maior performance.

Para nosso exemplo, usaremos de uma base de dados Fashion MNIST, base essa integrante das bases de exemplo da biblioteca keras.



Basicamente a biblioteca Fashion conta com 70000 amostras divididas em 10 categorias diferentes de roupas e acessórios. A partir dessa base, treinaremos uma rede neural capaz de classificar novas peças que forem fornecidas pelo usuário.

O grande diferencial que você notará em relação ao modelo anterior é a performance. Muito provavelmente você ao executar o modelo anterior teve um tempo de execução da rede de mais ou menos uma hora. Aqui faremos um tipo de classificação com um volume considerável de amostras que será feito em poucos segundos via TensorFlow.

Fashion Dataset TensorFlow.py

```
1 from __future__ import absolute_import, division, print_function, unicode_l
2 import tensorflow as tf
3 from tensorflow import keras
4 import numpy as np
5 import matplotlib.pyplot as plt
```

Como sempre o processo se inicia com a importação das bibliotecas, módulos e ferramentas necessárias. Note que aqui já de início estamos realizando um tipo de importação até agora não vista em outro modelo, from __future__ estamos importando uma série de ferramentas internas ou experimentais do Python, a nomenclatura com "__" antes e depois de um determinado nome significa que tal nome é reservado para uma função interna, não sendo possível atribuir a uma variável ou outra coisa.

De __future__ importamos absolute_import, division, print_function e unicode_literals. Em seguida importamos tensorflow e a referenciamos como tf, já as linhas abaixo são de importações de bibliotecas conhecidas.

```
7 base = keras.datasets.fashion_mnist
8
```

Em seguida criamos nossa variável de nome base que recebe o conteúdo de fashion_mnist das bibliotecas exemplo do keras.

```
 9 (etreino,streino),(eteste,steste) = base.load_data()
10
11 rotulos = ['T-shirt/top','Trouser','Pullover','Dress','Coat','Sandal','Shirt',
12             'Sneaker','Bag','Ankle boot']
13
14 etreino = etreino / 255
15 streino = streino / 255
```

Após selecionada e executada a linha de código anterior, é feita a respectiva importação, na sequência podemos criar nossas variáveis designadas para treino e teste que recebem o conteúdo contido em base, importado anteriormente.

Da mesma forma é criada a variável rotulos que por sua vez recebe como atributos, em forma de lista, os nomes das peças de vestuário a serem classificadas. Para finalizar este bloco, realizamos a conversão dos dados brutos de int para float para que possamos os tratar de forma matricial.

Nome	Tipo	Tamanho	Valor
eteste	uint8	(10000, 28, 28)	[[[0 0 0 ... 0 0 0] [0 0 0 ... 0 0 0]]
etreino	float64	(60000, 28, 28)	[[[0. 0. 0. ... 0. 0. 0.] [0. 0. 0. ... 0. 0. 0.]]
rotulos	list	10	['T-shirt/top', 'Trouser', 'Pullover', 'Dres 'Coat', 'Sandal', 'Shi ...
steste	uint8	(10000,)	[9 2 1 ... 8 1 5]
streino	float64	(60000,)	[0.03529412 0. 0. ... 0.0117 0. 0.01960784 ...

Se até o momento não houve nenhum erro de sintaxe, podemos verificar via explorador de variáveis que de fato foram criadas as referentes variáveis para entradas de treino e teste assim como para as saídas de treino e teste.

Apenas por hora note que etreino possui 60000 amostras que por sua vez estão no formato de uma matriz 28x28, eteste por sua vez possui 10000 amostras no mesmo formato.

```
17 plt.figure(figsize = (10,10))
18 for i in range(25):
19     plt.subplot(5,5,i+1)
20     plt.xticks([])
21     plt.yticks([])
22     plt.grid(False)
23     plt.imshow(etreino[i],
24                cmap = plt.cm.binary)
25     plt.xlabel(rotulos,[steste[i]])
26 plt.show()
```

Em seguida podemos fazer a visualização de uma amostra dessa base de dados, aqui, um pouco diferente do que estamos habituados a fazer, vamos reconstruir uma imagem pixel a pixel e exibir em nosso console. Para isso passaremos alguns argumentos para nossa matplotlib, o primeiro deles é que a imagem a ser exibida terá um tamanho de plotagem (e não de pixels) 10x10.

Em seguida criamos um laço de repetição para reagruparmos os dados necessários a reconstrução. Basicamente o que parametrizamos é a importação do primeiro objeto de nossa base, reagrupando todos dados referentes ao seu mapeamento de pixels e exibindo em tela.



```
['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat', 'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']
```

Selecionando e executando o bloco de código anterior podemos ver que a plotagem foi feta de forma correta, inclusive é possível reconhecer que tal amostra é uma bota.

```
28 classificador = keras.Sequential([
29     keras.layers.Flatten(input_shape=(28, 28)),
30     keras.layers.Dense(128, activation=tf.nn.relu),
31     keras.layers.Dense(10, activation=tf.nn.softmax)
32 ])
```

Dando início a criação da estrutura da rede neural, note algumas particularidades no que diz respeito ao formato da codificação mesmo.

Inicialmente criamos nossa variável de nome classificador que imediatamente instancia e inicializa keras.Sequential( ), internamente já fazemos a criação das camadas, sendo a primeira delas já realizando o processo de Flatten, pegando a amostra em tamanho 28x28, estendendo a mesma e definindo cada valor como um neurônio da camada de entrada.

Logo após é criada uma camada intermediária densa onde apenas é necessário definir o número de neurônios (128) e como ativação usamos relu, porém agora a sintaxe nos mostra que estamos usando relu função interna da biblioteca TensorFlow por meio do parâmetro tf.nn.relu.

Por fim é criada a camada de saída, com 10 neurônios e da mesma forma que na camada anterior, usamos a função de ativação interna do TensorFlow, neste caso, como é uma classificação com 10 saídas diferentes, softmax.

```
34 classificador.compile(optimizer='adam',
35               loss='sparse_categorical_crossentropy',
36               metrics=['accuracy'])
37 classificador.fit(etreino,
38                   streino,
39                   epochs = 5)
```

A estrutura da rede neural convolucional já está pronta (sim, em 3

linhas de código), podemos assim criar as camadas de compilação da mesma, com os mesmos moldes que estamos acostumados, e a camada que alimentará a rede e a colocará em funcionamento, onde apenas repassamos como dados o conteúdo de etreino e streino e definimos que tal rede será executada 5 vezes, não é necessário nenhum parâmetro adicional.

Como comentado anteriormente, você se surpreenderá com a eficiência desse modelo, aqui, processando 60000 amostras em um tempo médio de 3 segundos.

```
    ...:                    epochs = 5)
Epoch 1/5
60000/60000 [==============================] - 6s 107us/sample - loss: 0.0028 -
acc: 0.9995
Epoch 2/5
60000/60000 [==============================] - 6s 102us/sample - loss: 3.7276e-0
- acc: 1.0000
Epoch 3/5
60000/60000 [==============================] - 6s 101us/sample - loss: 1.7924e-0
- acc: 1.0000
Epoch 4/5
60000/60000 [==============================] - 6s 102us/sample - loss: 1.2905e-0
- acc: 1.0000
Epoch 5/5
60000/60000 [==============================] - 6s 102us/sample - loss: 1.1569e-0
- acc: 1.0000
```

Poucos segundos após o início da execução da rede a mesma já estará treinada, note que ela atinge uma margem de acertos de 99% (sendo honesto, o indicador de acc nos mostra 1.0 – 100%, mas este valor não deve ser real, apenas algo muito próximo a 100%)

Pronto, sua rede neural via TensorFlow já está treinada e pronta para execução de testes.

Código Completo:

```
from __future__ import absolute_import, division, print_function, unicode_lite
import tensorflow as tf
from tensorflow import keras
import numpy as np
import matplotlib.pyplot as plt

base = keras.datasets.fashion_mnist

(etreino,streino),(eteste,steste) = base.load_data()

rotulos = ['T-shirt/top','Trouser','Pullover','Dress','Coat','Sandal','Shirt',
           'Sneaker','Bag','Ankle boot']

etreino = etreino / 255
streino = streino / 255

plt.figure(figsize = (10,10))
for i in range(25):
    plt.subplot(5,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(etreino[i],
               cmap = plt.cm.binary)
    plt.xlabel(rotulos,[steste[i]])
plt.show()
```

```
classificador = keras.Sequential([
    keras.layers.Flatten(input_shape=(28, 28)),
    keras.layers.Dense(128, activation=tf.nn.relu),
    keras.layers.Dense(10, activation=tf.nn.softmax)
])

classificador.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
classificador.fit(etreino,
                  streino,
                  epochs = 5)
```

REDES NEURAIS ARTIFICIAIS RECORRENTES

Avançando com nossos estudos, hora de dedicarmos um tempinho a entender o que são as chamadas redes neurais recorrentes. Diferente dos demais tipos vistos até agora, este modelo de rede recebe um capítulo específico para si em função de ser um modelo com certas particularidades que os modelos anteriores não possuíam.

O primeiro grande diferencial de uma rede neural recorrente é que, diferente as outras, ela é focada no processamento de dados sequenciais dentro de um intervalo de tempo, dessa forma, pós processamento da rede é possível que sejam realizadas previsões das ações seguintes que irão ocorrer dentro da escala de tempo.

Para ficar mais claro, imagine um consultório médico, são captados e processados os dados de todos agendamentos de todos os dias das 8 da manhã até as 5 da tarde, digamos que o consultório passará a ter expediente até as 6 horas da tarde, a rede será capaz de descobrir o padrão das horas anteriores e prever qual será a demanda de atendimentos nessa hora adicional.

O exemplo que usaremos a seguir é uma análise temporal de abertura e fechamento de valores de ações, para realizar previsão se uma determinada ação irá subir ou baixar de valor com base no padrão dos dias anteriores.

Estes tipos de redes são facilmente adaptáveis para reconhecimento de linguagem natural e até mesmo previsão de texto. Tudo isso se dá por uma característica desse tipo de rede que é a chamada memória de longo e curto intervalo de tempo.

Em inglês Long Short-term Memory, é um modelo de rede neural artificial recorrente onde a rede tem capacidade de trabalhar em loops definidos para que se encontre padrões de informações persistentes. Raciocine por hora que essa arquitetura cria um modelo onde existem informações de entrada, o algoritmo irá decidir o que será armazenado e o que será removido com base numa primeira etapa de processamento de dados

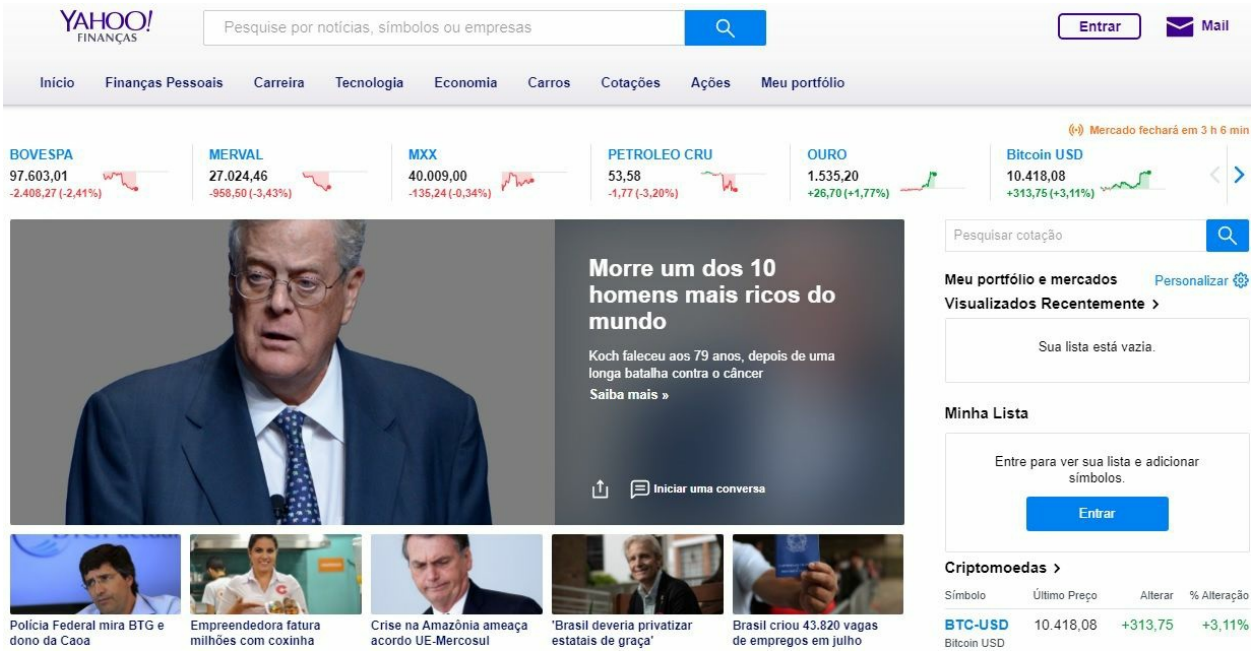
sob aplicação de uma função sigmoid, se o retorno for próximo a 0 tal dado é descartada, se próximo a 1 esse dado é mantido e reprocessado no mesmo neurônio, agora sob a aplicação de uma função chamada tangente hiperbólica.

Em seguida atualiza o estado antigo e parte para a etapa onde está situada a camada de saída. Dessa forma é criada uma espécie de “memória” capaz de ser reutilizada a qualquer momento para autocompletar um determinado padrão.

Previsão de Séries Temporais – Bolsa Dataset

Para entendermos de fato, e na prática, o que é uma rede neural artificial recorrente, vamos usar de uma das principais aplicações deste modelo, a de previsão de preços de ações.

O modelo que estaremos criando se aplica a todo tipo de previsão com base em série temporal, em outras palavras, sempre que trabalharmos sobre uma base de dados que oferece um padrão em relação a tempo, seja horas, dias, semanas, meses, podemos fazer previsões do futuro imediato. Para isso, usaremos de uma base real e atual extraída diretamente do site Yahoo Finanças.



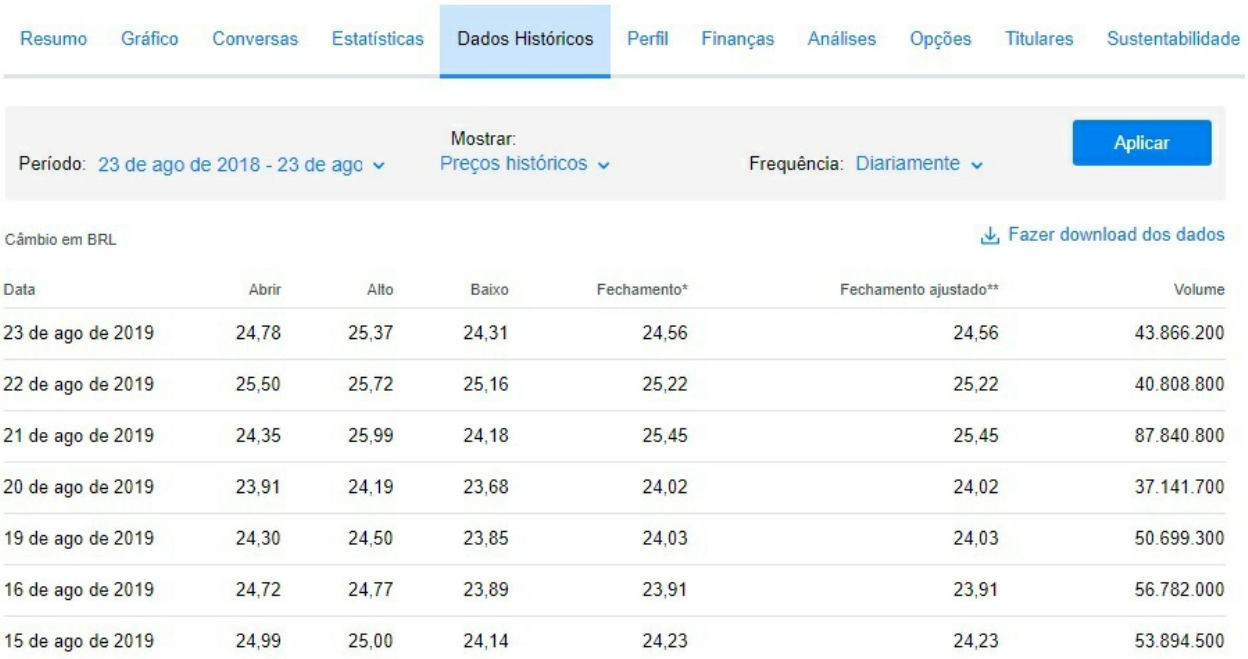
Inicialmente acessamos o site do Yahoo Finanças, usando a própria ferramenta de busca do site pesquisamos por Petrobras.



A base que usaremos é a PETR4.SA.



Na página dedicada a Petrobras, é possível acompanhar a atualização de todos os índices.



Resumo Gráfico Conversas Estatísticas Dados Históricos Perfil Finanças Análises Opções Titulares Sustentabilidade

Período: 23 de ago de 2018 - 23 de ago Mostrar: Preços históricos Frequência: Diariamente Aplicar

Câmbio em BRL Fazer download dos dados

Data	Abrir	Alto	Baixo	Fechamento*	Fechamento ajustado**	Volume
23 de ago de 2019	24,78	25,37	24,31	24,56	24,56	43.866.200
22 de ago de 2019	25,50	25,72	25,16	25,22	25,22	40.808.800
21 de ago de 2019	24,35	25,99	24,18	25,45	25,45	87.840.800
20 de ago de 2019	23,91	24,19	23,68	24,02	24,02	37.141.700
19 de ago de 2019	24,30	24,50	23,85	24,03	24,03	50.699.300
16 de ago de 2019	24,72	24,77	23,89	23,91	23,91	56.782.000
15 de ago de 2019	24,99	25,00	24,14	24,23	24,23	53.894.500

Por fim clicando em Dados Históricos é possível definir um intervalo de tempo e fazer o download de todos os dados em um arquivo formato .csv. Repare que a base que usaremos possui como informações a data (dia), o valor de abertura da ação, o valor mais alto atingido naquele dia, assim como o valor mais baixo e seu valor de fechamento. Essas informações já são mais do que suficientes para treinar nossa rede e realizarmos previsões a partir dela.

Apenas lembrando que um ponto importante de toda base de dados é seu volume de dados, o interessante para este modelo é você em outro momento baixar uma base de dados com alguns anos de informação adicional, dessa forma o aprendizado da rede chegará a melhores resultados.

Bolsa Dataset.py

```
1 import numpy as np
2 import pandas as pd
3 from keras.models import Sequential
4 from keras.layers import Dense, Dropout
5 from sklearn.preprocessing import MinMaxScaler
```

Como sempre, o processo se inicia com as devidas importações. Para nosso modelo importamos como de costume, as bibliotecas numpy e pandas, assim como as ferramentas Sequential, Dense e Dropout de seus respectivos módulos da biblioteca keras.

```
7 base = pd.read_csv('petr4-treinamento.csv')
8
```

Em seguida, como de costume, criamos nossa variável de nome base que receberá todos os dados importados pela função .read_csv( ) de nosso arquivo baixado do site Yahoo Finanças.

```
 9 base = base.dropna()
10 base_treino = base.iloc[:, 1:2].values
```

Na sequência realizamos um breve polimento dos dados, primeiro, excluindo todos registros faltantes de nossa base via função .dropna( ), em seguida criamos nossa variável de nome base_treino que recebe pelo método .iloc[ ].values todos os dados de todas as linhas, e os dados da coluna 1 (lembrando que fora a indexação criada pela IDE, a leitura de todo arquivo .csv começa em índice 0), referente aos valores de abertura das ações.

Nome	Tipo	Tamanho	Valor
base	DataFrame	(1242, 7)	Column names: Date, Open, High, Low, Close, A Close, Volume
base_treino	float64	(1242, 1)	[[19.99] [19.809999]]

Selecionando e executando o bloco de código anterior podemos verificar que as variáveis foram criadas corretamente, delas, base_treino agora possui 1242 registros divididos em apenas uma categoria (valor de abertura da ação).

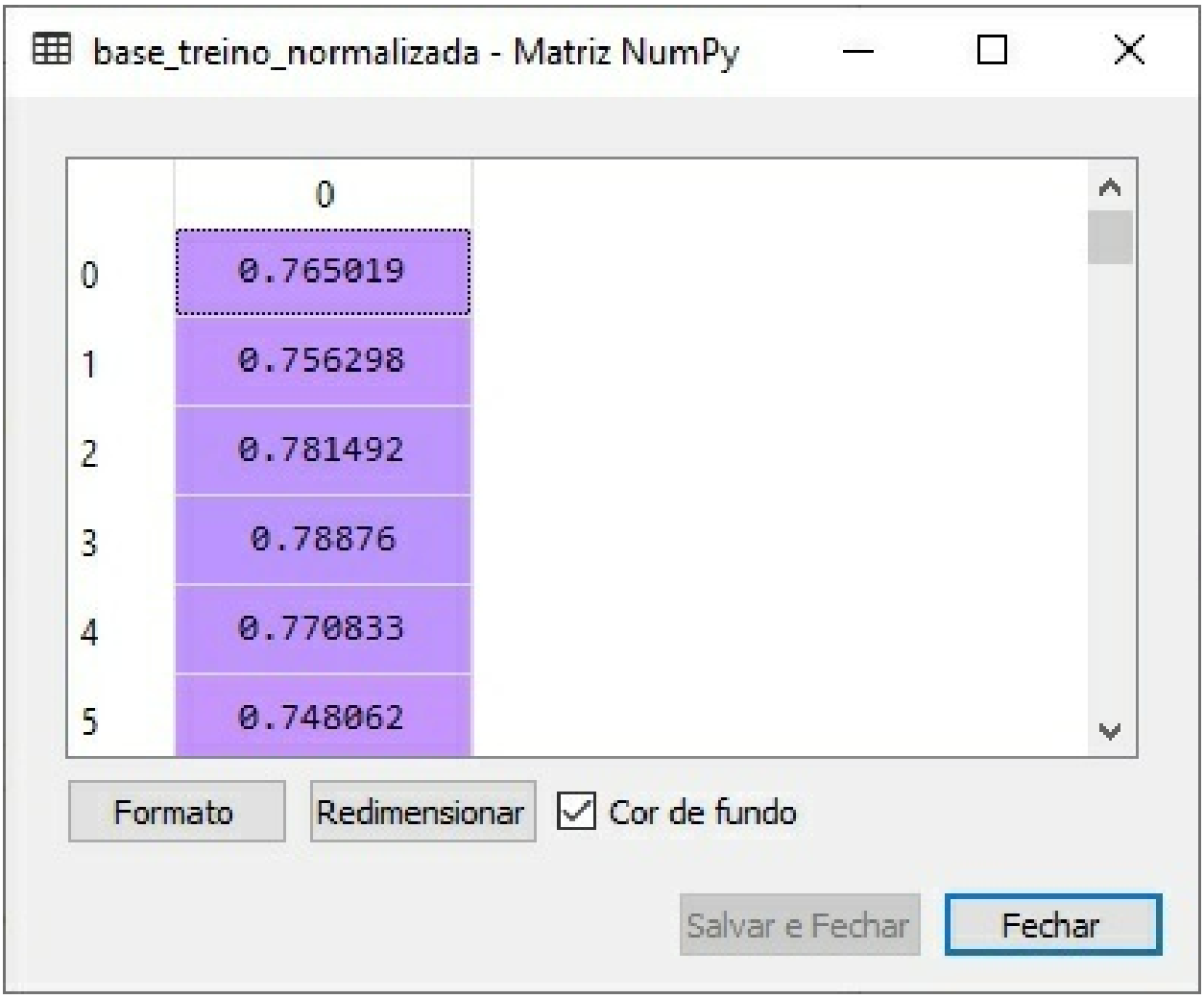
```
12 normalizador = MinMaxScaler(feature_range = (0,1))
13 base_treino_normalizada = normalizador.fit_transform(base_treino)
```

Em seguida realizaremos uma etapa de polimento desses dados bastante comum a este tipo de rede neural, onde normalizaremos os dados dentro de um intervalo numérico que fique mais fácil a visualização dos mesmos assim como a aplicação de certas funções.

Para isso criamos uma variável de nome normalizador que inicializa a ferramenta MinMaxScaler, nela, passamos como parâmetro feature_range = (0,1), o que em outras palavras significa que por meio dessa ferramenta transformaremos nossos dados em um intervalo entre 0 e 1, números entre 0 e 1.

Logo após criamos nossa variável base_treino_normalizada que aplica essa transformação sobre os dados de base_treino por meio da função

.fit_transform( ).



Selecionado e executado o bloco de código anterior podemos visualizar agora nossa base de dados de treino normalizados como números do tipo float32, categorizados em um intervalo entre 0 e 1.

```
previsores = []
preco_real = []

for i in range(90,1242):
    previsores.append(base_treino_normalizada[i-90:i,0])
    preco_real.append(base_treino_normalizada[i,0])

previsores, preco_real = np.array(previsores), np.array(preco_real)
previsores = np.reshape(previsores,(previsores.shape[0], previsores.shape[1],1
```

Uma vez terminado o polimento de nossos dados de entrada, podemos realizar as devidas separações e conversões para nossos dados de

previsão e de saída, que faremos a comparação posteriormente para avaliação da performance do modelo. Inicialmente criamos as variáveis previsores e preco_real com listas vazias como atributo. Em seguida criamos um laço de repetição que percorrerá nossa variável base_treino_normalizada separando os primeiros 90 registros (do número 0 ao 89) para que a partir dela seja encontrado o padrão que usaremos para nossas previsões.

Por fim, ainda dentro desse laço, pegamos os registros de nº 90 para usarmos como parâmetro, como preço real a ser previst pela rede. Fora do laço, realizamos as conversões de formato, transformando tudo em array do tipo numpy e definindo que a posição 0 na lista de previsores receberá os 1152 registros, enquanto a posição 1 receberá as informações referentes aos 90 dias usados como previsão.

Nome	Tipo	Tamanho	Valor
base	DataFrame	(1242, 7)	Column names: Date, Open, Hig Low, Close, Adj Close, Volume
base_treino	float64	(1242, 1)	[[19.99] [19.809999]]
base_treino_normalizada	float64	(1242, 1)	[[0.76501938] [0.7562984]]
i	int	1	1241
preco_real	float64	(1152,)	[0.76114341 0.76114341 0.7747
previsores	float64	(1152, 90, 1)	[[[0.76501938] [0.7562984]]

Uma vez encerrado o processo de tratamento dos dados, é possível conferir cada variável assim como seus valores via explorador de variáveis, e mais importante que isso, agora podemos dar início a criação do nosso modelo de rede neural recorrente.

```
25 from keras.layers import LSTM
26
```

Como de praxe, não será necessário criar manualmente toda a estrutura de nossa rede neural, mas criar a partir de uma ferramenta que possui essa estrutura pré-pronta e configurada. Par isso, realizamos a importação da ferramenta LSTM do módulo layers da biblioteca keras.

```
regressor = Sequential()
regressor.add(LSTM(units = 100,
                   return_sequences = True,
                   input_shape = (previsores.shape[1],1)))
regressor.add(Dropout(0.3))
regressor.add(LSTM(units = 50,
                   return_sequences = True,))
regressor.add(Dropout(0.3))
regressor.add(LSTM(units = 50))
regressor.add(Dropout(0.3))
regressor.add(Dense(units = 1,
                    activation = 'linear'))
```

Dando início a codificação da estrutura da rede, criamos inicialmente nossa variável regressor que inicializa Sequential sem parâmetros. Na sequência adicionamos a camada LSTM, que por sua vez tem como parâmetros input_shape que recebe os dados referentes aos registros e intervalo de tempo de nossa variável previsores, retornando essa sequência (recorrência) via parâmetro return_sequences definido como True (False será sem recorrência, em outras palavras, sem atualização de dados no próprio neurônio da camada) e por fim units especificando quantos neurônios estarão na primeira camada oculta.

Em seguida é adicionada a primeira camada de Dropout, que por sua vez, descartará 30% dos neurônios dessa camada aleatoriamente. Logo após criamos mais uma camada LSTM onde há recorrência, agora com tamanho de 50 neurônios. A seguir é adicionada mais uma camada Dropout, descartando mais 30% dos neurônios da camada, mais uma camada LSTM agora sem recorrência, ou seja, os valores que preencherem os neurônios dessa camada agora são estáticos, servindo apenas como referência para camada subsequente.

Por fim é criada uma camada Dense, onde há apenas 1 neurônio de saída e seu método de ativação é ‘linear’, que em outras palavras é uma função de ativação nula, o valor que chegar até esse neurônio simplesmente será replicado.

```
regressor.compile(optimizer = 'rmsprop',
                  loss = 'mean_squared_error',
                  metrics = ['mean_absolute_error'])
regressor.fit(previsores,
              preco_real,
              epochs = 100,
              batch_size = 32)
```

Como de costume, criadas as camadas estruturais da rede, hora de criar a camada de compilação da mesma e a que a alimentará, colocando a rede em execução. Sendo assim criamos nossa camada de compilação onde como parâmetros temos optimizer = ‘rsmprop’, método de otimização muito usado nesses tipos de modelo em função de oferecer uma descida de gradiente bastante precisa quando trabalhada uma série temporal, loss definimos como ‘mean_squared_error’, já usada em outro modelo, onde os valores de erro são elevados ao quadrado, dando mais peso aos erros e buscando assim melhor precisão, por fim metrics é parametrizado com nosso já conhecido ‘mean_absolute_error’. Para alimentação da rede via camada .fit( ) passamos todos os dados contidos em previsores, preco_real, assim como definimos que a rede será executada 100 vezes, atualizando seus pesos de 32 em 32 registros.

```
mean_absolute_error: 0.0288
Epoch 97/100
1152/1152 [==============================] - 8s 7ms/step - loss: 0.0013 -
mean_absolute_error: 0.0275
Epoch 98/100
1152/1152 [==============================] - 8s 7ms/step - loss: 0.0013 -
mean_absolute_error: 0.0267
Epoch 99/100
1152/1152 [==============================] - 8s 7ms/step - loss: 0.0014 -
mean_absolute_error: 0.0275
Epoch 100/100
1152/1152 [==============================] - 8s 7ms/step - loss: 0.0014 -
mean_absolute_error: 0.0273
Out[  ]:
```

Selecionado e executado o bloco de código anterior veremos a rede sendo processada, após o término da mesma podemos ver que em suas épocas finais, graças ao fator de dar mais peso aos erros, a mesma terminou com uma margem de erro próxima a zero.

```
48 base_teste = pd.read_csv('petr4-teste.csv')
49 preco_real_teste = base_teste.iloc[:, 1:2].values
50 base_completa = pd.concat((base['Open'], base_teste['Open']), axis = 0)
```

Rede treinada hora de começarmos a realizar as devidas previsões a partir da mesma, no caso, a previsão do preço de uma ação. Para isso, inicialmente vamos realizar alguns procedimentos para nossa base de dados.

Primeiramente criamos uma variável de nome base_teste que recebe todo conteúdo importado de nosso arquivo petr4-teste.csv. Da mesma

forma criamos nossa variável preco_real_teste que recebe todos os valores de todas as linhas e da coluna 1 de base_teste via método .iloc[ ].values.

Por fim criamos a variável base_completa que realiza a junção dos dados da coluna ‘Open’ de base e base teste por meio da função .concat( ).

Nome	Tipo	Tamanho	Valor
base	DataFrame	(1242, 7)	Column names: Date, Open, Hig Low, Close, Adj Close, Volume
base_completa	Series	(1264,)	Series object of pandas.core.series module
base_teste	DataFrame	(22, 7)	Column names: Date, Open, Hig Low, Close, Adj Close, Volume
base_treino	float64	(1242, 1)	[[19.99] [19.809999]
base_treino_normalizada	float64	(1242, 1)	[[0.76501938] [0.7562984]
i	int	1	1241
preco_real	float64	(1152,)	[0.76114341 0.76114341 0.7747
preco_real_teste	float64	(22, 1)	[[16.190001] [16.49]

Via explorador de variáveis é possível visualizar todos dados de todas variáveis que temos até o momento.

```
52 entradas = base_completa[len(base_completa) - len(base_teste) - 90:].values
53 entradas = entradas.reshape(-1,1)
54 entradas = normalizador.transform(entradas)
```

Dando continuidade, criamos uma variável de nome entradas que de forma parecida com a que fizemos anteriormente dentro de nosso laço de repetição i, recebe como atributos os últimos 90 registros (que serão usados para encontrar o padrão para previsão), em seguida são feitas as transformações de formato par cruzamento de dados. Nada muito diferente do que já fizemos algumas vezes em outros modelos.

```
56 previsores_teste = []
57 for j in range(90,112):
58     previsores_teste.append(entradas[j-90:j, 0])
59 previsores_teste = np.array(previsores_teste)
60 previsores_teste = np.reshape(previsores_teste,
61                               (previsores_teste.shape[0],
62                                previsores_teste.shape[1],
63                                1))
64 previsores_teste = regressor.predict(previsores_teste)
65 previsores_teste = normalizador.inverse_transform(previsores_teste)
```

Para criação de nossa estrutura previsora teríamos diversas opções de estrutura, seguindo como fizemos em outros modelos, vamos nos ater ao método mais simples e eficiente. Para uma série temporal, primeiro criamos uma variável de nome previsores_teste que recebe uma lista vazia como atributo, em seguida criamos um laço de repetição que irá percorrer, selecionar e copiar para previsores_teste os 90 primeiros registros de entradas (do 0 ao 89) através da função .append( ).

Na sequência realizamos as devidas transformações para array numpy e no formato que precisamos para haver compatibilidade de cruzamento. Por fim realizamos as previsões como de costume, usando a função .predict( ).

Repare na última linha deste bloco, anteriormente havíamos feito a normalização dos números para um intervalo entre 0 e 1, agora após realizadas as devidas previsões, por meio da função .inverse_transform( ) transformaremos novamente os dados entre 0 e 1 para seu formato original, de preço de ações.

Nome	Tipo	Tamanho	Valor
base_treino_normalizada	float64	(1242, 1)	[[0.76501938] [0.7562984]
entradas	float64	(112, 1)	[[0.47141473] [0.46317829]
i	int	1	1241
j	int	1	111
preco_real	float64	(1152,)	[0.76114341 0.76114341 0.774
preco_real_teste	float64	(22, 1)	[[16.190001] [16.49]
previsores	float64	(1152, 90, 1)	[[[0.76501938] [0.7562984]
previsores_teste	float32	(22, 1)	[[0.55642956] [0.5593487]

Repare que previsores_teste aqui está com valores normalizados.

Nome	Tipo	Tamanho	Valor
base_treino_normalizada	float64	(1242, 1)	[[0.76501938] [0.7562984]
entradas	float64	(112, 1)	[[0.47141473] [0.46317829]
i	int	1	1241
j	int	1	111
preco_real	float64	(1152,)	[0.76114341 0.76114341 0.774
preco_real_teste	float64	(22, 1)	[[16.190001] [16.49]
previsores	float64	(1152, 90, 1)	[[[0.76501938] [0.7562984]
previsores_teste	float32	(22, 1)	[[15.684706] [15.744957]

Após a aplicação da função .inverse_transform( ) os dados de previsores_teste voltam a ser os valores em formato de preço de ação.

```
preco_real_teste.mean()
previsores_teste.mean()
```

Como já realizado em outros modelos, aqui também se aplica fazer o uso de média dos valores, por meio da função .mean( ) tanto dos valores previstos quanto dos valores reais.

```
In [  ]: previsores_teste.mean()
Out[  ]: 17.157213

In [  ]: preco_real_teste.mean()
Out[  ]: 17.87454563636364
```

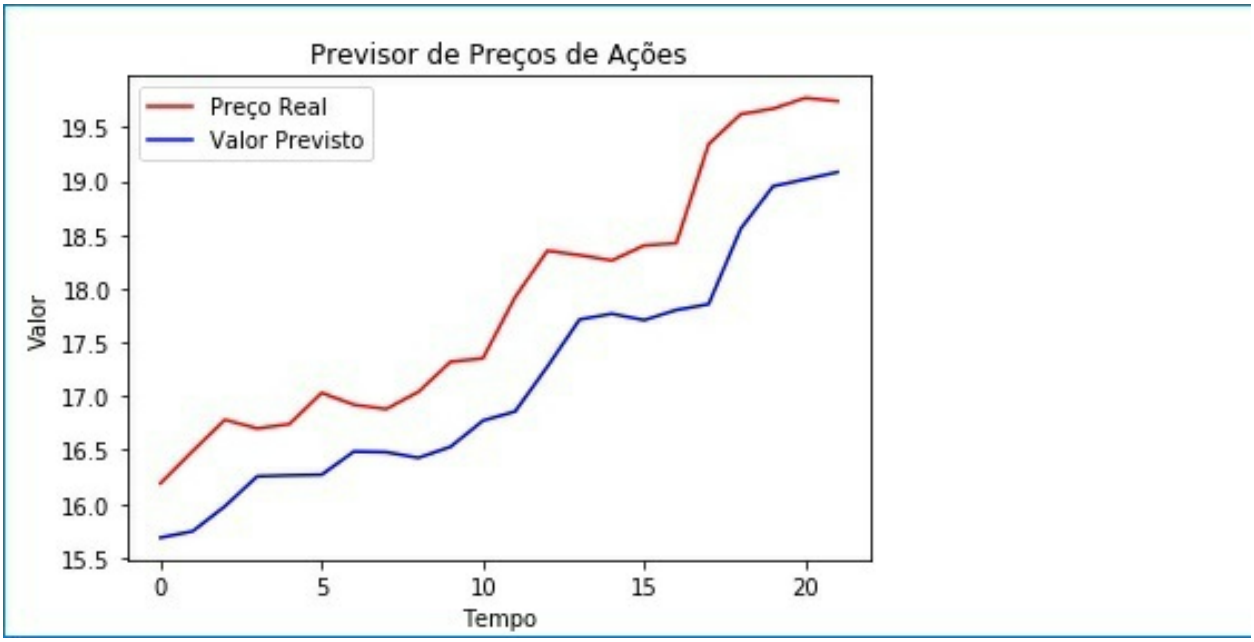
Selecionado e executado o bloco de código anterior podemos ver via console os valores retornados, note que o valor real da ação era de R$ 17,87, enquanto o valor previsto pela rede foi de R$ 17,15, algo muito próximo, confirmando a integridade de nossos dados assim como a eficiência de nossa rede.

Já que estamos trabalhando em cima de um exemplo de ações, quando vemos qualquer aplicação deste tipo de atividade costumamos ver percentuais ou gráficos apresentados para uma visualização mais clara, aqui podemos fazer a plotagem de nossos resultados para ter um retorno mais interessante.

```
71 import matplotlib.pyplot as plt
72 plt.plot(preco_real_teste,
73          color = 'red',
74          label = 'Preço Real')
75 plt.plot(previsores_teste,
76          color = 'blue',
77          label = 'Valor Previsto')
78 plt.title('Previsor de Preços de Ações')
79 plt.xlabel('Tempo')
80 plt.ylabel('Valor')
81 plt.legend()
82 plt.show()
```

Como de praxe, para plotagem de nossas informações em forma de gráfico importamos a biblioteca matplotlib. De forma bastante simples, por meio da função .plot( ) passamos como parâmetro os dados de preco_real_teste e de previsores_teste, com cores diferentes para fácil visualização assim como um rótulo para cada um. Também definimos um título para nosso gráfico assim como rótulos para os planos X e Y, respectivamente.

Por fim por meio da função .show( ) o gráfico é exibido no console.



Agora podemos visualizar de forma bastante clara a apresentação de nossos dados, assim como entender o padrão. Note a semelhança entre a escala dos valores reais em decorrência do tempo e dos dados puramente previstos pela rede neural. Podemos deduzir que esse modelo é bastante confiável para este tipo de aplicação.

Código Completo:

```
import numpy as np
import pandas as pd
from keras.models import Sequential
from keras.layers import Dense, Dropout
from sklearn.preprocessing import MinMaxScaler
from keras.layers import LSTM
import matplotlib.pyplot as plt

base = pd.read_csv('petr4-treinamento.csv')
base = base.dropna()
base_treino = base.iloc[:, 1:2].values
normalizador = MinMaxScaler(feature_range = (0,1))
base_treino_normalizada = normalizador.fit_transform(base_treino)

previsores = []
preco_real = []

for i in range(90,1242):
    previsores.append(base_treino_normalizada[i-90:i,0])
    preco_real.append(base_treino_normalizada[i,0])

previsores, preco_real = np.array(previsores), np.array(preco_real)
previsores = np.reshape(previsores,(previsores.shape[0], previsores.shape[1],1
```

```
regressor = Sequential()
regressor.add(LSTM(units = 100,
                   return_sequences = True,
                   input_shape = (previsores.shape[1],1)))
regressor.add(Dropout(0.3))
regressor.add(LSTM(units = 50,
                   return_sequences = True,))
regressor.add(Dropout(0.3))
regressor.add(LSTM(units = 50))
regressor.add(Dropout(0.3))
regressor.add(Dense(units = 1,
                    activation = 'linear'))
regressor.compile(optimizer = 'rmsprop',
                  loss = 'mean_squared_error',
                  metrics = ['mean_absolute_error'])
regressor.fit(previsores,
              preco_real,
              epochs = 100,
              batch_size = 32)
```

```
base_teste = pd.read_csv('petr4-teste.csv')
preco_real_teste = base_teste.iloc[:, 1:2].values
base_completa = pd.concat((base['Open'], base_teste['Open']), axis = 0)

entradas = base_completa[len(base_completa) - len(base_teste) - 90:].values
entradas = entradas.reshape(-1,1)
entradas = normalizador.transform(entradas)

previsores_teste = []
for j in range(90,112):
    previsores_teste.append(entradas[j-90:j, 0])
previsores_teste = np.array(previsores_teste)
previsores_teste = np.reshape(previsores_teste,
                              (previsores_teste.shape[0],
                               previsores_teste.shape[1],
                               1))
previsores_teste = regressor.predict(previsores_teste)
previsores_teste = normalizador.inverse_transform(previsores_teste)
previsores_teste.mean()
preco_real_teste.mean()
```

```
plt.plot(preco_real_teste,
         color = 'red',
         label = 'Preço Real')
plt.plot(previsores_teste,
         color = 'blue',
         label = 'Valor Previsto')
plt.title('Previsor de Preços de Ações')
plt.xlabel('Tempo')
plt.ylabel('Valor')
plt.legend()
plt.show()
```

OUTROS MODELOS DE REDES NEURAIS ARTIFICIAIS

Nos capítulos anteriores percorremos uma boa parte do que temos de modelos de redes neurais. Classificadores, regressores, identificadores, previsores, etc... cada modelo com suas pequenas particularidades, mas todos integrando um contexto parecido.

Agora vamos dedicar um pequeno tempo de nossos estudos a modelos de rede que não se encaixam nos moldes anteriores devido suas adaptações para resolução de problemas computacionais bastante específicos.

Mapas Auto Organizáveis – Kmeans – Vinhos Dataset

Um dos modelos mais simples dedicado ao agrupamento de objetos de mesmo tipo é o chamado modelo de mapa auto organizável. Você notará uma semelhança conceitual com o modelo estatístico KNN visto lá no início do livro, porém aqui a rede usa de alguns mecanismos para automaticamente reconhecer padrões e realizar o agrupamento de objetos inteiros (de várias características).

Em algumas literaturas este modelo também é chamado Kmeans, onde existe um referencial k e com base nele a rede reconhece padrões dos objetos próximos, trazendo para o agrupamento se estes objetos possuírem características em comum, e descartando desse agrupamento objetos que não compartilham das mesmas características, dessa forma, após um tempo de execução da rede os objetos estarão separados e agrupados de acordo com sua semelhança.

Para entendermos na prática como esse modelo funciona usaremos uma base de dados disponível na UCI onde teremos de classificar amostras de vinho quanto sua semelhança, a base foi construída com base em 178 amostras, de 3 cultivadores diferentes, onde temos 13 características para definição de cada tipo de vinho, sendo assim, nossa rede irá processar tais características, identificar e agrupar os vinhos que forem parecidos.

UCI Machine Learning Repository

Center for Machine Learning and Intelligent Systems

Wine Data Set

Download: Data Folder, Data Set Description

Abstract: Using chemical analysis determine the origin of wines



Data Set Characteristics:	Multivariate	**Number of Instances:**	178	**Area:**	Physical
Attribute Characteristics:	Integer, Real	**Number of Attributes:**	13	**Date Donated**	1991-07-01
Associated Tasks:	Classification	**Missing Values?**	No	**Number of Web Hits:**	1209764

Feito o download da base de dados podemos partir diretamente para prática.

Vinhos.py

```
1 import pandas as pd
2 from minisom import MiniSom
```

Como sempre, o processo se inicia com as importações dos módulos, bibliotecas e ferramentas que usaremos. Dessa forma importamos a biblioteca pandas e MiniSom da biblioteca minisom.

```
4 base = pd.read_csv('wines.csv')
5 entradas = base.iloc[:, 1:14].values
6 saidas = base.iloc[:, 0].values
```

Em seguida criamos nossas variáveis de base, entradas e saidas, importando todo conteúdo de wines.csv e dividindo essa base em atributos previsores e de saída respectivamente. O processo segue os mesmos moldes já utilizados anteriormente em outros modelos.

Nome	Tipo	Tamanho	Valor
base	DataFrame	(178, 14)	Column names: Class, Alcohol, Malic acid, Ash, Alcalinity of ash, Ma ...
entradas	float64	(178, 13)	[[1.423e+01 1.710e+00 2.430e+00 ... 1.040e+00 3. [1 ...
saidas	int64	(178,)	[1 1 1 ... 3 3 3]

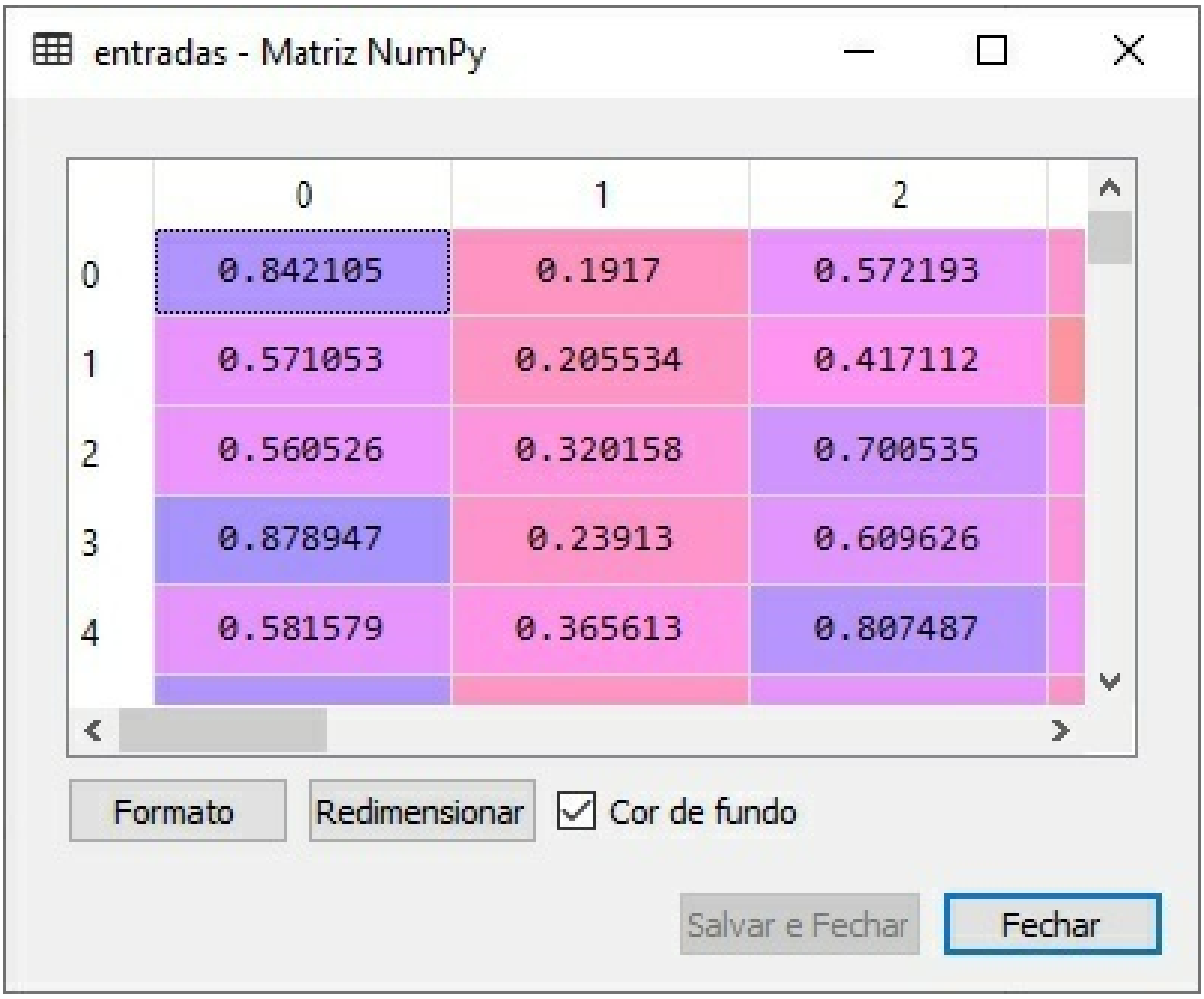
Se o processo de importação ocorreu normalmente temos à disposição as variáveis para visualização via explorador de variáveis.

```
8 from sklearn.preprocessing import MinMaxScaler
9
```

Logo após realizamos a importação da ferramenta MinMaxScaler, do módulo preprocessing da biblioteca sklearn. Lembrando que essa ferramenta é usada para conversão de números para um intervalo definido entre um número e outro (ex: de 0 até 1).

```
10 normalizador = MinMaxScaler(feature_range = (0,1))
11 entradas = normalizador.fit_transform(entradas)
```

Na sequência é realizado o processo de normalização nos mesmos moldes de sempre, por meio de uma variável dedicada a inicializar a ferramenta e definir um intervalo, nesse caso entre 0 e 1 mesmo, em seguida atualizando os dados da própria variável entradas.



Selecionado o bloco de código anterior e executado, podemos ver que de fato agora todos valores de entradas pertencem a um intervalo entre 0 e 1, pode parecer besteira, mas nessa escala temos um grande ganho de performance em processamento da rede.

```
som = MiniSom(x = 8,
              y = 8,
              input_len = 13,
              sigma = 1.0,
              learning_rate = 0.5,
              random_seed = 2)
som.random_weights_init(entradas)
som.train_random(data = entradas,
                 num_iteration = 100)
som._weights
som._activation_map
agrupador = som.activation_response(entradas)
```

Apenas um adendo, nosso agrupador MiniSom possui uma lógica

de funcionamento que você não encontrará facilmente em outras literaturas. Quando estamos falando de um mecanismo agrupador, é fundamental entender o tamanho do mesmo, uma vez que ele aplicará uma série de processos internos para realizar suas funções de identificação e agrupamento. Para definirmos o tamanho da matriz a ser usada na ferramenta devemos fazer um simples cálculo. 5 * √n , onde n é o número de entradas.

Nesse nosso exemplo, temos 178 entradas, dessa forma, 5 * √178 resulta 5 * 13.11 = 65 células, arredondando esse valor para um número par 64, podemos criar uma estrutura de matriz SOM de 8x8.

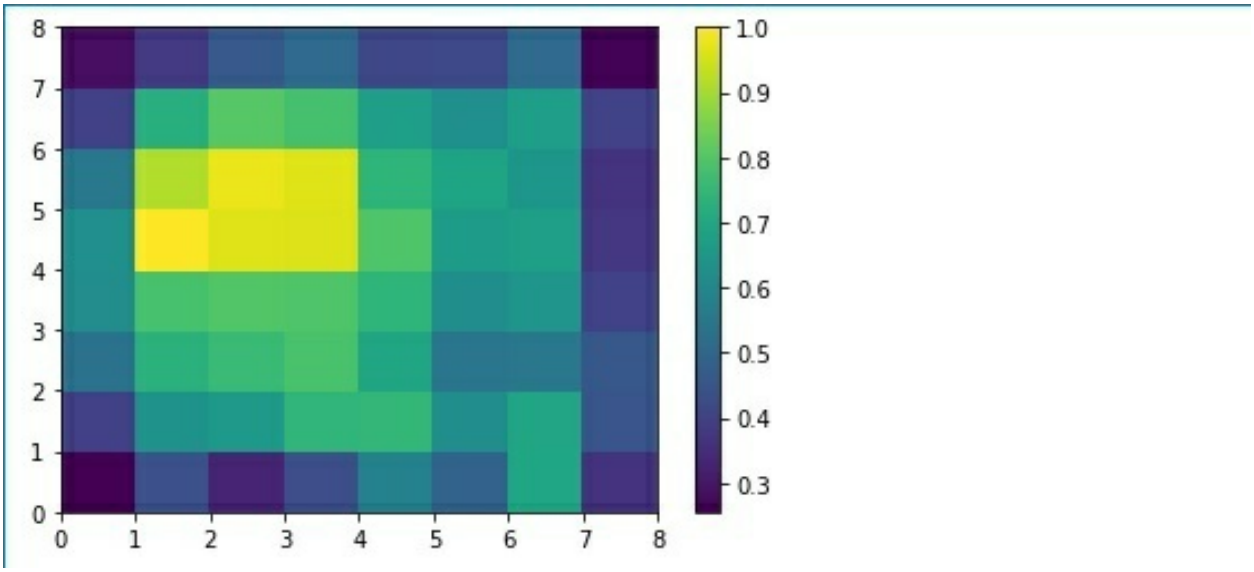
Uma vez feito o tratamento de nossos dados hora de criar a estrutura de nosso agrupador. Como estaremos usando uma ferramenta pré-pronta e pré-configurada, tudo o que temos de fazer é definir os parâmetros de tamanho, processamento e alimentação da ferramenta. Inicialmente criamos uma variável som que inicializa MiniSom passando alguns parâmetros, x e y definem o tamanho da matriz de agrupamento, input_len define quantas amostras serão usadas na entrada, sigma define a distância de proximidade entre as características das amostras, learning_rate parametriza o modo como serão atualizados os pesos e random_seed por sua vez define como se dará a alimentação da matriz para que sejam realizados os testes de proximidade, dessa vez atualizando as amostras. Na sequência são inicializados pesos com números aleatórios para entradas por meio da função .random_weights_init( ).

Em seguida é aplicada a função .train_random( ) que recebe como parâmetro os dados de entradas enquanto num_iterations define quantos testes de proximidade serão realizados entre as amostras, para cada amostra, a fim de encontrar os padrões de similaridade para agrupamento das mesmas. Para finalizar o modelo é inicializado _weights e _activation_map, finalizando criamos a variável agrupador que por meio da função .activation_response( ) parametrizado com entradas, criará nossa matriz de dados de agrupamento.

```
26 from pylab import pcolor, colorbar
27
28 pcolor(som.distance_map().T)
29 colorbar()
```

Por fim, para exibir nossa matriz de agrupamento de forma visual, importamos as ferramentas pcolor e colorbar da biblioteca pylab. Executando

a ferramenta pcolor, parametrizada com som.distance_map( ).T teremos uma representação visual do agrupamento. Executando colorbar( ) teremos em anexo uma barra de cores que auxiliará na identificação dos tipos de vinho quanto sua proximidade.



Selecionado e executado o bloco de código anterior podemos finalmente ter uma representação visual de nossos dados, identificando facilmente por exemplo que os blocos pintados em amarelo são de um determinado tipo de vinho, diferente dos demais, assim como os outros agrupamentos de acordo com a cor.

Código Completo:

```
import pandas as pd
from minisom import MiniSom
from sklearn.preprocessing import MinMaxScaler
from pylab import pcolor, colorbar

base = pd.read_csv('wines.csv')
entradas = base.iloc[:, 1:14].values
saidas = base.iloc[:, 0].values

normalizador = MinMaxScaler(feature_range = (0,1))
entradas = normalizador.fit_transform(entradas)

som = MiniSom(x = 8,
              y = 8,
              input_len = 13,
              sigma = 1.0,
              learning_rate = 0.5,
              random_seed = 2)
som.random_weights_init(entradas)
som.train_random(data = entradas,
                 num_iteration = 100)
som._weights
som._activation_map
agrupador = som.activation_response(entradas)

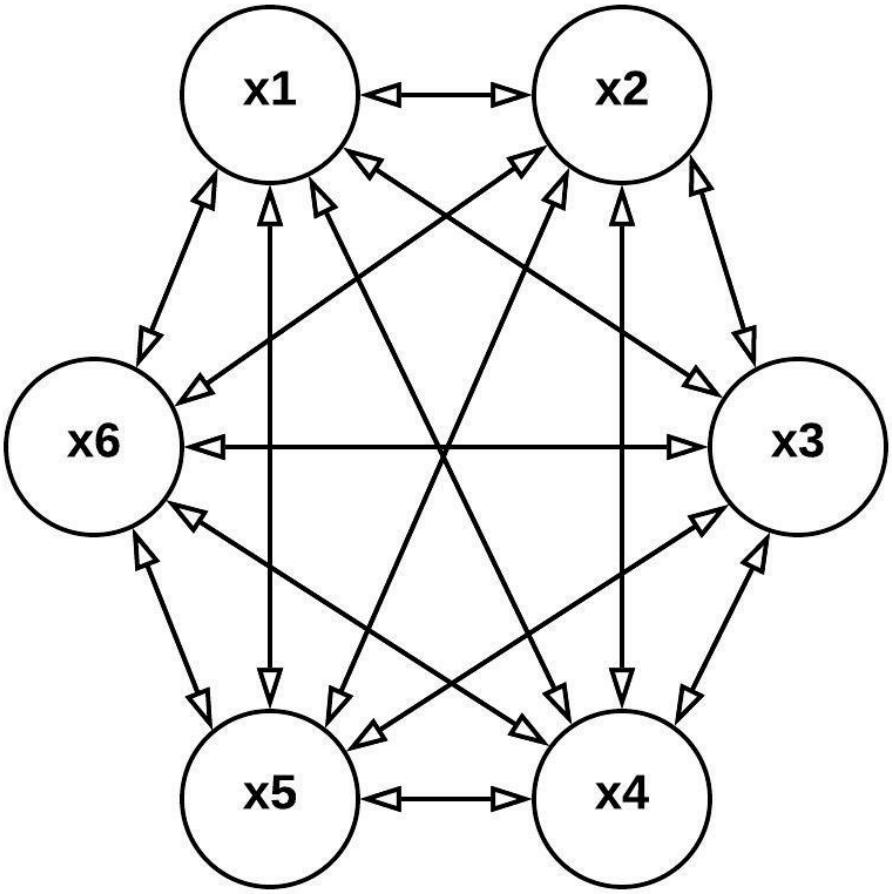
pcolor(som.distance_map().T)
colorbar()
```

Sistemas de Recomendação – Boltzmann Machines

Boltzmann Machines são modelos de redes neurais que são comumente usadas em sistemas de recomendação. Em algum momento você já deve ter terminado de assistir um vídeo no YouTube, Netflix ou até mesmo no Facebook e se deparou ao final do vídeo com uma série de sugestões apresentadas com base no vídeo que você acabou de assistir, muitas vezes essas recomendações parecem não fazer sentido, mas em boa parte das vezes ela realmente recomenda algo de relevância com base na sua última experiência, isto se dá por um modelo de rede neural bem característico, capaz de aprender os padrões de escolha do usuário e fazer recomendações.

Uma máquina de recomendação possui algumas particularidades, o primeiro grande destaque é o fato de as células de entrada, nesse tipo de rede, fazer parte dos nós da rede, se atualizando junto com o processamento da mesma, pois não há uma camada de entrada definida como nos outros modelos, da mesma forma não existe uma camada específica de saída.

Em outros modelos havia uma fase supervisionada, onde apontávamos para a rede onde estava o erro para que ela aprendesse, neste modelo de rede neural não há supervisionamento. Dessa forma, raciocine que uma Boltzmann Machine é um tipo de rede onde todos neurônios estão conectados entre si, e não por camadas, assim a rede identifica e aprende qual é o comportamento padrão de um sistema para que possa identificar alterações, ou identificar qualquer coisa fora do padrão.



Em suma, uma Boltzmann Machine é um modelo que quando instanciado em uma determinada plataforma ou contexto, imediatamente irá identificar os objetos desse contexto (assim como suas características e classificações) e a partir daí ela reforça os padrões aprendidos e monitora qualquer padrão desconhecido.

Como exemplo vamos simular um sistema de recomendação de filme. Supondo que você é usuário de uma plataforma de streaming que ao final da visualização de um vídeo pede que você avalie o mesmo simplesmente dizendo se gostou ou se não gostou, com base nessas respostas o sistema de recomendação irá aprender que tipo de vídeo você gosta para que posteriormente possa fazer recomendações.

Boltzmann.py

```
1 from rbm import RBM
2 import numpy as np
```

Dando início criamos nosso arquivo Boltzmann.py e em seguida já realizamos as primeiras importações. Importamos a biblioteca numpy e a ferramenta RBM (Restricted Boltzmann Machine). Para que esse processo ocorra de forma correta é necessário o arquivo rbm.py estar na mesma paste de Boltzmann.py.

```
4 rbm = RBM(num_visible = 6,
5           num_hidden = 2)
```

Em seguida criamos nossa variável rbm que inicializa a ferramenta RBM lhe passando como parâmetros num_visible referente ao número de nós dessa rede (lembrando que aqui não são tratados como neurônios de uma camada de entrada...) e num_hidden que define o número de nós na camada oculta.

```
7 base = np.array([[1,1,1,0,0,0],
8                  [1,0,1,0,0,0],
9                  [1,1,1,0,0,0],
0                  [0,0,1,1,1,1],
1                  [0,0,1,1,0,1],
2                  [0,0,1,0,1,0]])
```

Na sequência criamos nossa variável base que tem como atributo uma array numpy onde estamos simulando 6 usuários diferentes (cada linha) e que filmes assistiu ou deixou de assistir respectivamente representado em 1 ou 0.

Nome	Tipo	Tamanho	Valor
base	int32	(6, 6)	[[1 1 1 0 0 0] [1 0 1 0 0 0]

Se o processo foi feito corretamente, sem erros de sintaxe, a devida variável é criada.

```
14 filmes = ['O Exorcista',
15           'American Pie',
16           'Matrix',
17           'Forrest Gump',
18           'Documentário X',
19           'O Rei Leão']
```

Logo após é criada nossa variável filmes que recebe em formato de

lista o nome de 6 filmes, aqui neste exemplo, apenas diferenciados quanto ao seu estilo. Um de terror, um de comédia, um de ficção científica, um drama, um documentário e uma animação.

```
21 rbm.train(base,
22           max_epochs = 3000)
23 rbm.weights
```

Na sequência executamos duas funções de nossa rbm, .train( ) parametrizado com os dados de nossa base e max_epochs que define quantas vezes ocorrerão as atualizações dos pesos. Por fim são inicializados os pesos.

```
Epoch 2990: error is 1.1867429671954914
Epoch 2991: error is 1.1867275151709342
Epoch 2992: error is 2.4910594588085955
Epoch 2993: error is 1.187052233844001
Epoch 2994: error is 1.18702346961957
Epoch 2995: error is 1.18699579300992
Epoch 2996: error is 1.1869691220316294
Epoch 2997: error is 1.7041098896921478
Epoch 2998: error is 1.1870667506429766
Epoch 2999: error is 1.1870341673638922
Out[  ]:
```

Selecionado e executado o bloco de código anterior é possível ver a rede sendo treinada via console assim como a margem de erro calculada.

```
array([[ 4.51019147,  1.28650412, -1.01177789],
       [-3.95724928,  1.26896524,  6.97861864],
       [-4.13637439,  0.68705828,  4.1564823 ],
       [ 6.83260098,  2.50333604,  0.04463753],
       [ 3.32804508, -6.63982934, -3.58508583],
       [ 0.02403081,  2.49566332, -6.73135502],
       [ 3.32697787, -6.63173354, -3.61145216]])
```

Da mesma forma é criada uma array com os dados dos pesos. Aqui a maneira correta de fazer a leitura dessa matriz é não considerar nem a primeira linha nem a primeira coluna pois esses são dados de bias, a leitura é feita a partir da segunda linha da segunda coluna em diante. Dessa forma a rmb encontrará os padrões que dão característica a cada tipo de filme, por exemplo na segunda linha, referente ao filme O Exorcista, 1.26 e 6.97. A partir desse ponto, podemos simular um usuário e fazer a recomendação do próximo filme ao mesmo.

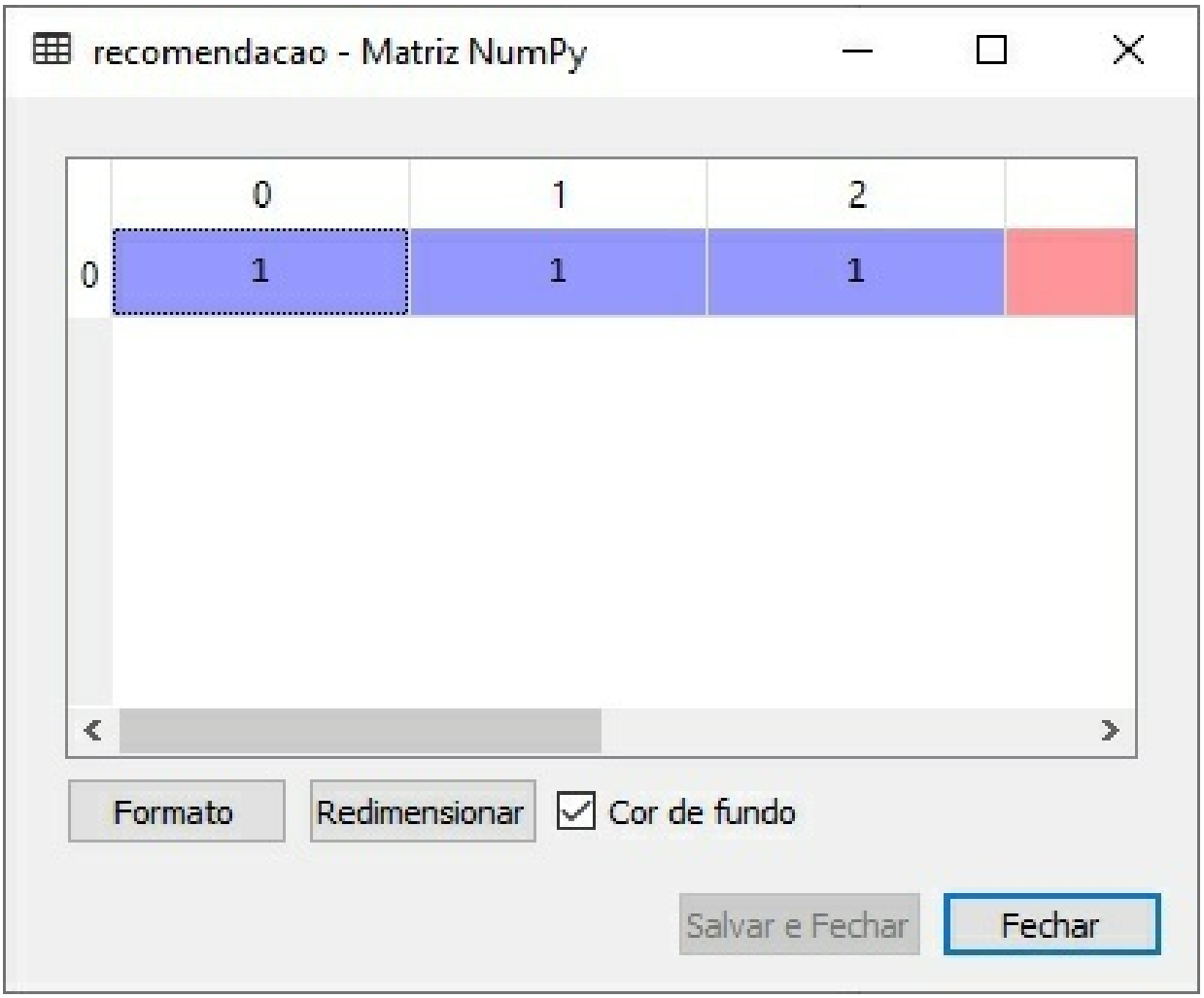
```
26 usuario = np.array([[1,1,0,1,0,0]])
27
```

Começando com a fase de previsão e recomendação, inicialmente criamos uma variável de nome usuario, que por sua vez, com base em sua array, assistiu O Exorcista, American Pie e Forrest Gump, deixando de assistir outros títulos (essa array em outro contexto também pode significar que o usuário assistiu todos os 6 filmes mas gostou apenas do nº 1, 2 e 4.

```
28 rbm.run_visible(usuario)
29 camada_oculta = np.array([[0,1]])
30 recomendacao = rbm.run_hidden(camada_oculta)
```

Em seguida criamos o mecanismo que fará a identificação do usuário (suas preferências). Para isso usamos a função .run_visible( ) parametrizada com os dados de usuario. Na sequência é criada a variável camada_oculta que possui uma array para classificação binária, ou um tipo de filme ou outro com base na proximidade de suas características.

Por fim é criada a variável recomendacao que roda a função .run_hidden( ) sobre camada_oculta. Desta maneira será identificado e recomendado um próximo filme com base nas preferências do usuário. Lembrando que aqui temos uma base de dados muito pequena, o ideal seria um catálogo de vários títulos para que se diferenciasse com maior precisão os estilos de filme.



Note que executado o bloco de código anterior, é criada a variável recomendacao onde podemos ver que há o registro do primeiro e do segundo filme, ao lado agora está ativado o registro correspondente ao terceiro filme, como o usuário ainda não viu este, essa coluna agora marcada como 1 será a referência para sugestão desse filme.

```
32 for i in range(len(usuario[0])):
33     print(usuario[0,i])
34     if usuario[0,i] == 0 and recomendacao[0,i] == 1:
35         print(filmes[i])
```

Para deixar mais claro esse processo, criamos um laço de repetição que percorre todos os registros de usuário (toda a linha), com base nisso é impresso o padrão das preferências desse usuário. Em seguida é criada uma estrutura condicional que basicamente verifica que se os registros do usuario forem igual a – e os registros de recomendacao na mesma posição da lista for igual a 1, é impresso o nome do filme desta posição da lista, como

recomendação.

Nome	Tipo	Tamanho	Valor
base	int32	(6, 6)	[[1 1 1 0 0 0] [1 0 1 0 0 0]
camada_oculta	int32	(1, 2)	[[0 1]]
filmes	list	6	['O Exorcista', 'American Pie', 'Matrix', 'Forr Gump', 'Documentári ...
i	int	1	5
recomendacao	float64	(1, 6)	[[1. 1. 1. 0. 0. 0.]]
usuario	int32	(1, 6)	[[1 1 0 1 0 0]]

Note que o processo executado dentro do laço de repetição confere com a projeção anterior.

```
1
1
0
Matrix
1
0
0
```

Sendo assim, é impresso o nome do filme a ser recomendado a este usuário com base em suas preferências, nesse caso, Matrix.

Código Completo:

```
from rbm import RBM
import numpy as np

rbm = RBM(num_visible = 6,
          num_hidden = 2)

base = np.array([[1,1,1,0,0,0],
                 [1,0,1,0,0,0],
                 [1,1,1,0,0,0],
                 [0,0,1,1,1,1],
                 [0,0,1,1,0,1],
                 [0,0,1,0,1,0]])

filmes = ['O Exorcista',
          'American Pie',
          'Matrix',
          'Forrest Gump',
          'Documentário X',
          'O Rei Leão']

rbm.train(base,
          max_epochs = 3000)
rbm.weights

usuario = np.array([[1,1,0,1,0,0]])

rbm.run_visible(usuario)
camada_oculta = np.array([[0,1]])
recomendacao = rbm.run_hidden(camada_oculta)

for i in range(len(usuario[0])):
    print(usuario[0,i])
    if usuario[0,i] == 0 and recomendacao[0,i] == 1:
        print(filmes[i])
```

REDES NEURAIS ARTIFICIAIS INTUITIVAS

BOT Para mercado de Ações

Para darmos início a nossas implementações, começaremos com problemas de menor complexidade. O exemplo a seguir tem o objetivo de demonstrar como que uma inteligência artificial é capaz de aprender a realizar corretagem no mercado de ações. Para isso, criaremos um **Bot** que por sua vez aprenderá a buscar de forma autônoma os dados de ações de uma determinada carteira, aprender seus padrões, analisar em tempo real as operações que estão sendo realizadas assim como tomar iniciativa de comprar ou vender uma determinada ação.

Conceitualmente pode parecer algo complexo, mas seguindo nossa lógica gradual, por hora raciocine que o que teremos de fazer é um agente que buscará as informações do mercado financeiro a partir da internet, criará uma base de dados em memória com informações dos padrões de atividade em geral (como em uma série temporal) e dos últimos lançamentos realizados, com base nisso, finalmente o agente realizará uma série de tentativas aprendendo a operar o mercado buscando obviamente, sempre obter lucro.

Importante deixar bem claro aqui que análise financeira e corretagem de ações é algo bastante complexo, que depende de uma série de fatores e nosso agente estará simulando as atividades de corretagem em forma de demonstração. Caso você queira usar este tipo de rede neural em suas aplicações de dinheiro real, recomendo fortemente que a rede neural apenas sirva como um viés de confirmação, apontando melhores momentos para se realizar ações de compra ou venda.

De forma resumida, e com propósito de organizar nossa sequência lógica dos fatos, basicamente o que faremos é:

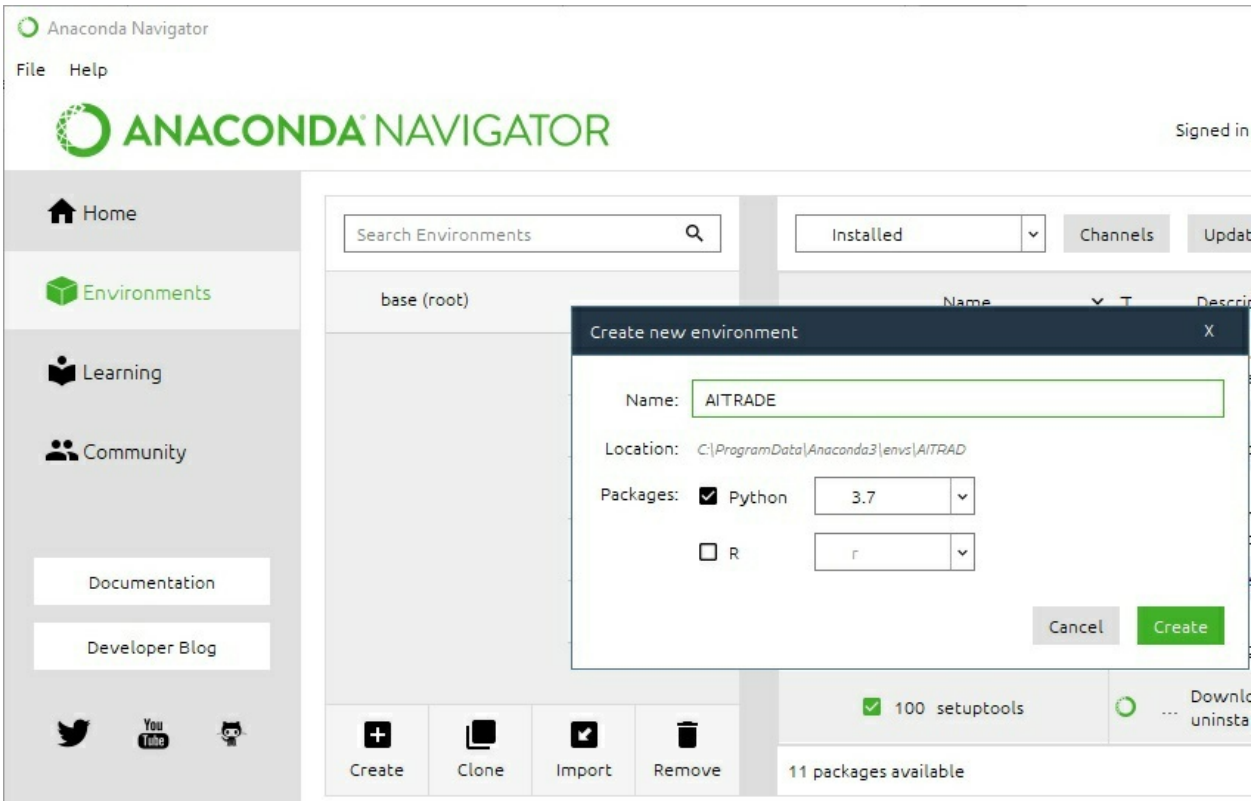
- Instalação das bibliotecas e módulos que serão utilizados em

nosso código.
- Importação das bibliotecas e módulos específicos.
- Construção da arquitetura de rede neural artificial intuitiva.
- Criação da base de memória e pré-processamento dos dados.
- Carregamento da memória de curto prazo.
- Criação dos estados iniciais de nosso agente.
- Treinamento da IA para seu devido propósito.
- Definição final do modelo.
- Execução de nossa rede neural artificial intuitiva.
- Interpretação dos dados obtidos em tempo real.

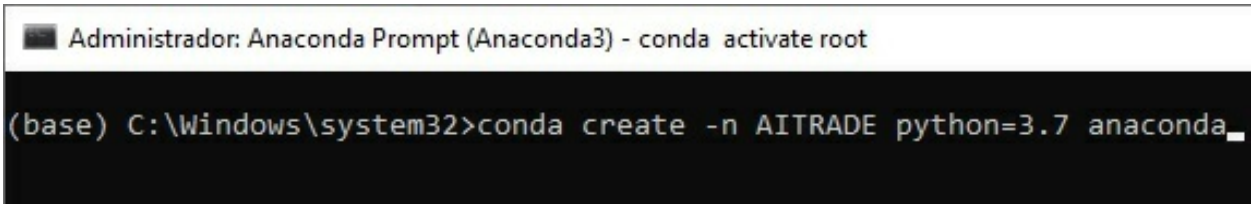
Partindo para a prática!!!

Para nossa rede neural artificial intuitiva, iremos criar um ambiente isolado do ambiente de virtualização padrão do Windows, dessa forma, teremos nele apenas o núcleo da linguagem Python e as bibliotecas que farão parte desse projeto, facilitando assim a exportação do mesmo para outra máquina.

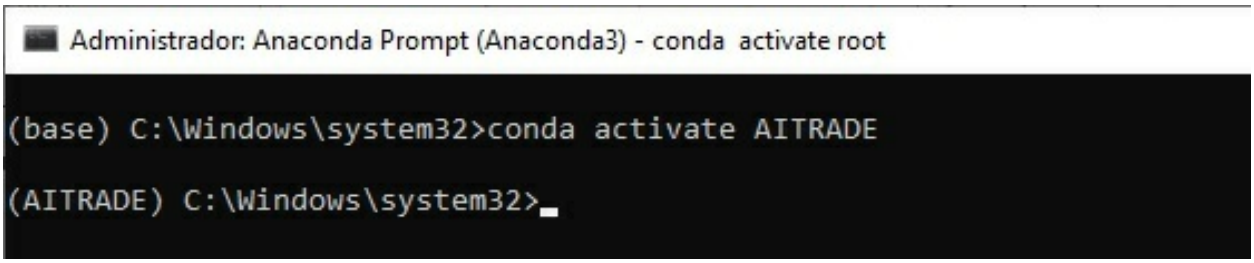
O processo de criar um ambiente virtual isolado pode ser feito de duas formas, via Anaconda Navigator ou via Prompt de Comando.



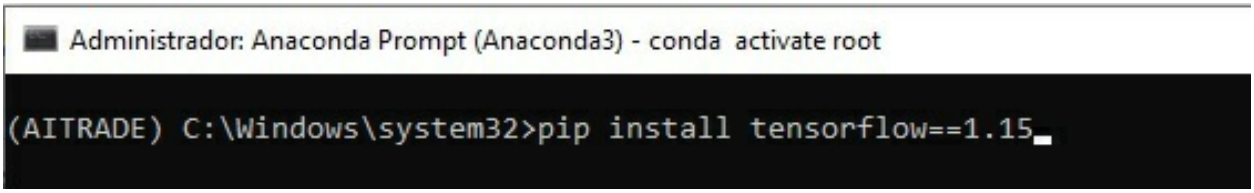
No Anaconda Navigator, abrindo a aba a esquerda Environments, em seguida clicando no botão Create. Na janela que irá surgir, definiremos um nome para nosso ambiente (no meu caso **AITRADE**), assim como a versão do *Python*. Clicando em Create basta esperar alguns minutos que o ambiente estará devidamente criado e pronto para ser ativado.



Via Prompt Anaconda, podemos criar o ambiente virtual de acordo com o comando mostrado na imagem, **conda create -n AITRADE python=3.7 anaconda**.



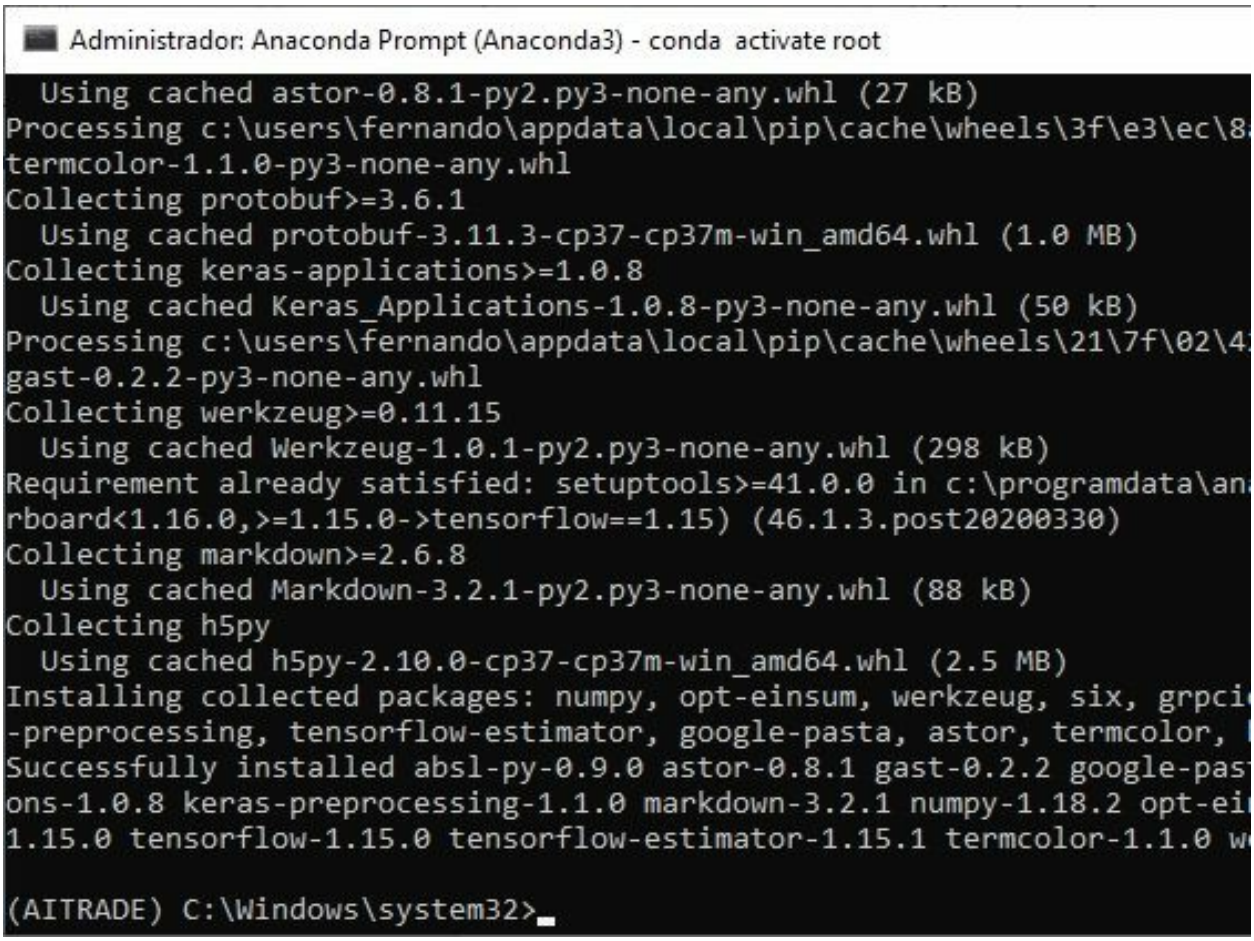
Em seguida para que possamos instalar as bibliotecas complementares no ambiente isolado recém criado precisamos ativar o mesmo via termina, isso é feito pelo comando **conda activate AITRADE**.



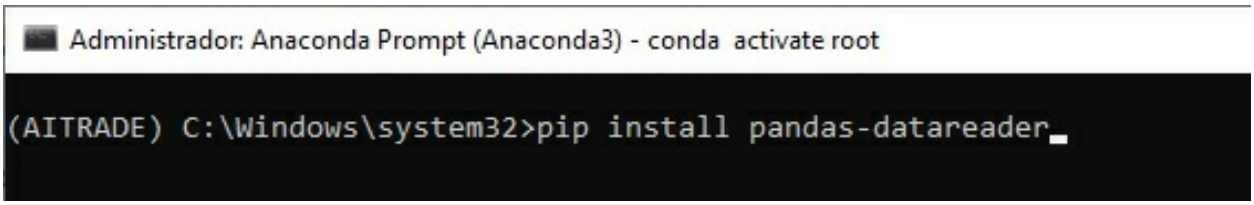
Na sequência, por meio do comando **pip install tensorflow==1.15** estamos instalando a referida versão em nosso ambiente isolado AITRADE do TensorFlow, biblioteca do Google que incorpora uma série de ferramentas otimizadas para criação de redes neurais artificiais.

Note que para este exemplo estamos usando a versão cpu 1.15, isto se dará porque a versão mais recente do TensorFlow (2.X) descontinuou algumas das funções que estaremos utilizando em nosso exemplo. Também foram realizados testes com a versão 2.0.0.alpha0 obtivendo apenas alguns avisos de funções a serem descontinuadas, tendo o código totalmente funcional.

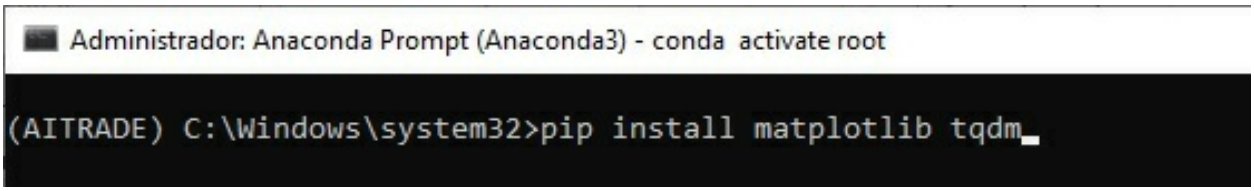
Sinta-se à vontade para instalar a versão gpu, caso você tenha uma placa de vídeo dedicada e obviamente queira uma melhor performance no processamento da rede.



Uma vez feita a instalação corretamente do TensorFlow e suas dependências, iremos instalar uma outra biblioteca independente chamada Pandas DataReader, módulo independentemente da biblioteca Pandas que nos permitirá a leitura das bases de dados a partir da internet e não apenas a partir de um arquivo local.

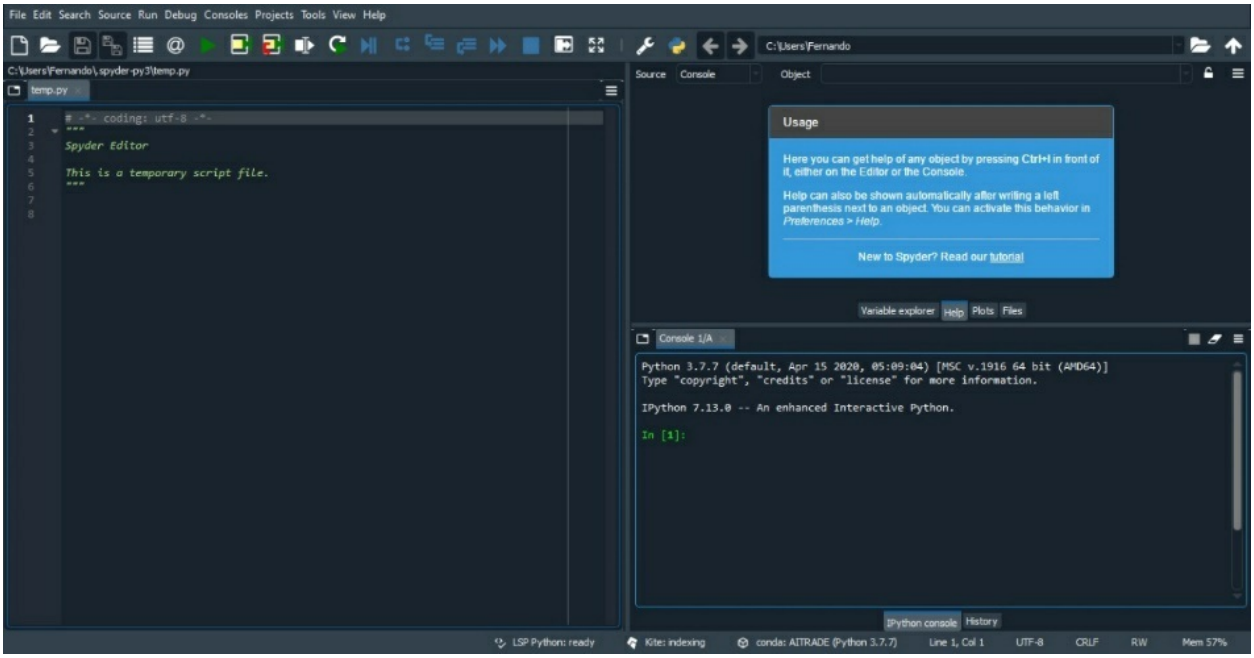


O processo é exatamente o mesmo da instalação anterior, apenas alterando o comando para **pip install pandas-datareader**.



Finalizando esta etapa, realizamos as instalações das duas últimas bibliotecas necessárias, matplotlib, responsável por oferecer plotagem de gráficos sobre os resultados de nossos processamentos e tqdm que servirá para criar barras de progresso mais intuitivas quando estivermos visualizando a execução de nossa rede neural.

Partindo para o código!!!



Uma última verificação antes de partirmos efetivamente para o código deve ser realizada, abrindo o Spyder, na barra inferior da janela da IDE verifique se a mesma está trabalhando no ambiente virtualizado correto (nesse caso AITRADE), se sim, podemos finalmente partir para o código.

```
1   import math
2   import random
3   import numpy as np
4   import pandas as pd
5   import tensorflow as tf
6   import matplotlib.pyplot as plt
7   import pandas_datareader as data_reader
8
```

Como sempre, todo processo se inicia com as importações das bibliotecas, módulos e ferramentas que estaremos utilizando ao longo de nosso código. Para isso, via comando import inicialmente importamos math, biblioteca padrão para operações matemáticas simples; random que nos permitirá gerar números aleatórios para nossos estados; numpy que por sua vez é muito utilizada para operações matemáticas sobre matrizes numéricas (arrays numpy); pandas que nos fornecerá ferramentas e funções com suporte a dados de planilhas; tensorflow que será a biblioteca onde criaremos toda arquitetura de nossa rede neural artificial; matplotlib que oferece ferramentas para plotagem de gráficos e por fim pandas_datareader, módulo independentemente da biblioteca pandas que nos permitirá realizar o carregamento de dados para a memória a partir da internet.

```
9   from tqdm import tqdm_notebook, tqdm
10  from pandas.util.testing import *
11  from collections import deque
12
```

Na sequência realizamos algumas importações pontuais, a fim de obter melhor performance, uma prática comum é quando possível, apenas carregar as ferramentas que de fato serão usadas de determinadas bibliotecas. Dessa forma, por meio dos comandos from import, da biblioteca tqdm, que por sua vez oferece meios de plotagem de linhas de progresso, tqdm_notebook e tqdm, para que possamos acompanhar de forma visual o progresso de cada tomada de decisão de nosso agente; De pandas.util.testing importamos todas funções, uma vez que uma das etapas a ser implementada é a de avaliação de performance de nossa rede neural artificial; Por fim, de collections importamos deque, função que permitirá a rápida manipulação da base de dados atualizando os últimos valores lidos e interpretados da mesma.

```
13  class AI_Trader():
14
15      def __init__(self, state_size, action_space=3, model_name="AITrader"):
16          self.state_size = state_size
17          self.action_space = action_space
18          self.memory = deque(maxlen = 2000)
19          self.model_name = model_name
20
```

Em seguida criamos nossa primeira classe, chamada AI_Trader( ), sem atributos mesmo. Dentro de sua estrutura criamos o método construtor __init__( ) onde passamos como parâmetros self, state_size que receberá um valor como definição do número de estados a serem processados; action_space parametrizado com 3, uma vez que teremos tomadas de decisão baseadas em 3 estados (último estado, estado atual e próximo estado); model_name apenas dando um nome a nosso Bot.

Também são criados os referidos objetos dentro do método construtor, alocando assim variáveis para receber dados/valores específicos a serem repassados por meio de outras funções.

Da mesma forma é criado o objeto memory que recebe como atributo deque por sua vez parametrizado com maxlen = 2000, aqui estamos criando uma variável (literalmente alocando um espaço na memória) que manterá sempre carregado uma base de dados com os últimos 2000 valores lidos a partir da fonte, essa é a memória de longa duração de nosso agente, é por meio dela que serão lidos e aprendidos os padrões de sua função de corretor.

Finalizando este bloco, é criado o objeto model_name que foi definido com o nome padrão AITrader.

```
21          self.gamma = 0.95
22          self.epsilon = 1.0
23          self.epsilon_final = 0.01
24          self.epsilon_decay = 0.995
25          self.model = self.model_builder()
26
```

Dando sequência, ainda indentado em nosso método construtor temos alguns parâmetros definidos manualmente (que de preferência devem receber outros valores e serem submetidos a novos testes). O primeiro deles, gamma se equivale com a taxa de aprendizado de outros modelos de rede neural artificial, porém tome cuidado para não confundir, não somente a

nomenclatura usual é diferente, mas a função de gama é parametrizar um ajuste de pesos em tempo real, diferente do learning rate que aprende “em gerações” de execução da rede neural.

Em seguida os objetos epsilon, epsilon_final e epsilon_decay terão o papel de manualmente definir parâmetros que combinados com a “taxa de aprendizado” simularão o processo de descida do gradiente, onde os ajustes dos pesos se dão de forma a rede neural buscar o melhor valor possível de taxa de acerto em suas tomadas de decisão, apenas não confundir também com a descida do gradiente de fato, que em um modelo de rede neural convencional possui seus valores atualizados a cada época de execução da rede neural, enquanto aqui, neste modelo em particular, esse processo é retroalimentado em ciclo ininterruptamente.

Por fim é criado o objeto model que por sua vez carrega a função model_builder( ), já que todos esses parâmetros iniciais são o gatilho inicial de execução da rede neural artificial intuitiva, porém a mesma a partir de certo momento se retroalimentará continuamente realizando processamento em tempo real.

```
27    def model_builder(self):
28        model = tf.keras.models.Sequential()
29        model.add(tf.keras.layers.Dense(units = 32,
30                                        activation = "relu",
31                                        input_dim = self.state_size))
32        model.add(tf.keras.layers.Dense(units = 64,
33                                        activation = "relu"))
34        model.add(tf.keras.layers.Dense(units = 128,
35                                        activation = "relu"))
36        model.add(tf.keras.layers.Dense(units = self.action_space,
37                                        activation = "linear"))
38        model.compile(loss = "mse",
39                      optimizer = tf.keras.optimizers.Adam(lr = 0.001))
40        return model
41
```

Prosseguindo com nosso código, hora de criar a arquitetura da rede neural artificial em si. Aqui existem dois pontos importantes a se observar, primeiro deles que a criação dessa estrutura se dará por meio da biblioteca *Keras*, incorporada ao *Tensorflow*, existem outras bibliotecas que oferecem até ferramentas mais simples para se criar a estrutura da rede neural, porém aqui, para fins de uma rede neural artificial intuitiva, conseguimos uma ótima performance via Keras. Outro ponto é que construiremos essa estrutura

situada dentro de uma classe já que iremos retroalimentar a mesma e a reprocessar constantemente como qualquer outra função

Sendo assim, inicialmente criamos a classe model_builder( ) apenas instanciando ela mesma, dentro de sua estrutura criamos um objeto de nome model que por sua vez recebe como atributo a função Sequential( ) que permitirá a criação de um modelo sequencial, em outras palavras, uma estrutura de rede neural multicamada onde as camadas estão interconectadas entre si, permitindo o processo de processamento de dados de forma subsequente. Desmembrando esse código, temos a função Sequential( ), parte do módulo models da biblioteca keras, parte integrante do TensorFlow que aqui estamos referenciando como tf.

Em seguida é criada a camada de entrada dessa rede neural por meio da função add( ), que por sua vez está parametrizada com a classe Dense( ), do módulo layers da biblioteca keras, parametrizada com units = 32, ou seja, 32 neurônios na camada de entrada; activation = “relu” (ReLU – Rectified Linear Unit), muito usada para problemas de classificação binária; input_dim = self.state_size, que de modo geral irá importar o tamanho da matriz conforme especificado pelo supervisor.

Da mesma forma são criadas mais duas camadas de neurônios, essas por sua vez, fazendo uso da função Dense( ) parametrizando a mesma com units = 64 e 128 respectivamente, ambas com activation = “relu”, ou seja, a primeira camada intermediária terá 64 neurônios em sua composição e a segunda 128, ambas usando da função de ativação ReLU, automaticamente realizando uma classificação de ativação ou não do neurônio da camada subsequente.

Nos mesmos moldes criamos uma terceira camada, que em modelos de redes neurais artificiais convencionais é a chamada camada de saída, uma vez que a mesma encerra o processamento da rede neural fornecendo dados de resultado. Aqui, units = self.action_space está pegando os valores de saída e transformando eles em um estado que retroalimentará a rede. *activation = “linear” apenas replica o resultado, não aplicando nenhuma função sobre ele.

Como de praxe criamos o compilador para essa rede, via função compile( ) parametrizado com loss = “mse”, definindo que a função de perda/custo será uma métrica chamada Mean Square Error, onde os valores obtidos são somados, elevados ao quadrado e comparado com os dados

estimados, dando assim mais peso aos erros para que a rede os identifique e aprenda a não replicá-los.

Finalizando, optimizer recebe Adam como função, parametrizado com lr = 0.001, o que significa que aplica-se sobre os resultados obtidos do processamento uma função de otimização que possui uma taxa de aprendizado de 0.001, este parâmetro define como será internamente o processo de atualização dos pesos das amostras, para um ajuste fino dos mesmos em busca de padrões.

Os valores encontrados pela rede são retornados e atribuídos a model.

```
42    def trade(self, state):
43      if random.random() <= self.epsilon:
44        return random.randrange(self.action_space)
45      actions = self.model.predict(state)
46      return np.argmax(actions[0])
47
```

Dando continuidade criamos uma das funções que simulam a arbitrariedade de nosso Agente. Logo, criamos a classe trade que por sua vez deve receber um estado. Dentro de sua estrutura é criada uma estrutura condicional onde se os números gerador pela função random( ) forem menores ou iguais aos valores de self.epsilon, é definido via randrange( ) um parâmetro de tamanho/intervalo para self.action_space, dessa forma criamos uma dependência de um estado para uma tomada de ação, continuamente.

Também é criada uma variável actions que por sua vez é atribuída com a função model_predict( ) que recebe um estado, ou seja, existe uma leitura constante do estado para que se possa ter o gatilho de desempenhar uma função. Finalizando, é retornado via função np.argmax( ) o valor máximo encontrado no índice 0 de actions. O que em outras palavras, nesse caso é simplesmente o último valor encontrado.

```
48    def batch_train(self, batch_size):
49      batch = []
50      for i in range(len(self.memory) - batch_size + 1, len(self.memory)):
51        batch.append(self.memory[i])
52      for state, action, reward, next_state, done in batch:
53        if not done:
54          reward = reward+self.gamma*np.amax(self.model.predict(next_state)[0])
55        target = self.model.predict(state)
56        target[0][action] = reward
57        self.model.fit(state, target, epochs=1, verbose=0)
58      if self.epsilon > self.epsilon_final:
59        self.epsilon *= self.epsilon_decay
60
```

Ainda criando o sistema de dependências de ações, que estipulam para nosso agente que ele sempre deve estar em busca da próxima ação a ser tomada, criamos um propósito a ser alcançado, nesse caso, uma recompensa.

Preste bastante atenção pois esta etapa pode parecer um tanto confusa, pois haverá o cruzamento de uma série de dados agora finalmente interligados.

Então, inicialmente criamos a classe batch_train( ) que recebe um batch_site, um tamanho que podemos entender como o tamanho da memória de curta duração de nosso Agente. Dentro de sua estrutura é criado um objeto de nome batch atribuído simplesmente com uma lista vazia e na sequência um laço de repetição onde é lido o tamanho de self.memory subtraído do valor de batch_size (valor ainda a ser repassado) somado de 1, também é lido novamente o valor de self.memory e a cada repetição do laço é adicionado o último valor encontrado em self.memory. Raciocine que esta primeira etapa está apenas construindo uma base de dados para a memória de nosso Agente.

Em seguida é criado um segundo laço de repetição, realizando a leitura dos dados de state, action, reward, next_state, done do objeto batch, para cada repetição é imposta a condição de que, se não houver um valor para done, uma variável chamada reward receberá como valor ela mesma somada do valor de gamma, multiplicada pelo valor máximo obtido via predict( ) parametrizado com next_state em seu índice 0.

Vamos entender o que está acontecendo aqui, repare que é criada a variável done como um objetivo final (internamente inclusive o valor padrão dessa variável é booleano) e o que queremos por hora é que justamente esse estado não seja facilmente alcançado, para que o processo não termine, mas funcione em ciclo. Para que esse loop não seja eterno, é criado um sistema de

recompensa, onde podemos especificar que quando um determinado valor for atingido em reward, aí sim o Agente pode parar.

Do mesmo modo é criado um objeto target que por sua vez recebe como atributo a função predict( ) parametrizada com o valor de state. Quando o valor no índice 0 de target e action tiverem o mesmo valor de reward, aí sim um estado está completo.

Para que isso funcione é necessário um gatilho inicial, onde por meio da função fit( ) alimentamos toda essa estrutura condicional com um estado, um alvo e uma época de processamento via suas referidas variáveis state, target e epochs. Encerrando esse bloco de código temos uma última estrutura condicional onde se o valor de epsilon for maior do que epsilon_final, o mesmo deve ser multiplicado por ele mesmo e por epsilon_decay, haja visto que esta estrutura de memória tem um limite de tamanho/extensão a não ser ultrapassado.

```
61  def sigmoid(x):
62    return 1 / (1 + math.exp(-x))
63
```

Dando sequência definimos manualmente nossa função sigmoid( ) parametrizado com uma variável temporária x. A função sigmoide como bem conhecemos é muito utilizada quando queremos realizar uma classificação binária, onde os dados estarão normalizados entre uma escala de 0 a 1.

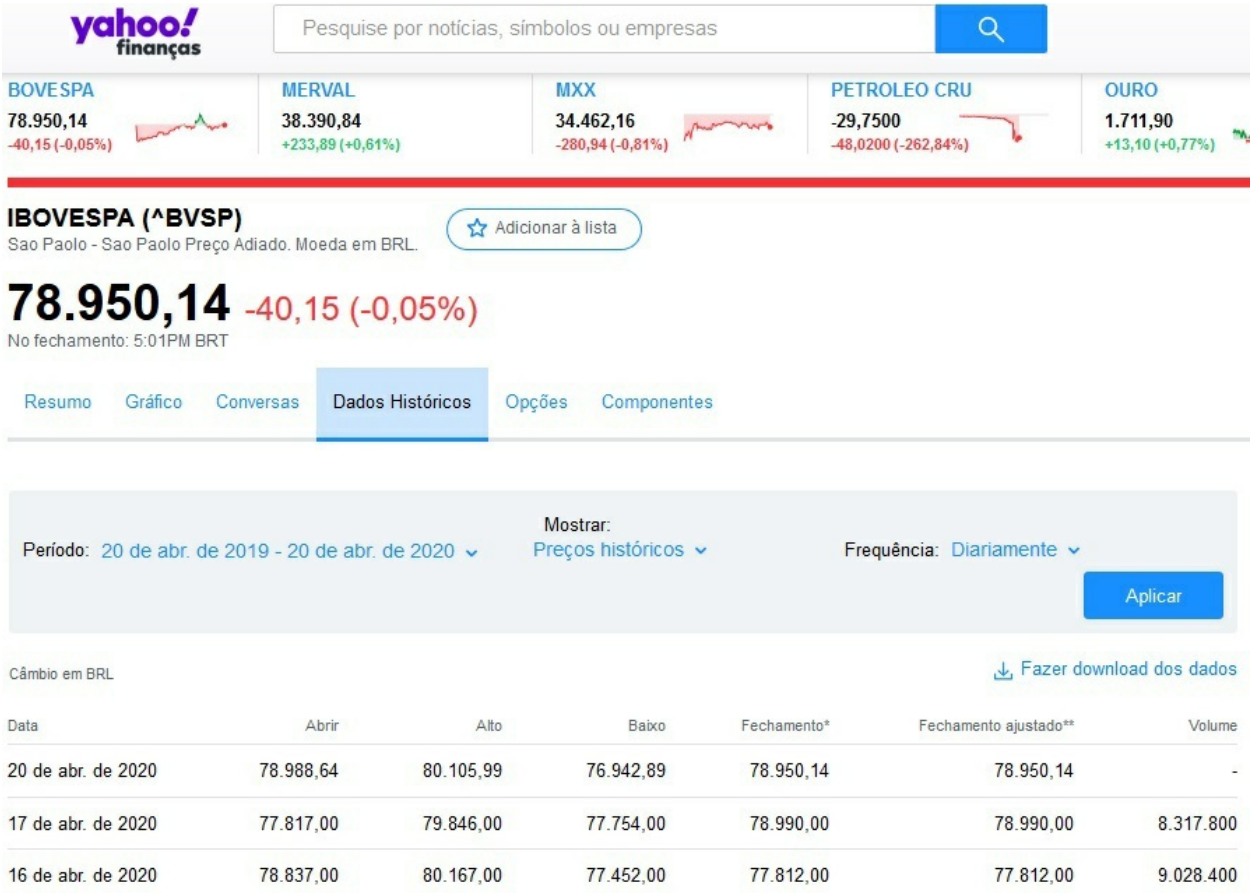
```
64  def stocks_price_format(n):
65    if n < 0:
66      return "- R$ {0:2f}".format(abs(n))
67    else:
68      return "R$ {0:2f}".format(abs(n))
69
```

Falando em normalização não podemos nos esquecer de criar uma pequena estrutura de código que transliterará os valores obtidos até então de uma forma visualmente melhorada. Para isso, criamos uma função stocks_price_format( ) que recebe uma variável temporária n, dentro de sua estrutura é criada uma estrutura condicional onde enquanto o valor de n for menor que 0, o valor será exibido como negativo, apresentando duas casas decimais, caso contrário, exibido como positivo.

```
70  dataset = stocks_price_format(100)
71  dataset = data_reader.DataReader("AAPL", data_source = "yahoo")
72
```

Criada a primeira metade de nossa estrutura de código, podemos começar a visualizar alguns dados. Seguindo com nosso código criamos uma variável de nome dataset que por sua vez recebe como atributo a função stocks_price_format( ) parametrizada em 100, em outras palavras estamos criando uma pequena base com os últimos 100 registros encontrados.

Diferente de outros modelos de redes neurais que temos uma base de dados fixa, aqui precisamos de algo que esteja sempre atualizado e em conformidade. Sendo assim, por meio da ferramenta DataReader( ) iremos realizar a importação desses dados a partir da internet. Uma vez que essa ferramenta já vem pré-configurada para esse propósito, basta passarmos dois parâmetros para que a mesma consiga realizar a leitura e importação dos dados. “AAPL” é a sigla oficial para ações da Apple em diferentes mercados, assim como para data_source passamos o parâmetro “yahoo” já que queremos que tais dados sejam importados do Yahoo Finanças.



Apenas por curiosidade, acessando o site Yahoo Finanças, consultando qualquer portfólio, em sua aba Dados Históricos é possível ver

os últimos lançamentos assim como baixar uma base de dados em planilha Excel com um intervalo de tempo predefinido.

```
73    print(dataset.head())
74    print(str(dataset.index[0]).split()[0])
75    print(dataset.index[-1])
76    print(dataset['Close'])
77
```

Assim como toda e qualquer base de dados gerada a partir de uma planilha Excel, podemos consultar os dados usando as funções básicas as quais temos familiaridade. Apenas como exemplo, dataset.head( ) mostrará os 5 primeiros registros, index[-1] mostrará o último registro, [‘Close’] exibirá apenas os valores de fechamento das ações, etc...

```
78  def dataset_loader(stock_name):
79      dataset = data_reader.DataReader(stock_name, data_source = "yahoo")
80      start_date = str(dataset.index[0]).split()[0]
81      end_date = str(dataset.index[-1]).split()[0]
82      close = dataset['Close']
83      return close
84
```

Em seguida, criando de fato a função que será acionada para a importação e atualização dos dados da base, criamos a função dataset_loader( ) parametrizada com os dados da variável stock_name. Dentro de sua estrutura é instanciada a variável dataset que como feito antes, por meio de DataReade( ) recebe os dados de stock_name e da fonte “yahoo”.

É definida manualmente também, atribuindo para start_date e end_date, respectivamente, os valores de início e fim do período especificado para esse dataset, assim como é criada a variável close que terá atribuídos para si apenas os valores de fechamento das ações. Note que toda essa estrutura justamente retorna os valores para close, é uma medida que tomamos porque o que nos interessará (na memória de curta duração de nosso Agente) será a atualização deste dado, na sua última atualização possível via site.

```
def state_creator(data, timestep, window_size):
  starting_id = timestep - window_size + 1

  if starting_id >= 0:
    windowed_data = data[starting_id:timestep + 1]
  else:
    windowed_data = - starting_id * [data[0]] + list(data[0:timestep + 1])

  state = []
  for i in range(window_size - 1):
    state.append(sigmoid(windowed_data[i + 1] - windowed_data[i]))

  return np.array([state]), windowed_data

```

Uma vez criada a função responsável pela leitura dos dados de entrada podemos avançar mais um pouco criando agora a função criadora de estados. Logo, criamos a função state_creator( ) que receberá posteriormente dados para data, timestep e window_size.

Inicialmente é criada, indentada dentro do corpo da função, a variável starting_id que recebe como atributos timestep (um dado temporal, correspondente a uma data) que subtrai desse valor o tamanho de window_size (intervalo de tempo) acrescentando 1 ao final da operação, dessa forma a data inicial a ser lida na base de dados é atualizada.

Em seguida é criada uma estrutura condicional onde se os valores de starting_id forem iguais ou maiores que 0, é atribuído para windowed_data o valor de data em sua última posição indexada. Caso contrário, é atribuído o valor de starting_id multiplicado pelos dados do índice 0 de data somado ao valor da última posição indexada. Repare que assim reforçamos aquela estrutura que gera uma dependência, sempre haverá a necessidade de que estes dados sejam atualizados, isso pode ser trabalhado em ciclo, reforçando a operacionalidade de nosso agente processando dados em tempo real simulado.

Em seguida é criada a variável state que inicialmente atribuída para si tem apenas uma lista vazia. Também criamos um laço de repetição onde é lido o último registro de window_size e por meio da função append( ) adicionamos nesse campo o resultado da função sigmoid( ) entre o penúltimo e o último dado registrado na base de dados. Apenas relembrando que a função sigmoide por sua vez retornará um valor 0 ou 1, que nesse contexto pode representar a tomada ou não de uma ação com base nos valores lidos.

```
stock_name = "AAPL"
data = dataset_loader(stock_name)
s = state_creator(data, 0, 5)
window_size = 10
episodes = 1000
batch_size = 32
data_samples = len(data) - 1

```

Seguindo com o código, agora que temos uma estrutura já funcional, podemos definir alguns parâmetros a serem usados. Lembrando que definimos esses parâmetros manualmente e por meio de variáveis para que possamos os modificar de forma mais fácil e dinâmica.

Seguindo essa lógica criamos a variável stock_name que recebe “AAPL” como atributo, uma vez que neste exemplo em particular estamos trabalhando com ações da Apple; data que recebe como atribuição o retorno da função dataset_loader( ) que por sua vez recebe o dado de stock_name (Caso você queira testar outro portfólio de carteiras de ações de outras empresas basta alterar o nome repassado a stock_name; s que instancia a função state_creator( ) passando um valor para data, 0 e 5 respectivamente. Apenas relembrando que s é o estado de nosso agente de acordo com a lógica da equação de Bellman.

```
trader = AI_Trader(window_size)

print(trader.model.summary())

```

Dando sequência criamos a variável trader que por sua vez instancia a classe AI_Trader( ) passando como atributo de classe os dados contidos em window_size. A partir desse ponto é possível também visualizar (caso não haja nenhum erro de sintaxe ou de cruzamento dos dados) o sumário de nossa rede neural artificial intuitiva, por meio da função print( ) parametrizada com trader.model.summary( ).

```
If using Keras pass *_constraint arguments to layers.
  0%|          | 0/1257 [00:00<?, ?it/s]Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 32)                352
_________________________________________________________________
dense_1 (Dense)              (None, 64)                2112
_________________________________________________________________
dense_2 (Dense)              (None, 128)               8320
_________________________________________________________________
dense_3 (Dense)              (None, 3)                 387
=================================================================
Total params: 11,171
Trainable params: 11,171
Non-trainable params: 0
_________________________________________________________________
None
```

Via console é possível visualizar o sumário que descreve a arquitetura de nossa rede neural artificial intuitiva. Como visto anteriormente trata-se de uma estrutura bastante básica, com poucos neurônios e camadas de processamento, ainda assim, note que, de acordo com os dados acima, temos a interconexão de 11,171 neurônios, cada um por sua vez com seus respectivos valores, pesos, funções de ativação, etc...

```
111  for episode in range(1, episodes + 1):
112      print(f'Etapa: {episode} de {episodes}')
113      state = state_creator(data, 0, window_size + 1)
114      total_profit = 0
115      trader.inventory = []
116      for t in tqdm(range(data_samples)):
117          action = trader.trade(state)
118          next_state = state_creator(data, t + 1, window_size + 1)
119          reward = 0
120
```

Concluída a estrutura de nosso agente em si, uma prática bastante comum é criar uma estrutura onde ao mesmo tempo em que os dados são lidos e processados, possamos acompanhar de forma visual o processamento dos mesmos.

Para tal feito, inicialmente criamos um laço de repetição que percorrerá cada episódio (cada ciclo de processamento) de nosso agente, exibindo em terminal o andamento dessas etapas. Novamente é instanciada a variável state chamando a função state_creator( ). É criada a variável total_profit

inicialmente com valor zerado, que será atualizada com os dados referentes ao lucro ou perda obtidos no processo de corretagem. Da mesma forma também é criada sobre trader um objeto de inventário que inicialmente recebe uma lista vazia atribuída a si, mas que armazenará os dados das transações.

Em seguida é criado um novo laço de repetição dentro do laço atual, onde fazendo o uso da ferramenta tqdm( ) parametrizada com o tamanho de data_samples nos exibirá uma barra de progresso em nosso console, facilitando a visualização do progresso de processamento em si.

Também é criada a variável reward, inicialmente com valor 0 atribuído, uma vez que no gatilho inicial de nosso agente ainda não há o feedback de uma ação e automaticamente ainda não há uma recompensa ou penalidade aplicada sobre seu estado atual.

```
121     if action == 1:
122       trader.inventory.append(data[t])
123       print("AI Trader comprou: ", stocks_price_format(data[t]))
124     elif action == 2 and len(trader.inventory) > 0:
125       buy_price = trader.inventory.pop(0)
126 
127       reward = max(data[t] - buy_price, 0)
128       total_profit += data[t] - buy_price
129       print("AI Trader vendeu: ", stocks_price_format(data[t]),
130             "Lucro de: " + stocks_price_format(data[t] - buy_price))
131 
```

Ainda dentro do laço de repetição inicial agora criamos uma estrutura condicional a ser executada. Basicamente, estipulamos que se o valor de action for igual a 1 será realizada a ação de compra de uma ação. Para isso simplesmente adicionamos ao inventário o valor lido em data em seu último estado. Também exibimos a mensagem que nosso Agente comprou uma determinada ação. Ainda nessa estrutura, caso o valor de action seja igual a 2 e o tamanho do inventário seja maior que 0, de buy_price é removido o valor 0 de seu inventário. Se você reparar, esta é apenas outra maneira de representar a necessidade de nosso agente de buscar o últimos valor obtido para que se realize todo um processamento sobre o mesmo, em continuidade.

Em seguida reward agora é atualizada com o valor máximo do estado atual de data subtraindo o valor de buy_price. Da mesma forma é possível também definir que total_profit quando atualizado recebe o valor dele mesmo somado do valor atual de data menos o valor de buy_price, em outras palavras, devemos considerar todo o processo para separar apenas o que diz

respeito ao lucro total da operação.

Finalizando, é exibida em tela uma primeira mensagem de que nesse caso nosso Agente vendeu uma ação e uma segunda mensagem exibindo qual foi o lucro obtido na operação, já que, corretagem trata-se de comprar uma determinada ação em um preço e vender em outro tentando sempre lucrar com essa diferença de valor de compra/venda.

```
132     if t == data_samples - 1:
133       done = True
134     else:
135       done = False
136
```

Finalizando esse bloco, ainda dentro do laço de repetição inicial, agora simplesmente criamos uma estrutura condicional que define que se o valor de t for igual ao valor de data_samples subtraído 1, done recebe o estado True, caso contrário, False e o ciclo se repete.

```
137     trader.memory.append((state, action, reward, next_state, done))
138     state = next_state
139
```

Apenas lembrando que “o ciclo se repete” significa uma atualização do estado atual e uma nova tomada de ação por parte de nosso Agente. Sendo assim caso a estrutura condicional anterior retorne False, é adicionado a memória de nosso agente novamente os últimos valores obtidos e atribuídos para state, action, reward e next_state, além da leitura do estado de done nessa própria condicional. Por fim é atualizado o valor de state com o valor de next_state ciclicamente.

```
140     if done:
141       print("########################")
142       print(f'Lucro Total Estimado: {total_profit}')
143       print("########################")
144
145     if len(trader.memory) > batch_size:
146       trader.batch_train(batch_size)
147
148   if episode % 10 == 0:
149     trader.model.save("ai_trader_{}.h5".format(episode))
150
```

Finalizando nosso código, é criada uma estrutura condicional onde simplesmente quando done for validado, é impresso em tela a mensagem do lucro total estimado, exibindo o valor contido em total_profit.

Também é criada uma estrutura condicional para atualização de batch_train, haja visto que nosso agente simultaneamente ao seu processamento intuitivo está realizando aprendizado de máquina e a cada ciclo de processamento os padrões encontrados são guardados como parâmetros de aprendizado. Uma última estrutura condicional é criada onde sempre que o valor do resto da divisão entre episode e 10 for 0, os parâmetros dos pesos da rede neural são salvos em um arquivo tipo .h5 reutilizável.

Essa é uma prática comum porque uma vez terminado todo um ciclo de processamento, podemos guardar esses valores e exportar nosso agente para outra máquina, por exemplo, onde lidos esses valores o mesmo não precisará realizar todo seu aprendizado novamente do zero.

```
[5 rows x 6 columns]
2015-04-23
2020-04-21 00:00:00
Date
2015-04-23    129.66
2015-04-24    130.27
2015-04-27    132.64
2015-04-28    130.55
2015-04-29    128.63
                ...
2020-04-15    284.42
2020-04-16    286.69
2020-04-17    282.79
2020-04-20    276.92
2020-04-21    268.63
Name: Close, Length: 1258, dtype: float64
```

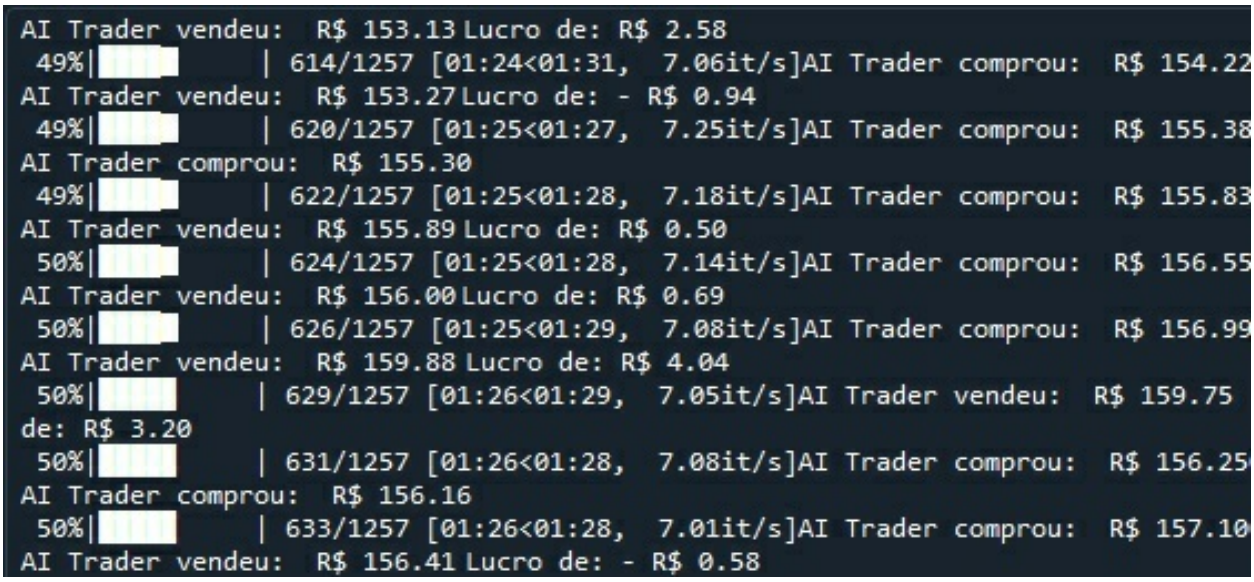
Sendo assim, ao rodar o código podemos acompanhar via console alguns dados, o primeiro deles, os dados do intervalo de tempo lidos a partir do Yahoo Finanças.

```
_________________________________________________________________
None
Etapa: 1 de 1000
AI Trader comprou:  R$ 127.62
AI Trader comprou:  R$ 126.32
AI Trader comprou:  R$ 126.01
AI Trader vendeu:  R$ 128.94 Lucro de: R$ 1.32
AI Trader vendeu:  R$ 128.77 Lucro de: R$ 2.45
AI Trader comprou:  R$ 130.19
AI Trader comprou:  R$ 130.07
AI Trader comprou:  R$ 130.05
AI Trader comprou:  R$ 131.38
AI Trader comprou:  R$ 132.03
AI Trader comprou:  R$ 131.77
AI Trader vendeu:  R$ 130.27 Lucro de: R$ 4.26
AI Trader vendeu:  R$ 130.53 Lucro de: R$ 0.34
AI Trader comprou:  R$ 129.96
AI Trader vendeu:  R$ 129.36 Lucro de: - R$ 0.71
```

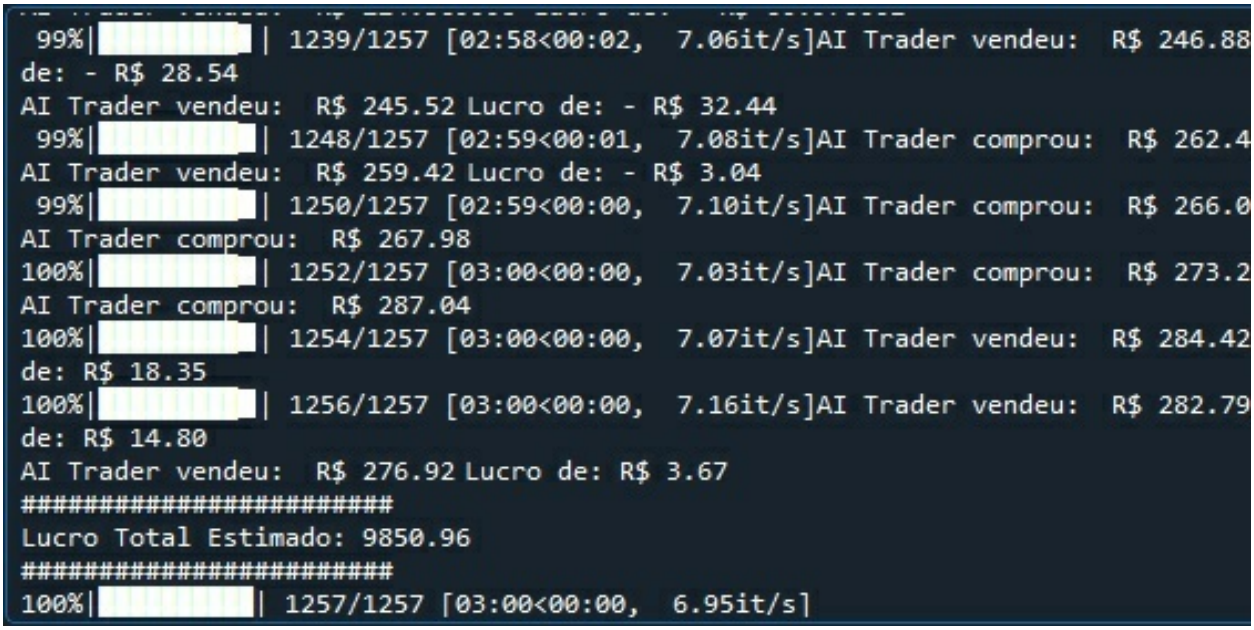
A visualização dos processamentos ciclo a ciclo, inclusive com as primeiras tomadas de decisão por parte de nosso Agente.

```
  3%|▌          | 33/1257 [00:00<00:19, 61.75it/s]AI Trader vendeu:  R$ 128.88
de: - R$ 2.89
AI Trader vendeu:  R$ 128.58 Lucro de: - R$ 1.37
  3%|▌          | 39/1257 [00:01<01:53, 10.69it/s]AI Trader comprou:  R$ 127.87
AI Trader vendeu:  R$ 126.59 Lucro de: - R$ 1.27
  4%|▋          | 48/1257 [00:02<02:52,  7.03it/s]AI Trader comprou:  R$ 125.43
AI Trader vendeu:  R$ 126.59 Lucro de: R$ 1.16
  4%|▋          | 52/1257 [00:03<02:48,  7.15it/s]AI Trader comprou:  R$ 125.69
AI Trader comprou:  R$ 122.57
  4%|▋          | 54/1257 [00:03<02:53,  6.94it/s]AI Trader vendeu:  R$ 120.07
de: - R$ 5.62
AI Trader comprou:  R$ 123.27
  4%|▋          | 55/1257 [00:03<01:21, 14.72it/s]AI Trader vendeu:  R$ 125.66
de: R$ 3.09
```

Progresso das primeiras corretagens, mantendo inconsistência entre valores de compra e venda, realizando os primeiros passos de aprendizado de máquina.



Resultados melhorando a medida que o Agente evolui.



Resultado final, nesse cenário simulado onde nosso Agente a partir das cotações das ações da Apple obtidas via Yahoo Finanças, aprendeu os padrões certos para obtenção de lucro a partir da corretagem das mesmas.

Nesse contexto, realizando operações baseadas nos últimos registros, retroalimentando a rede e tomando novas decisões, nosso Agente inicialmente possuía uma margem de acertos sobre suas ações praticamente nula, porém, com algumas horas de processamento é possível notar que o

objetivo principal de obter lucro com base na diferença entre os valores pagos por uma ação e os valores de venda da mesma não só mantiveram uma taxa de acertos alta como um lucro bastante significativo.

Código Completo:

```
import math
import random
import numpy as np
import pandas as pd
import tensorflow as tf
import matplotlib.pyplot as plt
import pandas_datareader as data_reader

from tqdm import tqdm_notebook, tqdm
from pandas.util.testing import *
from collections import deque

class AI_Trader():

  def __init__(self, state_size, action_space=3, model_name="AITra
    self.state_size = state_size
    self.action_space = action_space
```

```
    self.memory = deque(maxlen = 2000)
    self.model_name = model_name

    self.gamma = 0.95
    self.epsilon = 1.0
    self.epsilon_final = 0.01
    self.epsilon_decay = 0.995
    self.model = self.model_builder()

def model_builder(self):
    model = tf.keras.models.Sequential()
    model.add(tf.keras.layers.Dense(units = 32,
                                    activation = "relu",
                                    input_dim = self.state_size))
    model.add(tf.keras.layers.Dense(units = 64,
                                    activation = "relu"))
    model.add(tf.keras.layers.Dense(units = 128,
                                    activation = "relu"))
    model.add(tf.keras.layers.Dense(units = self.action_space,
                                    activation = "linear"))
    model.compile(loss = "mse",
                  optimizer = tf.keras.optimizers.Adam(lr = 0.001))
    return model
```

```
 def trade(self, state):
   if random.random() <= self.epsilon:
     return random.randrange(self.action_space)
   actions = self.model.predict(state)
   return np.argmax(actions[0])

 def batch_train(self, batch_size):
   batch = []
   for i in range(len(self.memory) - batch_size + 1, len(self.memory
     batch.append(self.memory[i])
   for state, action, reward, next_state, done in batch:
     if not done:
       reward = reward+self.gamma*np.amax(self.model.predict(nex
[0])
     target = self.model.predict(state)
     target[0][action] = reward
     self.model.fit(state, target, epochs=1, verbose=0)
   if self.epsilon > self.epsilon_final:
     self.epsilon *= self.epsilon_decay

def sigmoid(x):
 return 1 / (1 + math.exp(-x))
```

```
def stocks_price_format(n):
  if n < 0:
    return "- R$ {0:2f}".format(abs(n))
  else:
    return "R$ {0:2f}".format(abs(n))

dataset = stocks_price_format(100)
dataset = data_reader.DataReader("AAPL", data_source = "yahoo")

print(dataset.head())
print(str(dataset.index[0]).split()[0])
print(dataset.index[-1])
print(dataset['Close'])

def dataset_loader(stock_name):
  dataset = data_reader.DataReader(stock_name, data_source = "yah
  start_date = str(dataset.index[0]).split()[0]
  end_date = str(dataset.index[-1]).split()[0]
  close = dataset['Close']
  return close

def state_creator(data, timestep, window_size):
```

```
  starting_id = timestep - window_size + 1

  if starting_id >= 0:
    windowed_data = data[starting_id:timestep + 1]
  else:
    windowed_data = - starting_id * [data[0]] + list(data[0:timestep +

  state = []
  for i in range(window_size - 1):
    state.append(sigmoid(windowed_data[i + 1] - windowed_data[i])

  return np.array([state]), windowed_data

stock_name = "AAPL"
data = dataset_loader(stock_name)
s = state_creator(data, 0, 5)
window_size = 10
episodes = 1000
batch_size = 32
data_samples = len(data) - 1

trader = AI_Trader(window_size)
```

```
print(trader.model.summary())

for episode in range(1, episodes + 1):
  print(f'Etapa: {episode} de {episodes}')
  state = state_creator(data, 0, window_size + 1)
  total_profit = 0
  trader.inventory = []
  for t in tqdm(range(data_samples)):
    action = trader.trade(state)
    next_state = state_creator(data, t + 1, window_size + 1)
    reward = 0

    if action == 1:
      trader.inventory.append(data[t])
      print("AI Trader comprou: ", stocks_price_format(data[t]))
    elif action == 2 and len(trader.inventory) > 0:
      buy_price = trader.inventory.pop(0)

      reward = max(data[t] - buy_price, 0)
      total_profit += data[t] - buy_price
      print("AI Trader vendeu: ", stocks_price_format(data[t]),
            "Lucro de: " + stocks_price_format(data[t] - buy_price))
```

```
    if t == data_samples - 1:
        done = True
    else:
        done = False

    trader.memory.append((state, action, reward, next_state, done))
    state = next_state

    if done:
        print("##########################")
        print(f'Lucro Total Estimado: {total_profit}')
        print("##########################")

    if len(trader.memory) > batch_size:
        trader.batch_train(batch_size)

if episode % 10 == 0:
    trader.model.save("ai_trader_{}.h5".format(episode))
```

Cão Robô que aprende sobre seu corpo e sobre o ambiente via ARS

Para realizarmos a implementação de uma inteligência artificial baseada em Augumented Random Search, usaremos de um exemplo bastante simples porém capaz de nos elucidar os principais pontos em destaque deste tipo específico de rede neural artificial intuitiva que é o de um cão (ou qualquer outro animal semelhante) onde o mesmo aprenderá sobre o seu corpo (como controlar o mesmo) e sobre o ambiente a ser explorado, de forma que o aprendizado realizado para seu corpo não interfere no aprendizado realizado para sua mente.

Raciocine que, como mencionado no capítulo teórico, nesse contexto o processo de aprendizado para o corpo é contínuo e ciclo após ciclo de processamento o mesmo é cada vez mais refinado, uma vez que não se "desaprende" a caminhar. Porém, nosso Agente terá toda uma estrutura lógica onde sim, será aplicado o conceito teórico de aprendizado por reforço para uma rede neural artificial intuitiva, onde o mesmo por tentativa e erro, sistema de recompensas e penalidades baseadas em estados e ações aprenderá a realizar uma determinada tarefa.

Nesse simples exemplo, nosso Agente é um cão robô que inicialmente não sabe de absolutamente nada sobre sua estrutura, ele identificará cada estrutura de sua anatomia simulada, ciclos após ciclos de processamento o mesmo será capaz de ficar em pé e caminhar, a partir disso ele será submetido a um percurso em um ambiente simulado inclusive cheio de obstáculos, para que aprenda a caminhar perfeitamente superando obstáculos.

De forma resumida, o que faremos é:

- Importação das bibliotecas e módulos utilizados ao longo do código.
- Carregamento do ambiente de exemplo e definição de seus

hyperparâmetros.

- Criação das políticas de tomadas de decisão que separarão os processos para o seu corpo e sua mente.

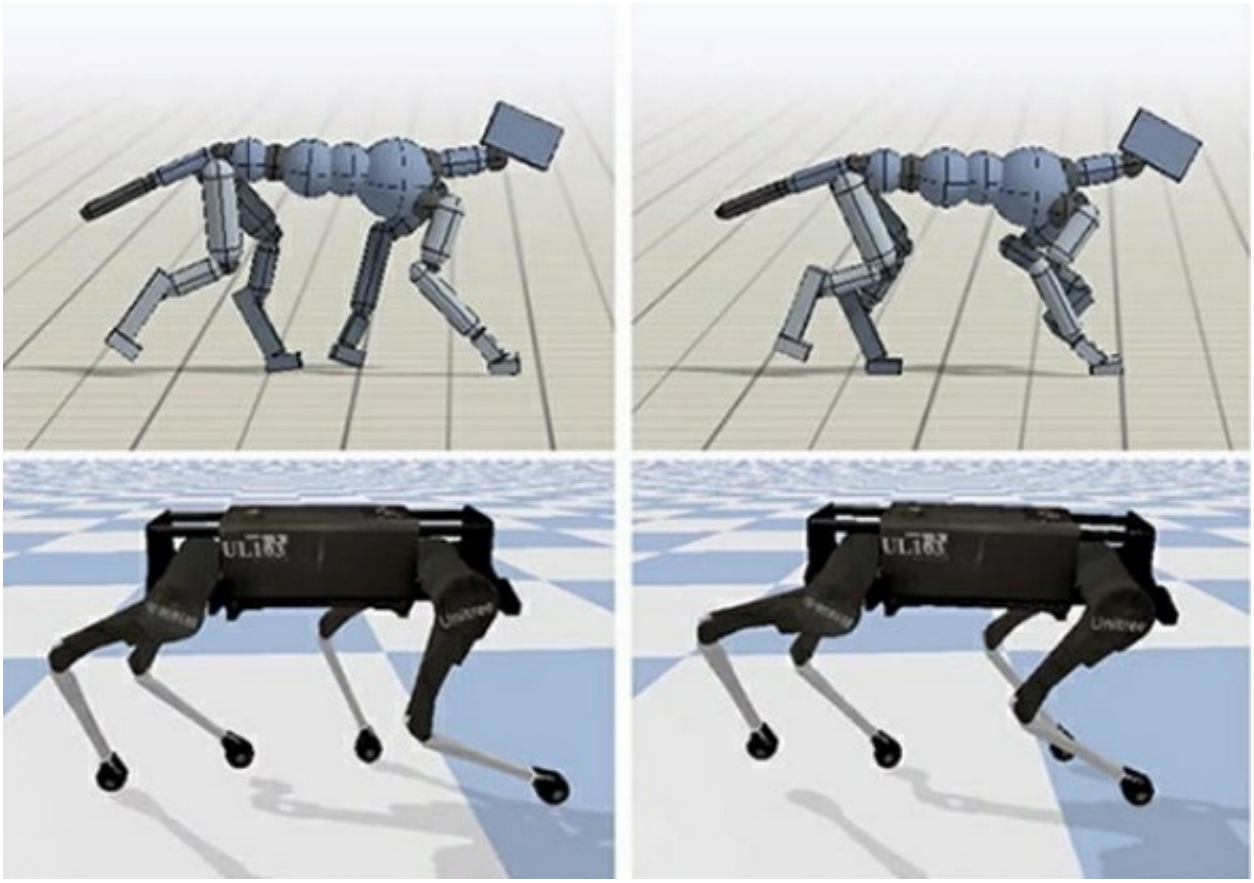
- Construção da arquitetura da rede neural artificial intuitiva e seu sistema de dependências.

- Criação dos estados iniciais de nosso Agente.

- Treinamento da IA para o seu devido propósito.

- Execução de nossa rede neural artificial intuitiva.

- Interpretação dos dados obtidos em processamento e em pós processamento.

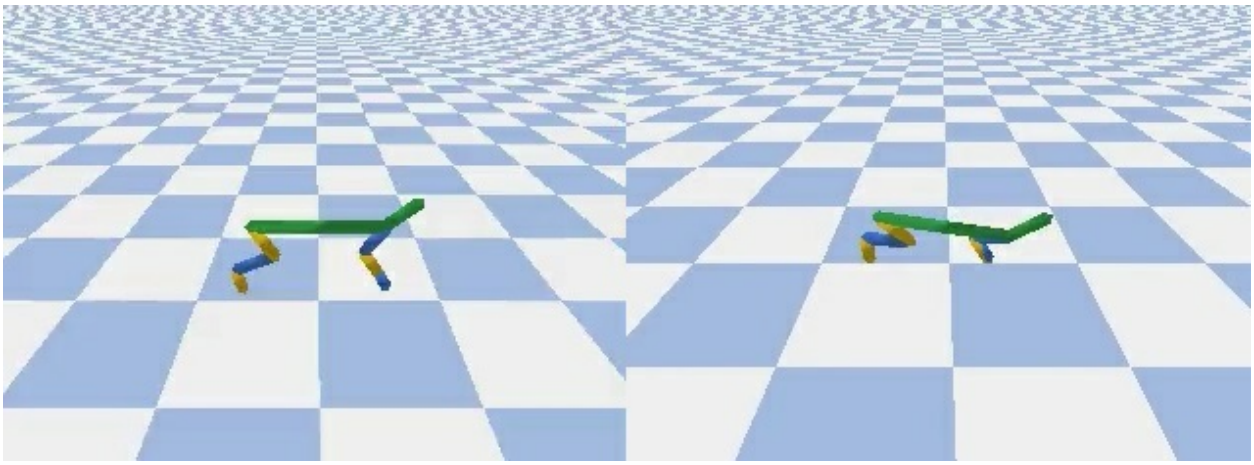


Neste exemplo estaremos implementando a inteligência artificial de um cão robô. Muito provavelmente você já deve ter visto, ao menos em vídeo,

algum destes tipos de robô em particular, onde o mesmo quando ativado pela primeira vez fica realizando movimentos totalmente aleatórios com seus membros, depois progressivamente vai realizando a interação dos membros com o corpo, ficando em pé, aprendendo a caminhar em alguma direção e por fim até mesmo superar obstáculos pelo caminho.

Estes tipos de robô, inicialmente idealizados para fins militares (para realizar atividades como carregar grandes volumes de equipamentos ou explorar ambientes sem pôr em risco um humano), hoje pode ser facilmente adaptado para os mais diversos propósitos.

O que não temos o costume de raciocinar sobre esses tipos de robô é que não há uma programação fixa scriptada para cada ação do mesmo, mas sim uma inteligência artificial intuitiva onde o mesmo aprende sobre seu corpo, seu ambiente e sobre a tarefa a ser realizada.



Estaremos na sequência implementando passo-a-passo a inteligência artificial deste tipo de robô em particular. Assim como ao final do processo teremos um modelo funcional pronto para ser colocado em execução.

Instalação das Dependências

Para nosso exemplo, única biblioteca adicional que deverá ser instalada é a PyBullet, por meio dela teremos o ambiente virtual ao qual nosso Agente estará inserido. O processo de instalação é feito como de costume, via Pip, através do comando **pip install pybullet**.

Partindo para o Código:

```
1   import os
2   import numpy as np
3   import gym
4   from gym import wrappers
5   import pybullet_envs
6
```

Como sempre, todo processo se inicia com as devidas importações das bibliotecas, módulos e ferramentas que estaremos utilizando ao longo de nosso código. Única diferença das demais importações realizadas nos outros exemplos é que agora estamos importando os ambientes de simulação da biblioteca PyBullet por meio de seu módulo pybullet_envs.

```
7   class Hp():
8       def __init__(self):
9           self.nb_steps = 1000
10          self.episode_length = 1000
11          self.learning_rate = 0.02
12          self.nb_directions = 16
13          self.nb_best_directions = 16
14          assert self.nb_best_directions <= self.nb_directions
15          self.noise = 0.03
16          self.seed = 1
17          self.env_name = 'HalfCheetahBulletEnv-v0'
18
```

Em seguida damos início a estrutura de código em si, primeiramente, criando o que para alguns autores é chamado de hyperparâmetros, o que em outras palavras simplesmente nos remete aos parâmetros que poderão ser livremente configurados para testes de eficiência de nosso modelo.

Sendo assim, inicialmente é criada a classe Hp( ), dentro de si há um simples método construtor __init__ e alguns objetos que definem alguns parâmetros. Como a essa altura você já deve estar familiarizado com a nomenclatura usual, vamos apenas revisar alguns pontos. Primeiro deles nb_steps define quantos ciclos estarão sendo realizados, o mesmo que

quantas vezes todo o código era executado com seus reajustes dos pesos em modelos de redes neurais artificiais convencionais. Em seguida episode_length define um intervalo de execuções dos ciclos (lembrando que aqui, apenas para exemplo, nosso Agente irá encerrar sua atividade ao alcançar o seu objetivo, em situações reais ele permanece constantemente ligado e funcional, apenas processando novas instruções de novas tarefas a serem realizadas, mas seu processamento é contínuo).

Da mesma forma como em outros exemplos, learning_rate especifica a taxa de aprendizado, de quantas em quantas amostras serão realizados os reajustes dos pesos; nb_directions e nb_best_directions aqui definem manualmente o número de ações a serem consideradas por ciclo, tenha em mente que agora múltiplas ações são realizadas em conjunto apenas por parte do corpo de nosso Agente se movendo, fora as várias possíveis ações a serem tomadas perante o ambiente.

Na sequência criamos uma estrutura condicional especial via assert, que por sua vez realiza uma ou mais verificações considerando o tempo de execução, para se certificar de o número atribuído para nb_best_directions deve ser igual ou menos que o valor de nb_directions. Isso é feito porque é possível que todas as ações que foram tomadas estejam corretas em seu contexto, mas uma exceção será criada caso esse valor exceda o número padrão máximo de possíveis ações a serem tomadas por nosso Agente.

Dando sequência é criado o objeto noise com valor atribuído de 0.03, este objeto define literalmente um ruído, um número específico de amostras aleatórias que serão enxertadas em meio aos estados e ações para eliminar vícios de processamento de nossa rede neural. Se a mesma estiver funcionando com o esperado irá imediatamente identificar esses ruídos e os eliminar de seu processamento.

Em seguida seed com valor 1 simplesmente é o gatilho inicial para a execução dos devidos processamentos.

Por fim, para env_name passamos como atributo ‘HalfCheetahBulletEnv-v0’, o que fará que seja carregado nosso modelo de Agente assim como seu ambiente de exemplo.

```
19  class Normalizer():
20      def __init__(self, nb_inputs):
21          self.n = np.zeros(nb_inputs)
22          self.mean = np.zeros(nb_inputs)
23          self.mean_diff = np.zeros(nb_inputs)
24          self.var = np.zeros(nb_inputs)
25
```

Na sequência criamos uma classe que ficará responsável por uma série de normalizações para os estados de nosso Agente. Inicialmente criamos a classe Normalizer( ), dentro da mesma é criado o método construtor __init__ que receberá um parâmetro para nb_inputs. Seguindo, são criados os objetos n que por sua vez recebe como atributo uma matriz de zeros a ser atualizada com os valores de nb_inputs; Da mesma forma mean, mean_diff e var são criadas da mesma forma.

```
26      def observe(self, x):
27          self.n += 1.
28          last_mean = self.mean.copy()
29          self.mean += (x - self.mean) / self.n
30          self.mean_diff += (x - last_mean) * (x - self.mean)
31          self.var = (self.mean_diff / self.n).clip(min = 1e-2)
32
```

Ainda dentro da classe Normalizer( ) criamos a função observe( ) que recebe um objeto x. Dentro de sua estrutura de código é inicialmente definido que o valor da soma de n com ele mesmo deve ser igual a 1. Em seguida é criado o objeto last_mean que receberá o último valor atribuído para mean por meio da função copy( ); mean por sua vez, agora justificando seu nome que significa média, recebe como atributo o valor da divisão da soma de seu valor por ele mesmo subtraído de x, dividido pelo valor de n. Note que aqui o que é feito é a simples média dos valores de mean para atualizar o valor de last_mean.

Na sequência é feito algo parecido para mean_diff, mas agora, para obtermos o valor da diferença entre essas médias calculadas, mean_diff

recebe como valor a soma de seu valor pela multiplicação de last_mean e mean subtraídos de x. Uma vez que temos os valores das médias e da diferença entre elas, nosso objeto var recebe como atributo o resultado da divisão entre mean_diff e n, dentro de um intervalo mínimo pré-estabelecido entre 1 e -2 via função clip( ).

```
33      def normalize(self, inputs):
34          obs_mean = self.mean
35          obs_std = np.sqrt(self.var)
36          return (inputs - obs_mean) / obs_std
37
```

Ainda indentado para Normalize( ) é criada a função normalize( ) que recebe inputs como parâmetro. Dentro de si o objeto obs_mean recebe como atributo o valor de mean, da mesma forma obs_std recebe o valor da raiz quadrada de var, finalizando retornando o valor da divisão entre a subtração dos valores de inputs e obs_mean para obs_std.

```
38  class Policy():
39      def __init__(self, input_size, output_size):
40          self.theta = np.zeros((output_size, input_size))
41
```

Logo após é iniciada a criação da estrutura voltada para a inteligência artificial em si, para isso inicialmente é criada a classe Policy( ), dentro de si é criado um método construtor __init__ que recebe input_size e output_size, finalizando, é criado o objeto theta que por sua vez recebe como atributo inicialmente uma matriz de zeros a ser preenchida com os valores de output_size e de input_size, respectivamente.

```
42      def evaluate(self, input, delta = None, direction = None):
43          if direction is None:
44              return self.theta.dot(input)
45          elif direction == "positive":
46              return (self.theta + hp.noise*delta).dot(input)
47          else:
48              return (self.theta - hp.noise*delta).dot(input)
49
```

Seguindo com o código, indentado para Policy( ) criamos a função evaluate( ) que recebe um dado/valor para input e já define manualmente que delta e direction são inicializados em None, ou seja, sem nenhum dado atribuído, apenas reservando essas variáveis para um propósito futuro.

Dentro de evaluate( ) é criada uma estrutura condicional onde se o estado de direction for None, é retornado o valor do produto escalar de theta por meio da função dot( ) parametrizada com os valores de input. Caso essa primeira condição não for verdadeira, é verificado então se o dado atribuído para direction é “positive”, caso sim, é retornado o produto escalar da soma entre os valores de theta e noise multiplicado por delta, equiparado a input. Por fim, caso essa condição não seja verdadeira, é retornado a subtração entre theta e a multiplicação entre noise e delta, equiparado a input.

```
50      def sample_deltas(self):
51          return [np.random.randn(*self.theta.shape) for _ in range(hp.nb_directions)]
52
```

Ainda em Policy( ), é criada a função sample_deltas, que retorna uma lista preenchida com os valores lidos para nb_directions convertidos para o formato de theta.

```
53      def update(self, rollouts, sigma_r):
54          step = np.zeros(self.theta.shape)
55          for r_pos, r_neg, d in rollouts:
56              step += (r_pos - r_neg) * d
57          self.theta += hp.learning_rate / (hp.nb_best_directions * sigma_r) * step
58
```

Encerrando esse bloco é criada a função update( ) que receberá rollouts e sigma_r. Dentro de sua estrutura de código é criado o objeto step, inicialmente parametrizado com uma matriz de zeros no formato de theta. Em seguida é criado um laço de repetição onde para as variáveis temporárias r_pos, r_neg e d em rollouts, step tem seu valor atualizado com o valor da soma entre seu valor atual e a diferença entre r_pos e r_neg, multiplicado pelo valor de d. Finalizando, theta também tem seu valor atualizado, recebendo o valor da divisão entre leaning_rate e nb_best_directions multiplicado por sigma_r, multiplicado pelo valor de step.

Vamos entender a lógica do que está acontecendo aqui, lembre-se que conceitualmente a ideia é termos o processo de aprendizado por reforço as ações para o corpo e separadamente para a mente de nosso Agente. A nível de código isso é feito reutilizando estados do corpo para novos processamentos da mente do mesmo.

Para alguns autores, o mecanismo de rollouts é caracterizado desta maneira, nosso agente em um determinado estado, quando penalizado, reaproveitará todo processamento realizado para o corpo, mudando somente as diretrizes lógicas em busca de recompensa positiva. Como dito anteriormente no embasamento teórico, nosso Agente está caminhando de um ponto A para um ponto B, quando o mesmo erra uma ação, ele não precisa desaprender a caminhar, mas apenas desaprender a ação lógica incorreta.

```
59  def explore(env, normalizer, policy, direction = None, delta = None):
60      state = env.reset()
61      done = False
62      num_plays = 0.
63      sum_rewards = 0
64      while not done and num_plays < hp.episode_length:
65          normalizer.observe(state)
66          state = normalizer.normalize(state)
67          action = policy.evaluate(state, delta, direction)
68          state, reward, done, _ = env.step(action)
69          reward = max(min(reward, 1), -1)
70          sum_rewards += reward
71          num_plays += 1
72      return sum_rewards
73
```

Seguindo com nosso código é criada a função explore( ) que receberá env, normalizer, policy e define que direction e delta são instanciadas, porém inicialmente sem nenhum dado.

Dentro de sua estrutura de código é criado o objeto state que por sua vez, devido a sequência da leitura léxica por parte do interpretador, reseta todos estados de nosso Agente por meio da função reset( ).

Na mesma lógica done é iniciado como False, já que seu estado será alterado para True apenas quando a tarefa a ser realizada por nosso Agente for concluída; num_plays e sum_rewards são inicializados com valor 0, pois no estado inicial ainda não há uma memória criada.

Em seguida é criada uma estrutura condicional onde enquanto nem o valor de done nem o de num_plays for menor que episode_length, os objetos normalizer.observe e state tem seus valores atualizados com os dados oriundos de state. Da mesma forma action realiza alguns testes por meio da função evaluate( ) por sua vez parametrizada com os dados de state, delta e direction.

Na sequência os objetos state, reward, done e a variável vazia _ chamam a função step( ) parametrizada com os dados de action; reward por sua vez recebe o valor máximo atribuído para o mínimo de reward, isso faz com que se crie um intervalo de amostras a serem consideradas, tentando obter amostras dos melhores valores encontrados.

Finalizando, sum_rewards é atualizado com o seu próprio valor somado com o de reward, o mesmo é feito para num_plays, que por sua vez tem seu valor atualizado pela soma de seu próprio valor acrescido de 1. Por fim é retornado o valor de sum_rewards.

Repare que o que é feito aqui nada mais é do que fazer o uso daquela política de tomada de decisão baseada no último estado, porém como não havia um estado anterior, é necessário criar um estado, mesmo que com valores zerados, para que se possa ser atualizado.

```
74  def train(env, policy, normalizer, hp):
75      for step in range(hp.nb_steps):
76          deltas = policy.sample_deltas()
77          positive_rewards = [0] * hp.nb_directions
78          negative_rewards = [0] * hp.nb_directions
79          for k in range(hp.nb_directions):
80              positive_rewards[k] = explore(env, normalizer, policy,
81                                            direction = "positive",
82                                            delta = deltas[k])
83          for k in range(hp.nb_directions):
84              negative_rewards[k] = explore(env, normalizer, policy,
85                                            direction = "negative",
86                                            delta = deltas[k])
```

Uma vez criada toda a estrutura base, podemos agora de fato treinar a AI de nosso Agente por meio de algumas sequências de cruzamento de dados. Para isso inicialmente criamos a função train( ) que recebe env, policy, normalizer e hp.

Dentro do bloco de código associado a train( ) já de início temos um laço de repetição onde para cada step em nb_steps uma série de atualizações será realizada. Primeira delas para o objeto deltas é atribuído os valores padrão de policy.sample_deltas( ), a serem importados dos arquivos de exemplo, normalmente uma matriz preenchida de zeros. Em seguida para positive_rewards e para negative_rewards é atribuído o valor de seu índice zero multiplicado pelo valor de nb_directions.

Na sequência, após alocar espaço para estes dados, hora de preenche-los, e isso será feito por um segundo laço de repetição onde uma variável temporária k percorre toda a extensão de nb_directions, atualizando os valores de positive_rewards por meio da função explore( ) parametrizada com env, normalizer, policy e especificando manualmente que para este objeto direction será “positive” e que delta receberá seus valores na posição k. O mesmo é feito com um terceiro laço de repetição, porém processando dados para negative_rewards.

```
87          all_rewards = np.array(positive_rewards + negative_rewards)
88          sigma_r = all_rewards.std()
89          scores = {k:max(r_pos, r_neg) for k,(r_pos,r_neg) in enumerate(zip(positive_rewards, negative_rewards))}
90          order = sorted(scores.keys(), key = lambda x:scores[x])[:hp.nb_best_directions]
91          rollouts = [(positive_rewards[k], negative_rewards[k], deltas[k]) for k in order]
92          policy.update(rollouts, sigma_r)
93          reward_evaluation = explore(env, normalizer, policy)
94          print('Step:', step, 'Reward:', reward_evaluation)
95
```

Finalizando esse bloco mais algumas atualizações devem ser feitas, inicialmente o objeto all_rewards recebe atribuído para si uma matriz composta dos dados da soma entre positive_rewards e negative_rewards; sigma_r recebe para si os valores da média absoluta de all_rewards.

Em seguida scores recebe por sua vez um dicionário alimentado com os valores máximos de cada posição k. Em função disso é aplicado um laço de repetição em que k percorrerá cada valor de positive_rewards e de negative_rewards.

Na sequência order recebe para si como atributos os valores encontrados para scores e nb_best_directions, respectivamente ordenados dos valores maiores para os menores referentes as suas chaves. Da mesma forma rollouts remonta sua lista, agora com os valores ordenados de positive_rewards, negative_rewards e deltas. Em outras palavras aqui serão considerados os maiores valores encontrados pois estes, nos processos de reajustes dos pesos terão maior peso no processamento em contraponto a outros dados de menor relevância que poderão ser descartados conforme escolha do supervisor.

Encerrando esse bloco, é executada a função update( ) de policy, repassando como parâmetros os dados de rollouts e de sigma_r. Método semelhante é feito para reward_evaluation que chama a função explore repassando para mesma como parâmetros os dados de env, normalizer e policy.

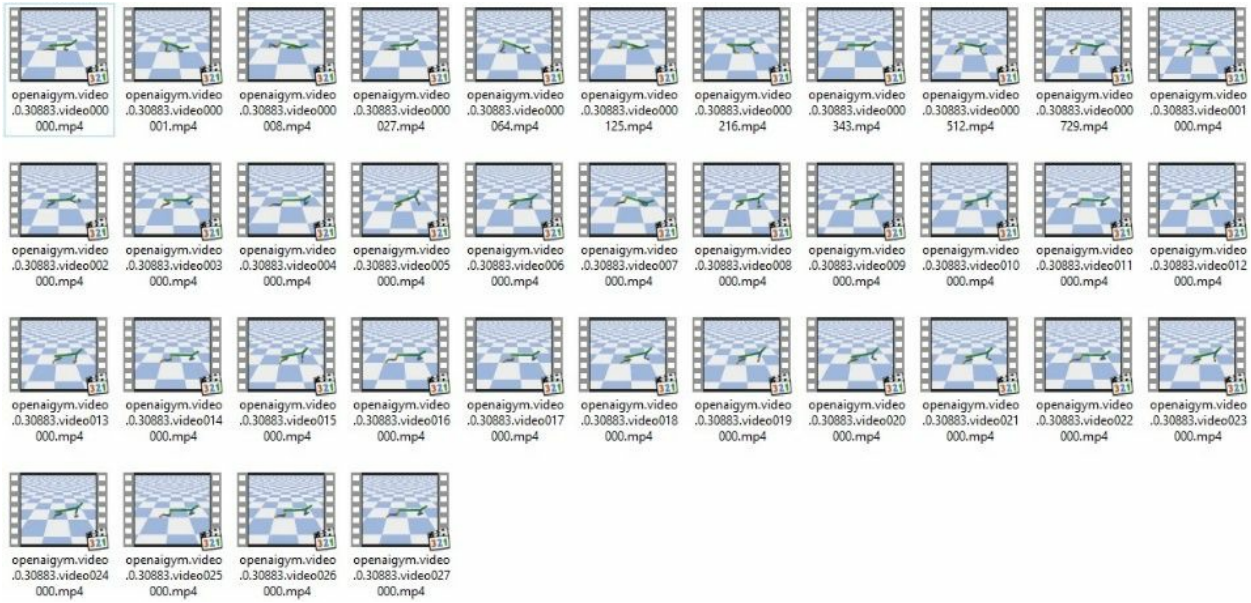
Por fim é exibido em console cada estado e sua respectiva recompensa ou penalidade, instanciando tais dados de step e de reward_evaluation, respectivamente.

```
96  def mkdir(base, name):
97      path = os.path.join(base, name)
98      if not os.path.exists(path):
99          os.makedirs(path)
00      return path
01  work_dir = mkdir('exp', 'brs')
02  monitor_dir = mkdir(work_dir, 'monitor')
03
```

Em seguida é realizada uma pequena codificação para criação de um diretório onde serão salvos algumas amostras de nosso modelo em operação.

```
104  hp = Hp()
105  np.random.seed(hp.seed)
106  env = gym.make(hp.env_name)
107  env = wrappers.Monitor(env, monitor_dir, force = True)
108  nb_inputs = env.observation_space.shape[0]
109  nb_outputs = env.action_space.shape[0]
110  policy = Policy(nb_inputs, nb_outputs)
111  normalizer = Normalizer(nb_inputs)
112  train(env, policy, normalizer, hp)
113
```

Terminando esse processo, são feitas algumas últimas definições e instancias de alguns objetos com seus “hyperparâmetros”. Se até esse momento não houve nenhum erro de sintaxe, será possível acompanhar via console o andamento de nosso Agente, assim como ao final do processo será possível visualizar várias amostras referentes a várias etapas do processo de aprendizado por reforço de nosso Agente, em formato de vídeo, no diretório criado anteriormente.



Código Completo:

```
import os
import numpy as np
import gym
from gym import wrappers
import pybullet_envs

class Hp():
    def __init__(self):
        self.nb_steps = 1000
        self.episode_length = 1000
        self.learning_rate = 0.02
        self.nb_directions = 16
        self.nb_best_directions = 16
        assert self.nb_best_directions <= self.nb_directions
        self.noise = 0.03
```

```
        self.seed = 1
        self.env_name = 'HalfCheetahBulletEnv-v0'

class Normalizer():
    def __init__(self, nb_inputs):
        self.n = np.zeros(nb_inputs)
        self.mean = np.zeros(nb_inputs)
        self.mean_diff = np.zeros(nb_inputs)
        self.var = np.zeros(nb_inputs)

    def observe(self, x):
        self.n += 1.
        last_mean = self.mean.copy()
        self.mean += (x - self.mean) / self.n
        self.mean_diff += (x - last_mean) * (x - self.mean)
        self.var = (self.mean_diff / self.n).clip(min = 1e-2)

    def normalize(self, inputs):
        obs_mean = self.mean
        obs_std = np.sqrt(self.var)
        return (inputs - obs_mean) / obs_std

class Policy():
    def __init__(self, input_size, output_size):
        self.theta = np.zeros((output_size, input_size))

    def evaluate(self, input, delta = None, direction = None):
        if direction is None:
            return self.theta.dot(input)
        elif direction == "positive":
            return (self.theta + hp.noise*delta).dot(input)
        else:
```

```
            return (self.theta - hp.noise*delta).dot(input)

    def sample_deltas(self):
        return [np.random.randn(*self.theta.shape) for _ in range(hp.nb

    def update(self, rollouts, sigma_r):
        step = np.zeros(self.theta.shape)
        for r_pos, r_neg, d in rollouts:
            step += (r_pos - r_neg) * d
        self.theta += hp.learning_rate / (hp.nb_best_directions * sigma

def explore(env, normalizer, policy, direction = None, delta = None
    state = env.reset()
    done = False
    num_plays = 0.
    sum_rewards = 0
    while not done and num_plays < hp.episode_length:
        normalizer.observe(state)
        state = normalizer.normalize(state)
        action = policy.evaluate(state, delta, direction)
        state, reward, done, _ = env.step(action)
        reward = max(min(reward, 1), -1)
        sum_rewards += reward
        num_plays += 1
    return sum_rewards

def train(env, policy, normalizer, hp):
    for step in range(hp.nb_steps):
        deltas = policy.sample_deltas()
        positive_rewards = [0] * hp.nb_directions
        negative_rewards = [0] * hp.nb_directions
        for k in range(hp.nb_directions):
```

```
            positive_rewards[k] = explore(env, normalizer, policy,
                                          direction = "positive",
                                          delta = deltas[k])
        for k in range(hp.nb_directions):
            negative_rewards[k] = explore(env, normalizer, policy,
                                          direction = "negative",
                                          delta = deltas[k])
        all_rewards = np.array(positive_rewards + negative_rewards)
        sigma_r = all_rewards.std()
                              scores = {k:max(r_pos, r_neg) for k,
(r_pos,r_neg) in enumerate(zip(positive_rewards, negative_rewards
                order = sorted(scores.keys(), key = lambda x:scores[x])
[:hp.nb_best_directions]
        rollouts = [(positive_rewards[k], negative_rewards[k], deltas[k
        policy.update(rollouts, sigma_r)
        reward_evaluation = explore(env, normalizer, policy)
        print('Step:', step, 'Reward:', reward_evaluation)

def mkdir(base, name):
    path = os.path.join(base, name)
    if not os.path.exists(path):
        os.makedirs(path)
    return path
work_dir = mkdir('exp', 'brs')
monitor_dir = mkdir(work_dir, 'monitor')

hp = Hp()
np.random.seed(hp.seed)
env = gym.make(hp.env_name)
env = wrappers.Monitor(env, monitor_dir, force = True)
nb_inputs = env.observation_space.shape[0]
nb_outputs = env.action_space.shape[0]
```

```
policy = Policy(nb_inputs, nb_outputs)
normalizer = Normalizer(nb_inputs)
train(env, policy, normalizer, hp)
```

Agente Autônomo que joga BreakOut (Atari)

Uma das aplicações que vem crescendo exponencialmente, quando tratamos sobre redes neurais artificiais, são as áreas que englobam visão computacional e inteligência artificial. Tenha em mente que nunca teremos um contexto totalmente isolado de aplicação, mas o contrário disso, combinando diversos recursos dos mais diversos modelos de redes neurais artificiais.

Ao meu ver, é um grande desperdício de tempo tentar classificar os modelos separando-os conforme suas funcionalidades específicas, uma vez que sempre haverão estruturas compartilhadas em mais de um modelo.

Sendo assim, raciocine que treinar um Agente para que o mesmo, por exemplo, consiga jogar um jogo de videogame, envolve o uso de redes neurais artificiais convolucionais (que lê dados a partir de imagens), deep learning (aprendizado de máquina profundo) e redes neurais artificiais intuitivas (haja visto que o propósito geral aqui é não somente aprender a jogar o jogo, mas dominá-lo e usar do mesmo aprendizado para outros jogos do mesmo tipo).

Então, apenas separando o joio do trigo, lembre-se que a área da visão computacional trabalha sobre estruturas de sensores que abstraem nosso sentido da visão, realizando um mapeamento de um ambiente 2D ou 3D, gerando dados a serem transformados para matrizes em um processo de convolução até chegar no cruzamento de dados em busca de padrões, prática comum a praticamente todos modelos de redes neurais artificiais. O pulo do gato aqui se dará pela capacidade de nosso Agente de realizar a leitura de frames em tempo real e reagir aos mesmos, inicialmente testando todas as possíveis ações, posteriormente baseado em sua memória de replay e aprendizado de máquina (retroalimentada e processada em ciclos) conseguir progredir reagindo da maneira correta as mais diversas situações.

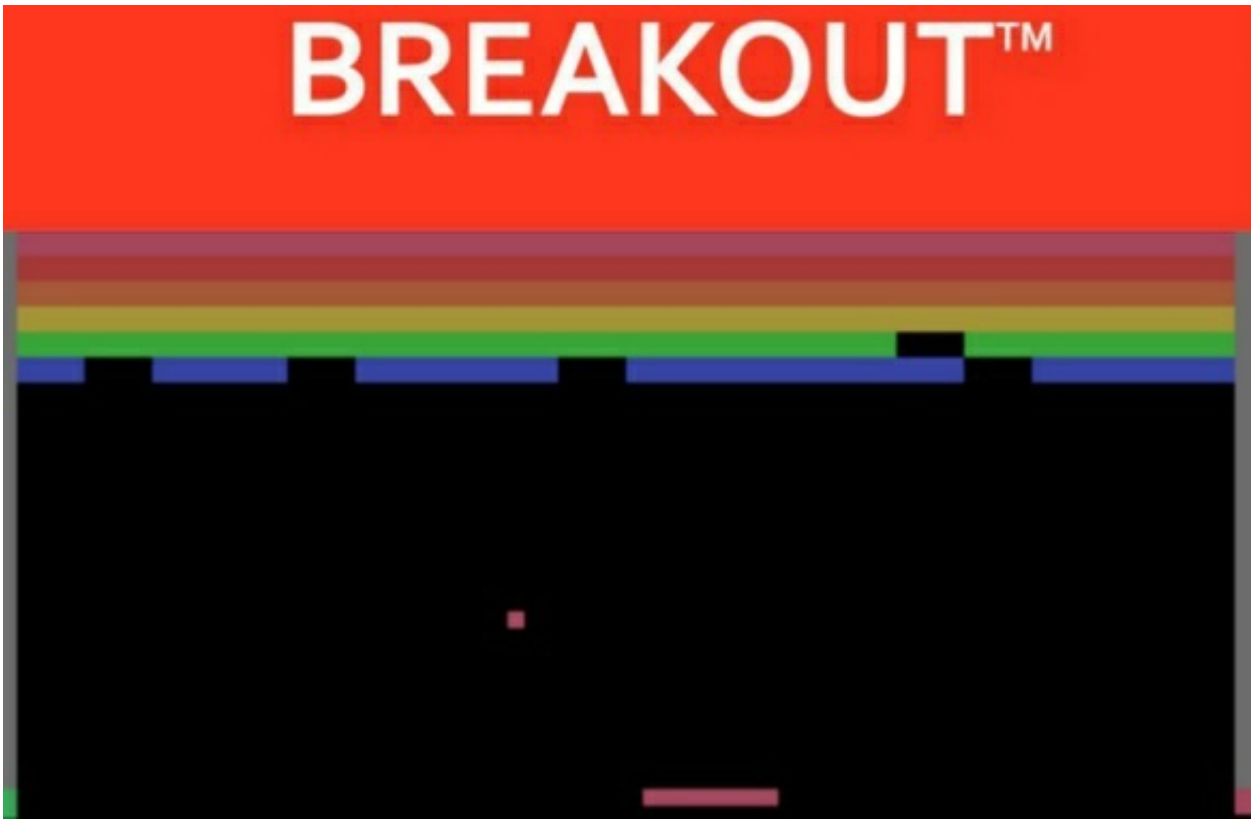
Apenas como exemplo implementaremos uma simples rede neural artificial intuitiva que por sua vez aprenderá a jogar o famoso BreakOut do Atari, porém esse modelo pode ser adaptado para qualquer fim 2D.

Talvez você esteja se perguntando sobre a complexidade da criação de uma IA para um carro autônomo, por exemplo. Novamente, tenha em mente que por baixo dos panos tudo se trata em realizar a leitura do ambiente, em tempo real, frame a frame, transliterando tais dados para arrays e realizando seu devido processamento.

De forma resumida, e novamente com propósito de organizar nossa sequência lógica dos fatos, basicamente o que faremos é:

- Instalação das bibliotecas e módulos que serão utilizados em nosso código.
- Importação das bibliotecas e módulos específicos.
- Construção da arquitetura de rede neural artificial convolucional
- Construção da arquitetura de rede neural artificial intuitiva.
- Criação da base de memória e pré-processamento dos dados.
- Carregamento da memória de curto prazo.
- Criação dos estados iniciais de nosso agente.
- Treinamento da IA para seu devido propósito.
- Definição final do modelo.
- Execução de nossa rede neural artificial intuitiva.
- Interpretação dos dados obtidos em tempo real.

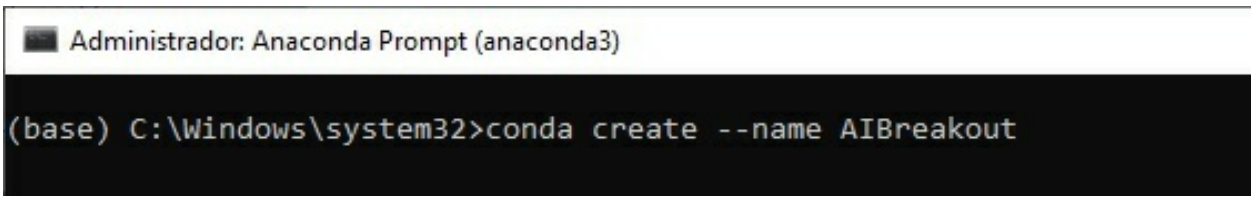
BreakOut



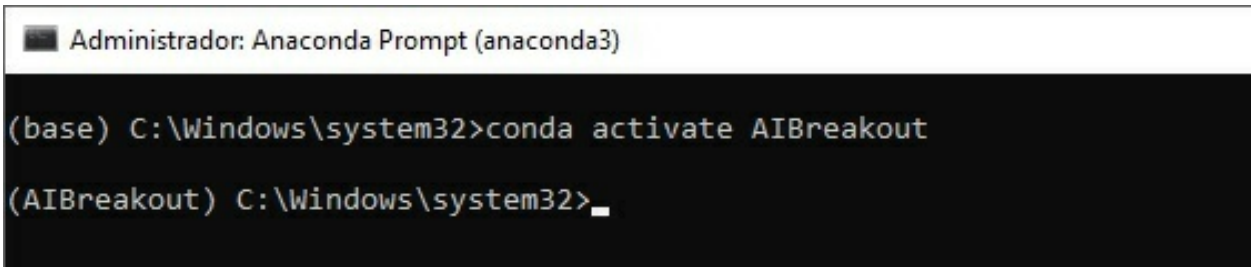
BreakOut é um jogo da Atari, desenvolvido em 1976, inicialmente para fliperamas tendo um port para o próprio console da Atari, o 2600. O objetivo do jogo é destruir todos os blocos/tijolos rebatendo uma bola sobre os mesmos ao mesmo evitando que a bola toque a margem inferior da tela. O jogador por sua vez controla uma palheta para rebater a bola que segue a trajetória respeitando a física da angulação sobre a mesma.

Instalação das Dependências

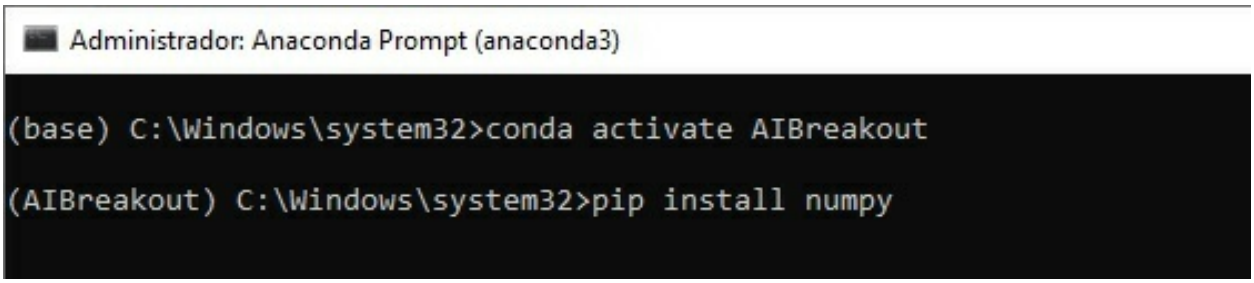
Para esse projeto em particular, optei por criar um ambiente isolado apenas com o núcleo Python e as bibliotecas que usaremos para este exemplo.



Sendo assim, via prompt Anaconda por meio do Código acima é criado o ambiente virtual isolado AIBreakout.



Da mesma forma como fizemos com o exemplo anterior, é necessário ativar este ambiente para que possamos instalar no mesmo as bibliotecas, módulos e ferramentas que faremos uso.



Para nosso exemplo faremos o uso das bibliotecas Numpy, que oferecerá uma série de ferramentas para que possamos trabalhar com dados em forma matricial; TensorFlow, que permitirá a criação de forma simples das arquiteturas de nossas redes neurais artificiais; gym que por sua vez nos permitirá implementar nossa AI diretamente nos processos do jogo BreakOut; imageio que nos será útil para o carregamento dos dados a partir de imagens/frames e por fim o modulo skimage da biblioteca SciKit-Learn, que nos permitirá a manipulação das imagens obtidas.

As instalações são feitas como de praxe, via pip.

pip install numpy

pip install tensorflow

pip install gym==0.7.4

pip install imageio

pip install scikit-image

Partindo para a prática!!!

```
1   import os
2   import random
3   import gym
4   import tensorflow as tf
5   import numpy as np
6   import imageio
7   from skimage.transform import resize
8
```

Como sempre, todo processo se inicia com as devidas importações das bibliotecas, módulos e ferramentas que usaremos ao longo de nosso código. Única especificação diferente do que fizemos anteriormente é que agora, do módulo transform da biblioteca skimage importaremos resize, que por sua vez permitirá, como o próprio nome sugere, o escalonamento da imagem obtida para um tamanho menor.

Lembre-se que uma prática comum em redes neurais convolucionais é sempre que possível, converter a escala de cores da imagem para preto e branco, assim como reduzir e padronizar os tamanhos das imagens que passarão pelo processo de convolução, ganhando assim, performance nesta camada de processamento.

```
9   ENV_NAME = 'BreakoutDeterministic-v3'
10
```

Em seguida é necessário realizar um processo um pouco diferente do convencional, repare que criamos um objeto de nome ENV_NAME, e pela forma como o mesmo foi declarado podemos deduzir que trata-se de uma palavra reservada do sistema certo? Não necessariamente do sistema, mas essa é uma palavra reservada da biblioteca gym, por meio desse objeto realizaremos o carregamento do jogo BreakOut, nesse caso, em sua versão 3.

Internamente a biblioteca gym este é o modelo de exemplo que vem na mesma para que possamos usar de seus frames como dados de entrada para nossa rede neural artificial. No site da mesma é possível encontrar outros exemplos para diferentes propósitos.

```
11  class FrameProcessor(object):
12      def __init__(self, frame_height=84, frame_width=84):
13          self.frame_height = frame_height
14          self.frame_width = frame_width
15          self.frame = tf.placeholder(shape=[210, 160, 3], dtype=tf.uint8)
16          self.processed = tf.image.rgb_to_grayscale(self.frame)
17          self.processed = tf.image.crop_to_bounding_box(self.processed, 34, 0, 160, 160)
18          self.processed = tf.image.resize_images(self.processed,
19                                                  [self.frame_height, self.frame_width],
20                                                  method=tf.image.ResizeMethod.NEAREST_NEIGHBOR)
21
```

Uma vez realizadas corretamente as devidas importações e o carregamento da estrutura do jogo BreakOut da biblioteca gym, podemos dar início a criação de nossa arquitetura de código a ser usada por nosso Agente.

Para isso inicialmente criamos uma classe de nome FrameProcessor( ) que por sua vez receberá um atributo de classe object. Essa estrutura será responsável pela conversão da matriz RGB de cada frame para uma escala de cinza, assim como o escalonamento das imagens para um tamanho menor e padrão. Esse processo é uma prática comum de ser realizada em diversos modelos de redes neurais artificiais, pois com pequenos ajustes como este obtemos um ganho de performance que justifica os mesmos.

Tenha em mente que uma matriz de cores convertida para escala de cinza torna o processo mais eficiente, escalonar as imagens para um tamanho menor padronizado também, assim como os dados de nossas matrizes em uma janela de 0 a 1 também. Sempre que possível serão realizados processos dessa natureza, puramente a fim de ganho de performance.

Internamente a estrutura dessa classe criamos nosso método construtor

__init__ parametrizado com frame_height = 84 e frame_width = 84, o que em outras palavras nos diz que assim que carregado um frame o mesmo já será escalonado para uma matriz 84x84.

Em seguida são criados alguns objetos, como frame_height e frame_width que respectivamente recebem o valor definido anteriormente 84; frame, que via função placeholder( ) cria e aloca um objeto de tamanhos [210,160 3], esta estrutura funciona internamente reservando um invólucro a ser preenchido por dados posteriormente, guardando-os como matriz nas dimensões especificadas.

É de suma importância esse tipo de conversão pois internamente, quando houverem os respectivos cruzamentos dos dados em forma matricial ou vetorial os mesmos devem estar padronizados em um só formato.

Também é criado o objeto processed, que recebe como atributos os dados da conversão de cor para escala de cinza via função rgb_to_gray( ) onde será repassado um frame, assim como pela função crop_to_bounding_box( ) esses mesmos dados obtidos até então da matriz da imagem sofrerão um corte, mantendo apenas a região de maior interesse.

Para quem está familiarizado com visão computacional, uma bounding box é uma marcação que cria uma caixa destacando, por exemplo, o rosto de uma pessoa em uma imagem, essa caixa por sua vez tem dados em forma das posições x e y onde esse rosto se encontra, em nosso exemplo, os parâmetros 34, 0, 160, 160 dizem respeito a área de maior interesse, enquanto tudo o que está fora dessa marcação é desconsiderado, tudo a fim de obter melhor performance.

Por fim, via função resize_images( ) parametrizada com os dados obtidos até então, é aplicado o método NEAREST_NEIGHBOR, que por sua vez, agrupará os dados com base em um parâmetro em comum, neste caso, empilhará frame a frame respeitando sua bounding box.

```
21
22      def __call__(self, session, frame):
23          return session.run(self.processed, feed_dict={self.frame:frame})
24
```

Finalizando o bloco de código dessa classe, é criado um método chamado __call__ que recebe uma session e um frame. Se você já é

familiarizado com TensorFlow deve se lembrar que cada pacote de um processamento é rodado dentro de uma sessão.

Aqui, esta camada de processamento deverá ser realizada a cada frame lido. Repare que a nível de código simplesmente é criada e executada uma sessão via session.run( ) onde serão processados os dados de processes que alimentarão um dicionário com dados de cada frame.

Apenas relembrando que o propósito aqui, pós leitura e cruzamento desses dados, é que os dados de entrada (cada frame obtido, em tamanho [210, 160, 3], colorido) seja convertido para um novo frame de tamanho [84, 84, 1] em escala de cinza.

```
25  class DQN(object):
26      def __init__(self, n_actions, hidden=1024, learning_rate=0.00001,
27                   frame_height=84, frame_width=84, agent_history_length=4):
28          self.n_actions = n_actions
29          self.hidden = hidden
30          self.learning_rate = learning_rate
31          self.frame_height = frame_height
32          self.frame_width = frame_width
33          self.agent_history_length = agent_history_length
34          self.input = tf.placeholder(shape=[None, self.frame_height,
35                                             self.frame_width,
36                                             self.agent_history_length],
37                                             dtype=tf.float32)
```

Em seguida damos início a criação de nossa estrutura de rede neural convolucional, que aqui terá algumas particularidades haja visto que os dados processados pela mesma retroalimentarão uma rede neural intuitiva.

Apenas relembrando, conceitualmente uma rede neural artificial convolucional é o tipo de rede neural artificial onde os dados de entrada são obtidos a partir de imagens, que passam pelo processo de convolução (onde o mapeamento de cada pixel que compõe a imagem é decodificado e transliterado para uma matriz) e finalmente terão seus dados convertidos como os neurônios de entrada de nossa rede neural.

A nomenclatura DQN costuma ser utilizada por convenção para toda e qualquer rede neural artificial intuitiva profunda, um belo nome não, encontrado na literatura como Deep Q Network.

Retomando a codificação, é criada a classe DQN( ) que receberá um

object. Dentro de sua estrutura criamos um método construtor __init__ que por sua vez recebe uma série de parâmetros, o primeiro deles n_actions definirá um tamanho específico de possíveis ações a serem tomadas (apenas abstraindo nosso exemplo, a palheta que o Agente controlará pode “não fazer nada”, assim como mover-se para direita ou para esquerda); hidden = 1024 define manualmente que a primeira camada oculta dessa rede neural terá o número fixo inicial de 1024 neurônios; learning_rate = 0,00001 define manualmente também a taxa de aprendizado, ou de quantas em quantas amostras os pesos serão atualizados; frame_height e frame_width que como visto anteriormente, definem a altura e largura do frame, respectivamente; Finalizando com agent_history_length = 4, que define que serão levadas em consideração os últimos 4 valores de estado e ação de nosso Agente.

```
38          self.inputscaled = self.input/255
39          self.conv1 = tf.layers.conv2d(inputs=self.inputscaled,
40              filters=32, kernel_size=[8, 8], strides=4,
41              kernel_initializer=tf.variance_scaling_initializer(scale=2),
42              padding="valid", activation=tf.nn.relu, use_bias=False, name='conv1')
43          self.conv2 = tf.layers.conv2d(
44              inputs=self.conv1, filters=64, kernel_size=[4, 4], strides=2,
45              kernel_initializer=tf.variance_scaling_initializer(scale=2),
46              padding="valid", activation=tf.nn.relu, use_bias=False, name='conv2')
47          self.conv3 = tf.layers.conv2d(
48              inputs=self.conv2, filters=64, kernel_size=[3, 3], strides=1,
49              kernel_initializer=tf.variance_scaling_initializer(scale=2),
50              padding="valid", activation=tf.nn.relu, use_bias=False, name='conv3')
51          self.conv4 = tf.layers.conv2d(
52              inputs=self.conv3, filters=hidden, kernel_size=[7, 7], strides=1,
53              kernel_initializer=tf.variance_scaling_initializer(scale=2),
54              padding="valid", activation=tf.nn.relu, use_bias=False, name='conv4')
```

Na sequência é criado um objeto de nome inputscaled que recebe como atributo o valor de input dividido por 255, esta é uma simples forma de escalonar os dados entre 0 e 1, ganhando assim um pouco em performance.

Logo após é criado o bloco com a estrutura convolucional em si, repare que serão 4 camadas de convolução com diferentes parâmetros, essa prática se dá porque assim “forçamos” nossa rede a encontrar os padrões certos.

A primeira camada de convolução, instanciada em objeto como conv1, por meio da função conv2d( ) recebe os parâmetros inputs com os valores atribuídos de inputscaled; filters = 32, mecanismo interno que aplicará 32 filtros de polimento; kernel_size, aqui de tamanho 8x8, realizará a leitura dos

blocos separada e isoladamente em busca de padrões, strides = 4 que pegará essas informações dos blocos obtidos por kernel_size e armazenará em vetores isolados; kernel_initializer que chama a função variance_scaling_initializer( ) parametrizada com scale = 2, o que em outras palavras significa que amostras intercaladas escolhidas de strides serão agrupadas de acordo com sua semelhança e da semelhança com um parâmetro arbitrário, dando mais pesos aos erros na identificação dos blocos de mapeamento; padding = “valid” estipula que os dados de saída desta camada de processamento sejam validados somente se estiverem todos no mesmo padrão; activation = tf.nn.relu que é o nosso velho conhecido ReLU, função de ativação que retorna valores de classificação binária; use_bias = False, prática comum em redes neurais convolucionais, o descarte da camada de processamento de bias, apenas a fim de melhor performance; encerrando com name = ‘conv1’ apenas para melhor organização e sumarização de nossa rede.

O processo é repetido para outras 3 camadas de convolução apenas alterando os tamanhos dos vetores a serem processados, isso força a rede a detectar e reforçar os padrões corretos encontrados pela mesma nesse processo de mapeamento de cada bloco de dados que compõe cada pixel da composição da imagem.

```
55          self.valuestream, self.advantagestream = tf.split(self.conv4, 2, 3)
56          self.valuestream = tf.layers.flatten(self.valuestream)
57          self.advantagestream = tf.layers.flatten(self.advantagestream)
58          self.advantage = tf.layers.dense(
59              inputs=self.advantagestream, units=self.n_actions,
60              kernel_initializer=tf.variance_scaling_initializer(scale=2),
61              name="advantage")
62          self.value = tf.layers.dense(
63              inputs=self.valuestream, units=1,
64              kernel_initializer=tf.variance_scaling_initializer(scale=2),
65              name='value')
```

Ainda em nossa classe DQN( ) damos sequência criando uma cadeia de objetos que por sua vez irão gerar aquele mecanismo de dependência abordado no exemplo anterior, onde os dados serão cruzados de forma a que sempre haverá um novo estado ou uma tomada de ação por nosso Agente, e isso se dá pela interligação de uma série de objetos e funções a serem executadas a partir do gatilho inicial. Para isso, criamos um objeto de nome valuestream, que recebe como atributo a função split( ) que separa cada

dado lido a partir das saídas apresentadas, transformando cada valor desse em um neurônio empilhado do que seria a camada de saída dessa rede (porém lembrando, aqui a saída deve se comportar diferente, ao invés de encerrar o processo, a mesma retroalimentará a rede continuamente, gerando ciclos de processamento).

Da mesma forma é criado um objeto de nome advantagestream que por sua vez recebe esses dados da camada flatten obtidos anteriormente. Em seguida é criado o objeto advantage que por sua vez será mais uma camada de rede neural artificial, dessa vez, densa, com todos neurônios interconectados.

Repare que estamos realizando a interconexão entre duas redes neurais, de forma a ler dados a partir da imagem, convertê-los de forma que possam alimentar uma rede neural densa para que seja realizada as etapas de aprendizado de máquina.

Logo, advantage por sua vez recebe como atribuição a camada de entrada da rede neural densa via dense( ), por sua vez parametrizado com inputs que conterá os dados de advantagestream como número de neurônios para essa camada de entrada; units aqui recebendo n_actions, em outras palavras, o número de possíveis ações a serem tomadas por nosso Agente; kernel_initializer e name definidos da mesma forma que na estrutura da rede neural convolucional.

Por fim, encerrando essa cadeia de código, é criado um objeto de nome value que atuará como a conhecida camada de saída de uma rede neural convencional, repare que sua única diferença em relação a camada anterior é que a mesma retornará apenas um valor. Lembre-se que todo esse processamento segue a lógica de ler uma imagem e a partir dela tomar a decisão de realizar uma ação.

```
66          self.q_values = self.value + tf.subtract(self.advantage,tf.reduce_mean(self.advantage,
67                                                                                  axis=1,
68                                                                                  keepdims=True))
69          self.best_action = tf.argmax(self.q_values, 1)
70          self.target_q = tf.placeholder(shape=[None], dtype=tf.float32)
71          self.action = tf.placeholder(shape=[None], dtype=tf.int32)
72          self.Q = tf.reduce_sum(tf.multiply(self.q_values, tf.one_hot(self.action,
73                                                                       self.n_actions,
74                                                                       dtype=tf.float32)),
75                                                                       axis=1)
76          self.loss = tf.reduce_mean(tf.losses.huber_loss(labels=self.target_q,predictions=self.Q))
77          self.optimizer = tf.train.AdamOptimizer(learning_rate=self.learning_rate)
78          self.update = self.optimizer.minimize(self.loss)
79
```

Seguindo com o código, hora de criarmos aquela estrutura compreendida anteriormente pela equação de Bellman, onde criaremos uma série de objetos que representarão para nosso agente parâmetros como seus estados, assim como suas tomadas de decisão com base em sua memória de curta duração.

Sendo assim é criado o objeto q_values, por sua vez atribuído com o resultado da função subtract( ) parametrizada com os valores de advantage subtraídos da média dos próprios valores via reduce_mean( ).

Em seguida é criado o objeto best_action, que chama a função argmax( ) parametrizada de forma a obter o último valor, ou o valor mais alto obtido a partir de q_values.

Da mesma forma é criado o objeto target_q, inicialmente criando um objeto vazio e sem forma, a receber um dado no formato float. O mesmo processo é realizado para o objeto action, alterando apenas que o mesmo espera um dado em formato int. Na sequência para action é feito o mesmo.

Logo após chegamos no objeto Q, codificado aqui de forma a receber como atributo o resultado da multiplicação das somas entre q_values, action e n_actions. Entenda que essa fórmula por si só está realizando uma versão reduzida da equação de Bellman, a fim de definir o estado atual de nosso Agente com base no estado passado (Q).

Encerrando essa parte é definida nossa função de custo loss que por sua vez nada mais está fazendo do que recebendo a média dos valores da diferença entre os valores de Q encontrados se comparado com os valores previstos para o mesmo. Obviamente quanto maior a proximidade desses dados significa melhor performance de nosso algoritmo, em função de que

temos um objetivo a alcançar, um alvo a atingir e a avaliação da performance se dá pela proximidade desse alvo, quanto mais próximo, mais eficiente, mais distante, menos eficiente é nosso algoritmo.

Ainda no encerramento é instanciado o objeto optimizer, agora definido para treinar nosso agente a partir do otimizador Adam para que os valores de leaning_rate, ou seja, da taxa de aprendizado seja equivalente ao valor definido manualmente anteriormente.

Por fim é criado um objeto muito importante, senão o mais importante deles nesse contexto que é o update, com base na função minimize( ) parametrizada com os valores de loss, é definido uma espécie de estrutura condicional onde enquanto o aprendizado de máquina não atinge um nível de excelência, o ciclo todo se repete, novamente, trabalhando com aquela ideia de estrutura de dependência, estrutura lógica que “motivará” nosso Agente a continuar vivo, realizando suas funções de forma contínua.

Importante destacar aqui que ao visualizar esses conceitos é comum imaginar que quando finalmente atingido um determinado objetivo por nosso Agente o mesmo perde seu propósito e poderia ficar estagnado ou até mesmo ser desligado. Porém, raciocine que essa continuidade simulada, por parte de processamento de nosso agente, não necessariamente tem uma estimativa de acabar, porque na estrutura que estamos elaborando, sempre haverão novas situações a serem exploradas, o ambiente inicial assim como os primeiros padrões aprendidos não apenas o equivalente a fase de treino de uma rede neural artificial, a ideia é que uma vez dominado este primeiro ambiente ou atingido um determinado objetivo, nosso Agente seja utilizado em outros contextos.

```
class ExplorationExploitationScheduler(object):
    def __init__(self, DQN, n_actions, eps_initial=1, eps_final=0.1, eps_final_frame=0.01,
                 eps_evaluation=0.0, eps_annealing_frames=1000000,
                 replay_memory_start_size=50000, max_frames=25000000):
        self.n_actions = n_actions
        self.eps_initial = eps_initial
        self.eps_final = eps_final
        self.eps_final_frame = eps_final_frame
        self.eps_evaluation = eps_evaluation
        self.eps_annealing_frames = eps_annealing_frames
        self.replay_memory_start_size = replay_memory_start_size
        self.max_frames = max_frames
        self.slope = -(self.eps_initial - self.eps_final)/self.eps_annealing_frames
        self.intercept = self.eps_initial - self.slope*self.replay_memory_start_size
        self.slope_2 = -(self.eps_final - self.eps_final_frame)/
                        (self.max_frames - self.eps_annealing_frames - self.replay_memory_start_size)
        self.intercept_2 = self.eps_final_frame - self.slope_2*self.max_frames
        self.DQN = DQN

```

Partindo para nossa próxima classe, denominada ExplorationExploitationScheduler( ) que recebe um object para si, dentro de sua estrutura é criado um método construtor __init__ que recebe DQN, n_actions, eps_initial, eps_final, eps_final_frame, eps_evaluation, eps_annealing_frames, replay_memory_start_size e max_frames. Vamos entender o que cada objeto desses realiza dentro dessa classe:

Inicialmente precisamos ter de forma clara o que essa classe em particular realiza, e nesse caso, essa classe ficará responsável por determinar uma ação a ser tomada por nosso Agente com base nos dados obtidos até então. O conceito de exploration e exploitation para alguns autores determina a maneira como nosso Agente se portará em seu ambiente, que tipo de abordagem o mesmo terá para realizar o reconhecimento e o mapeamento do mesmo, realizando a cada passo uma série de avaliações de suas tomadas de decisão assim como planejando quais serão seus próximos passos de acordo com a tendência/probabilidade de maior chance de estar correta.

Sendo assim vamos aos objetos. Toda vez que essa classe for instanciada, a mesma será alimentada com uma série de dados definidos até então, o primeiro deles n_actions, que como visto anteriormente, define um intervalo de quantas são as possíveis ações a serem tomadas por nosso Agente, ou em outras palavras, o número de possíveis ações; eps_initial por sua vez receberá um valor de probabilidade de exploração/tomada de decisão para o primeiro frame processado na memória de replay; eps_final receberá um dado probabilístico a ser usado para definir qual a próxima ação; eps_final_frames receberá um valor obtido após o processamento de todos os

frames envolvidos nesta etapa; eps_evaluation receberá valores oriundos de testes de performance, para comparação com os demais valores de epsilon, avaliando assim, a performance do aprendizado por reforço; eps_annealing_frames terá o número de frames agrupados e levados em consideração para a nova tomada de decisão; replay_memory_start_size define manualmente o número de frames que o Agente usará como referência para sua primeira leitura em busca de reconhecimento de padrões; Por fim, max_frames determina o número máximo de frames a serem usados para alimentar nosso Agente assim como serem armazenados em memória.

Repare em alguns pontos importantes, esses parâmetros, assim como tantos outros, são definidos aqui manualmente para um contexto padrão, para avaliação de nosso Agente é inclusive recomendados testes com outros valores para tais parâmetros.

Apenas realizando uma rápida leitura destes parâmetros, note que os valores de epsilon variam entre 0 e 1, porém, assim como em uma descida de gradiente de outros modelos, aqui a ideia é que estes valores de aproximem de 0.1, indicando que existe um número muito pequeno de probabilidade de ações a serem tomadas, porém essas ações serão as corretas. Raciocine que em um estado inicial nosso Agente não terá discernimento para saber se suas ações estão certas ou erradas, a medida que o mesmo vai via tentativa e erro recebendo recompensas ou penalizações, o mesmo tende a começar a guardar os padrões tidos como corretos, quanto mais treinado, maior será a taxa de acertos do mesmo, até o ponto onde ele não precisa mais tomar muitas decisões erradas, mas se basear na melhor probabilidade dentre as corretas.

Da mesma forma repare o número de frames a serem processados, frame a frame por ciclo, em processo de aprendizagem por reforço, onde para cada frame serão realizadas diversas camadas de processamento para obtenção dos seus respectivos estados e ações.

```
93          self.slope = -(self.eps_initial - self.eps_final)/self.eps_annealing_frames
94          self.intercept = self.eps_initial - self.slope*self.replay_memory_start_size
95          self.slope_2 = -(self.eps_final - self.eps_final_frame)/
96                           (self.max_frames - self.eps_annealing_frames - self.replay_memory_start_size)
97          self.intercept_2 = self.eps_final_frame - self.slope_2*self.max_frames
98          self.DQN = DQN
99
```

Retomando as últimas linhas desse bloco de código, é criado o objeto slope, que para alguns autores é a nomenclatura usual para uma tendência de

ação. Logo, slope recebe como atributo o resultado da divisão entre a subtração de eps_initial e eps_final pelo valor de eps_annealing_frames.

Também é criado o objeto intercept que recebe atribuído para si o resultado da multiplicação da subtração entre eps_initial e slope pelo valor em replay_memory_start_size.

Do mesmo modo é criado slope_2 que recebe o resultado da subtração entre eps_final e eps_final_frame dividido pela subtração entre os valores de max_frames, eps_annealing_frames e replay_memory_start_size.

Da mesma maneira é criado intercept_2 que recebe como atributos os valores obtidos pela subtração entre eps_final_frame e slope_2, multiplicado pelo valor de max_frames.

Entendendo este cruzamento de dados em particular, note que aqui estamos definindo um valor de probabilidade mínima e máxima para tomada de decisões de nosso agente, e este intervalo é construído com base no número de frames processados, uma vez que não teremos uma janela fixa, mas de tamanho variável, estreitando essa janela diretamente proporcional aos valores dos estados e tomadas de decisão de nosso agente.

Novamente, como mencionado nos parágrafos anteriores, o número de possíveis ações a serem tomadas por nosso agente no início de sua “vida” é grande, porém, conforme o mesmo aprende o que é errado e vai descartando tais ações de sua memória, proporcionalmente o mesmo terá uma janela menor de número de ações a serem tomadas.

Finalizando esse bloco, é criado uma espécie de laço de herança da classe atual com DQN( ) criado anteriormente e aqui instanciado para o objeto de classe DQN.

```
100     def get_action(self, session, frame_number, state, evaluation=False):
101         if evaluation:
102             eps = self.eps_evaluation
103         elif frame_number < self.replay_memory_start_size:
104             eps = self.eps_initial
105         elif frame_number >= self.replay_memory_start_size and frame_number <
106                                 self.replay_memory_start_size + self.eps_annealing_frames:
107             eps = self.slope*frame_number + self.intercept
108         elif frame_number >= self.replay_memory_start_size + self.eps_annealing_frames:
109             eps = self.slope_2*frame_number + self.intercept_2
110         if np.random.rand(1) < eps:
111             return np.random.randint(0, self.n_actions)
112         return session.run(self.DQN.best_action, feed_dict={self.DQN.input:[state]})[0]
113
```

Encerrando essa classe, é criada a função get_action( ) parametrizada com session, frame_number e state, além de já definir evaluation como False. Dentro dessa estrutura é criado uma condicional onde para evaluation é criada a variável eps que recebe como atributo eps_evaluation. Isso pode parecer um tanto quanto confuso, mas raciocine que ao gatilho inicial de nosso Agente, o mesmo não terá dados passados, de feedback, para saber avaliar sua própria performance, logo, este parâmetro deve ainda ser construído, em função disso criamos essa estrutura que inicia dessa forma.

Uma tomada de ação normalmente se dará a partir da leitura de estados passados, porém a primeira tomada de decisão será totalmente aleatória por parte de nosso agente, haja visto que o mesmo apenas reconhece um ambiente a ser explorado.

Dessa forma, caso não haja este dado inicial (e não existe mesmo), é levado em consideração que se o valor de frame_number for menor que o valor de replay_memory_start_size, eps recebe os dados de eps_initial. Este é o gatilho inicial de processamento de nosso Agente em para sua primeira ação e que se repetirá inúmeras vezes.

Caso essa condição não seja válida, é realizada uma terceira verificação onde na verdade estamos realizando uma validação onde se os valores de frame_number forem iguais ou maiores que os valores de replay_memory_start_size e frame_number for menor que os valores da soma entre replay_memory_start_size e eps_annealing_frames, eps recebe atribuído a si o resultado da multiplicação das somas entre frame_number e intercept por slope. O mesmo é feito em uma segunda validação onde se o

valor de frame_number for maior ou igual ao valor da soma entre replay_memory_start_size com eps_annealing_frames, eps recebe o resultado da soma entre frame_number por intercept_2 multiplicado por slope_2.

Note que simplesmente estamos forçando a criação de um estado inicial para que nosso agente tome sua primeira ação, uma vez que não existe uma codificação somente para a primeira ação e outra para as demais, logo, simplesmente realizamos esse cruzamento destes dados para criar um ambiente inicial a ser reconhecido por nosso agente, de forma que o mesmo consiga ler alguns valores, o suficiente para sua primeira tomada de decisão.

Finalizando esse bloco é criada uma última estrutura condicional onde se um número inicial aleatório, gerado via função random.rand( ) for menor que o valor de eps, é retornado um número para n_actions. Em outras palavras, até então criamos a estrutura do gatilho que acionará pela primeira vez nosso Agente, essa função em particular agora irá gerar a semente que alimentará esse gatilho, logo nos primeiros ciclos de processamento essa estrutura será validada e o gatilho será acionado, “ligando” nosso Agente.

Como mencionado anteriormente, por uma questão de processamento, toda camada de execução de uma rede neural criada via TensorFlow deve rodar em uma sessão, isso aqui é feito repassando para session.run( ) os primeiros valores para best_action e para input:[state] em sua posição 0 do índice, em outras palavras, é iniciada uma sessão com uma ação e um estado inicial.

Como esse conteúdo está se tornando mais extenso que o do modelo anterior, vamos dar uma pausa e revisar alguns pontos abordados até então em nosso atual exemplo.

Até o momento criamos a estrutura responsável por realizar a leitura e conversão frame a frame de nosso jogo, assim como a rede neural convolucional responsável por converter tais dados de imagem para matrizes numéricas. Da mesma forma criamos uma estrutura de rede neural artificial densa, as primeiras estruturas dedicadas a criar a “consciência” de nosso agente baseada em sensores de leitura e interpretação do ambiente, assim como seus estados e capacidade de tomadas de decisão. Tudo até então, apesar de já ocupar um número considerável de linhas de código, é apenas a

base de nosso Agente.

Dando prosseguimento, criaremos agora as estruturas lógicas que desencadearão a continuidade do processamento de nosso agente, simulando a temporalidade constante da inteligência artificial do mesmo.

```
114  class ReplayMemory(object):
115      def __init__(self, size=1000000, frame_height=84, frame_width=84,
116                   agent_history_length=4, batch_size=32):
117          self.size = size
118          self.frame_height = frame_height
119          self.frame_width = frame_width
120          self.agent_history_length = agent_history_length
121          self.batch_size = batch_size
122          self.count = 0
123          self.current = 0
124          self.actions = np.empty(self.size, dtype=np.int32)
125          self.rewards = np.empty(self.size, dtype=np.float32)
126          self.frames = np.empty((self.size, self.frame_height,
127                                  self.frame_width), dtype=np.uint8)
128          self.terminal_flags = np.empty(self.size, dtype=np.bool)
129          self.states = np.empty((self.batch_size, self.agent_history_length,
130                                  self.frame_height, self.frame_width),
131                                 dtype=np.uint8)
132          self.new_states = np.empty((self.batch_size, self.agent_history_length,
133                                      self.frame_height, self.frame_width),
134                                     dtype=np.uint8)
135          self.indices = np.empty(self.batch_size, dtype=np.int32)
136
```

Seguindo com nosso exemplo atual, é criada a classe ReplayMemory( ) que por sua vez recebe object. Dentro de si, como de costume, é criado um método construtor __init__ que recebe size, frame_height, frame_width, agente_history_length e batch_size. Alguns destes já vistos anteriormente, porém agora cruzando dados de forma a simular a temporalidade.

Este conceito pode parecer bastante abstrato, e de fato, é complexo traduzirmos em palavras a passagem do tempo, mas tenha em mente que em uma rede neural convencional todo processo tem um início, meio e fim, sendo que ao fim do mesmo, não existe mais o conceito de tempo de processamento.

Já para uma rede neural artificial intuitiva esse conceito é entendido de outra forma, uma vez “ligado” nosso agente o mesmo estará sempre em processamento. Assim como nossos cérebro está sempre operante (mesmo

enquanto dormimos) aqui já que estamos abstraindo o funcionamento do mesmo, estamos criando estruturas lógicas que, em seu processo de simulação de uma inteligência, estarão realizando diversas camadas de processamento, seja em primeiro ou segundo plano, onde não há intervalos de tempo definidos, mas processamento constante em decorrência do tempo.

Dessa maneira, são instanciados os objetos size, frame_height, frame_width, agente_history_length e batch_size com suas variáveis homônimas, porém temos agora alguns objetos novos, o primeiro deles count, inicialmente com valor 0 atribuído e current também atribuído da mesma forma, com 0.

Em seguida é criada a estrutura que pre-alocará blocos de memória para armazenamento de estados. Isso é feito para actions, rewards, frames e terminal_flags por meio da função np.empty( ), apenas diferenciando que para cada objeto destes o valor atribuído estará em um formato específico. Para actions em formato int32, para rewards, float32, para frames uint8 e para terminal_flags em formato booleano.

Criada a estrutura básica de pre-alocação de memória, podemos também realizar algumas alocações para os novos estados em uma mini batch a ser sempre atualizada pelos ciclos de processamento. Então, states aqui recebe como atribuição espaços vazios alocados para batch_size, agente_history_length, frame_height e frame_width. Exatamente a mesma estrutura de código é criada para new_states. Por fim é atribuído o valor de batch_size para índices.

```
137     def add_experience(self, action, frame, reward, terminal):
138         if frame.shape != (self.frame_height, self.frame_width):
139             raise ValueError('Dimensão do frame está incorreta!')
140         self.actions[self.current] = action
141         self.frames[self.current, ...] = frame
142         self.rewards[self.current] = reward
143         self.terminal_flags[self.current] = terminal
144         self.count = max(self.count, self.current+1)
145         self.current = (self.current + 1) % self.size
146
```

Na sequência é criada a função add_experience( ) parametrizada com action, frame, reward e terminal. Dentro de seu bloco é criada uma estrutura condicional onde se o tamanho do frame for diferente daqueles

valores estabelecidos anteriormente para frame_height e frame_width (84x84), é levantada uma exceção, apontando para o usuário que houve algum erro por parte do escalonamento da imagem.

Também são realizados alguns cruzamentos de dados utilizando agora de nosso objeto current anteriormente criado mas até então sem uso. Repare que count (também inutilizado até então) receberá atribuído para si o valor máximo encontrado si próprio e em current somado de 1. O objeto current por sua vez ao final receberá atribuído para si o valor do módulo de current + 1 e size.

```
147     def _get_state(self, index):
148         if self.count is 0:
149             raise ValueError("A memória de replay está vazia!")
150         if index < self.agent_history_length - 1:
151             raise ValueError("Indice mínimo deve ser 3")
152         return self.frames[index-self.agent_history_length+1:index+1, ...]
153
```

Quando estamos desenvolvendo algum programa que dentre suas funções, muitas delas dependem de interação com o usuário, é normal criar estruturas de validação, tentando contornar todos os erros previstos a serem cometidos pelo usuário. Em um quadro onde o usuário é humano, tentamos prever as poucas possíveis prováveis ações que o mesmo pode realizar e criamos validadores para isso. Em nosso exemplo atual, nosso Agente por sua vez testará todas as possíveis possibilidades e ao mínimo erro de cruzamento dos dados toda a rede irá suspender sua atividade.

Pensando nisso, precisamos aqui também criar estruturas validadoras, a primeira delas por meio da função _get_state( ). Note que essa função por sua vez está fora do escopo global de acordo com a sintaxe utilizada no momento de sua declaração.

Dentro da mesma criamos uma estrutura condicional onde se o valor de count for 0, será levantado um erro de memória de replay vazia. Do mesmo modo criamos uma segunda estrutura condicional onde se o valor de index for menor que o valor de agente_history_length subtraído de 1, é levantado um erro dizendo que o valor mínimo para que se obtenha um índice é 3.

Por fim, é retornado para frames uma estrutura de autopreenchimento que nada mais fará do que validar os requisitos mínimos

de count e index.

```
154     def _get_valid_indices(self):
155         for i in range(self.batch_size):
156             while True:
157                 index = random.randint(self.agent_history_length, self.count - 1)
158                 if index < self.agent_history_length:
159                     continue
160                 if index >= self.current and index - self.agent_history_length <= self.current:
161                     continue
162                 if self.terminal_flags[index - self.agent_history_length:index].any():
163                     continue
164                 break
165             self.indices[i] = index
166
```

Da mesma forma criaremos uma estrutura para validação apenas do índice, caso possa haver alguma inconsistência por parte dele. Para isso criamos a função _get_valid_indices( ) sem parâmetros mesmo. Dentro dela inicialmente é criado um laço de repetição que fará a leitura dos valores em batch_size. A partir desse ponto é criada uma estrutura condicional que estipula que enquanto a condição for verdadeira, o objeto index recebe como atributo um número a ser gerado aleatoriamente via random.randint( ) respeitando a métrica de agente_history_length e do valor de count subtraído 1.

Em seguida é criado dentro da estrutura condicional atual uma nova estrutura condicional onde se o valor de index for menor que o valor de agente_history_length é para ir para próxima condição (algo muito parecido com as estruturas de switch/case de outras linguagens de programação).

A próxima estrutura condicional por sua vez verifica se o valor de index é igual ou maior ao valor de current e se o valor de agente_history_length é igual ou menor ao valor de current. Caso válida essa condição é dito ao interpretador que pule para a próxima condicional.

A última estrutura condicional verifica se terminal_flags em seu conteúdo possui dados para index (de qualquer tipo). Assim que atingido esse estado a estrutura validadora cessa e é retornado para índices o valor atribuído a index.

```
167     def get_minibatch(self):
168         if self.count < self.agent_history_length:
169             raise ValueError('ERRO: Não foi possível alocar espaço na memória.')
170
```

Na sequência também é criada uma função dedicada a ser o gatilho para que nossa memória de curta duração de nosso Agente seja alimentada. Isso é feito simplesmente criando uma função de nome get_minibatch( ) onde dentro da mesma é criada uma estrutura condicional que verifica se o valor de count é menor do que o valor de agente_history_length. Raciocine que essa estrutura está diretamente ligada aos validadores criados anteriormente, de forma que caso haja alguma inconsistência é levantado um erro de alocação de espaço na memória.

```
171         self._get_valid_indices()
172         for i, idx in enumerate(self.indices):
173             self.states[i] = self._get_state(idx - 1)
174             self.new_states[i] = self._get_state(idx)
175         return np.transpose(self.states, axes=(0, 2, 3, 1)),
176                             self.actions[self.indices],
177                             self.rewards[self.indices],
178                             np.transpose(self.new_states,
179                             axes=(0, 2, 3, 1)),
180                             self.terminal_flags[self.indices]
181
```

Indentado no bloco de código da função get_minibatch( ) instanciamos _get_valid_indices( ) realizando alguns cruzamentos de dados. Repare que neste momento o que faremos é criar uma estrutura que retroalimentará esses dados em forma de ciclo, uma vez que queremos continuidade de processamento de dados por parte de nosso Agente.

Sendo assim, criamos um laço de repetição onde a cada interação com índices, states em sua posição i recebe os dados de _get_state( ) parametrizado com o valor de idx subtraído de 1, em outras palavras estamos retroalimentando o último valor de states com o último valor de índice. Do mesmo modo new_states em sua posição i recebe os dados de _get_state( ) parametrizado com idx. De forma rápida podemos entender que essa estrutura por si realiza a atualização necessária para que se obtenha um novo estado que substituirá o estado atual.

Em seguida é gerado um retorno bastante incomum, haja visto que comumente retornamos um dado ou valor simples, aqui nesse caso estamos retornando toda uma gama de parâmetros para retroalimentará as estruturas de validação. Isso se dá pelo retorno da função transpose( ) parametrizada

com states, action e, rewards, recursivamente aplicando transpose( ) também para new_states e para os valores de índices de terminal_flags.

```
182  def learn(session, replay_memory, main_dqn, target_dqn, batch_size, gamma):
183      states, actions, rewards, new_states, terminal_flags = replay_memory.get_minibatch()
184      arg_q_max = session.run(main_dqn.best_action, feed_dict={main_dqn.input:new_states})
185      q_vals = session.run(target_dqn.q_values, feed_dict={target_dqn.input:new_states})
186      double_q = q_vals[range(batch_size), arg_q_max]
187      target_q = rewards + (gamma*double_q * (1-terminal_flags))
188      loss, _ = session.run([main_dqn.loss, main_dqn.update],
189                            feed_dict={main_dqn.input:states,
190                                       main_dqn.target_q:target_q,
191                                       main_dqn.action:actions})
192      return loss
193
```

Terminados os procedimentos validadores e de retroalimentação, podemos finalmente criar o mecanismo de aprendizado de nosso Agente. Isso se dará criando a função learn( ) por sua vez parametrizada com dados de session, replay_memory, main_dqn, target_dqn, batch_size e gamma.

Dentro de sua estrutura os objetos states, actions, rewards, new_states e terminal_flags recebem como atributo replay_memory repassando seus dados para get_minibatch( ). Também é criado o objeto arg_q_max que em uma sessão própria roda main_dqn.best_action alimentado com os dados de new_states. Da mesma forma q_vals roda em sessão q_values alimentado com os dados de new_states.

Na sequência é criado o objeto double_q que por sua vez recebe os dados de q_vals, porém apenas os melhores valores encontrados, já que esses nessa situação são oriundos de arg_q_max.

O objeto target_q agora finalmente instancia rewards somado do valor de gamma multiplicado pelo valor de double_q, multiplicado pelo valor de terminal_flags subtraído de 1. Por mais confuso que isso possa parecer de momento, preste bastante atenção, estamos simplesmente transformando alguns dados obtido em todas as ações anteriores e convertendo-os agora em forma de uma recompensa a ser aplicada sobre nosso Agente.

Encerrando essa parte, loss e uma variável por hora vazia _ rodam uma sessão onde é realizado o cruzamento dos dados de input em relação a states, target_q em relação a ele mesmo e action em relação a actions. Entendendo esta etapa, aqui são criados os laços de processamento cíclico de nossa rede neural, já que a mesma estará sempre lendo estados passados e atuais,

realizando tomadas de ação e retroalimentando a rede atualizando sequencialmente esses dados.

Por fim, apenas para obtermos um retorno, é retornado um dado/valor para loss, podemos usar desta informação posteriormente para avaliação da performance do modelo.

```
194  class TargetNetworkUpdater(object):
195      def __init__(self, main_dqn_vars, target_dqn_vars):
196          self.main_dqn_vars = main_dqn_vars
197          self.target_dqn_vars = target_dqn_vars
198
```

Dando prosseguimento em nosso código, vamos criar uma estrutura de validação para a atualização da rede neural em si. Para este propósito inicialmente criamos a classe TargetNetworkUpdater( ) que recebe um object. Como de costume é criado um método construtor __init__ que dessa vez receberá main_dqn_vars e target_dqn_vars, nas linhas abaixo são criados os respectivos objetos.

```
199      def _update_target_vars(self):
200          update_ops = []
201          for i, var in enumerate(self.main_dqn_vars):
202              copy_op = self.target_dqn_vars[i].assign(var.value())
203              update_ops.append(copy_op)
204          return update_ops
205
```

Também é criada para essa classe a função _update_target_vars( ). Dentro de seu bloco é criado o objeto update_ops que por hora possui como atributo apenas uma lista vazia. Na sequência é criado um laço de repetição onde para cada repetição do mesmo é realizada a contagem de main_dqn_vars. Dentro desse laço é criado o objeto copy_op que recebe os valores de target_dqn_vars como valor. Finalizando, acrescentamos os valores de copy_op em update_ops, retornando o valor de update_ops.

```
206      def __call__(self, sess):
207          update_ops = self._update_target_vars()
208          for copy_op in update_ops:
209              sess.run(copy_op)
210
```

Agora é chamada uma função padrão do Python chamada __call__( ) parametrizada com sess. Dentro de seu código é determinado que os valores de update_ops devem ser equiparados aos de _update_target_vars( ). Como você já deve ter percebido, boa parte desses códigos criam estruturas de dependência, onde sempre há um dado a se buscar, a se atualizar, a continuar buscando... para que possamos simular a continuidade de nosso Agente.

Encerrando esse bloco é criado um laço de repetição que simplesmente percorre os valores de update_ops e os insere em copy_op, repassando os mesmos para uma sessão a ser executada. O que fará uma espécie de sincronização entre as redes neurais quando carregadas e processadas.

```
211  def generate_gif(frame_number, frames_for_gif, reward, path):
212      for idx, frame_idx in enumerate(frames_for_gif):
213          frames_for_gif[idx] = resize(frame_idx, (420, 320, 3),
214                                       preserve_range=True, order=0).astype(np.uint8)
215
216      imageio.mimsave(f'{path}{"ATARI_frame_{0}_reward_{1}.gif".format(frame_number, reward)}',
217                      frames_for_gif, duration=1/30)
218
```

Logo após o bloco de código anterior criamos (novamente fora da última classe) a função generate_gif( ) que recebe um frame_number, frames_for_gif, reward e path. Esta por sua vez será a função que com base nas imagens processadas até o momento irá gerar como retorno um gif mostrando diferentes etapas do andamento de nosso Agente, em outras palavras, será gerado um gif animado com amostras de sequências de jogadas sendo realizadas por nosso Agente para que possamos acompanhar o mesmo “jogando”.

Então, é criado um laço de repetição em frame_idx a ser preenchido com os dados contados de frames_for_gif. O que esse laço por sua vez realiza é a equiparação dos dados idx de frames_for_gif com frame_idx, em cada dimensão de sua matriz, mantendo seu formato original por meio do parâmetro preserve_range = True e codificando esses dados em formato uint8.

Finalizando, é criada a estrutura que salvará esse gif animado, isso é feito por meio da função mimsave( ) da biblioteca imageio. Aqui simplesmente é repassado o diretório onde esse gif será salvo, assim como o seu nome contendo o número do frame e sua respectiva recompensa, além é claro dos parâmetros de onde virão esses dados, nesse caso de

frames_for_gif, numa amostragem de 1 frame a cada 30.

```
219  class Atari(object):
220      def __init__(self, envName, no_op_steps=10, agent_history_length=4):
221          self.env = gym.make(envName)
222          self.process_frame = FrameProcessor()
223          self.state = None
224          self.last_lives = 0
225          self.no_op_steps = no_op_steps
226          self.agent_history_length = agent_history_length
227
```

Chegou a hora de finalmente criamos nossa classe Atari( ) que recebe um object e que a partir desse ponto começaremos a de fato instanciar objetos do jogo, carregando certos dados dos jogos amostra da biblioteca gym.

Inicialmente é, como sempre, criado um método construtor __init__ que recebe envName, no_op_steps e agente_history_length para que se faça o cruzamento desses dados com os demais.

Criando seus respectivos objetos é criado env que recebe como atributo gym.make por sua vez parametrizado com envName. Note que aqui, estamos importando o ambiente do jogo BreakOut, em sua versão 3 como havíamos especificado lá no início de nosso código logo após as importações das bibliotecas. Por convenção, essa variável fica responsável por carregar e instanciar qual jogo da biblioteca gym estamos usando para exemplo.

Em seguida é criado process_frame que agora é chama a função FrameProcessor( ); state inicialmente é parametrizada em None, ou seja, sem dados iniciais aqui predefinidos, uma vez que serão importados de outra classe;

last_lives aqui é um objeto padrão da biblioteca gym, basicamente sempre virá parametrizado com 0 pois a ideia é que independente do jogo, há um número de vidas de nosso personagem, quando essas chegam a 0 o jogador perde; no_op_steps e agente_history_length instanciam suas respectivas variáveis.

```
239     def step(self, sess, action):
240         new_frame, reward, terminal, info = self.env.step(action)
241         if info['ale.lives'] < self.last_lives:
242             terminal_life_lost = True
243         else:
244             terminal_life_lost = terminal
245         self.last_lives = info['ale.lives']
246         processed_new_frame = self.process_frame(sess, new_frame)
247         new_state = np.append(self.state[:, :, 1:],
248                               processed_new_frame,
249                               axis=2)
250         self.state = new_state
251         return processed_new_frame, reward, terminal, terminal_life_lost, new_frame
252
```

Dando sequência é criada uma função padrão para a biblioteca gym que é a função step( ), que por sua vez recebe sess e action, e age diretamente sobre a passagem de tempo simulada no jogo. Tenha em mente que sempre é feita uma determinada programação interna ao jogo, seja por sua engine ou por algum parâmetro específico onde é determinado que no espaço de tempo de 1 segundo serão processados um número x de frames de forma uniforme e sequencial.

Aqui inicialmente são criadas as variáveis new_frame, reward, terminal e info, todas elas recebendo os dados de env.step(action), uma vez que como dito anteriormente, há uma programação implícita sobre a passagem do tempo no jogo, normalmente referenciada como step.

Na sequência é criado uma estrutura condicional onde se o dado/valor de ale.lives de info for menor que last_lives, terminal_life_lost é setada como True. Em outras palavras, cada vez que o Agente erra em sua jogada e, nesse jogo, deixa a bolinha tocar a base da tela, uma vida é descontada, quando ele perde sua última vida, terminal_life_lost é acionada sinalizando isso.

Caso a condição acima não seja alcançada, terminal_life_lost tem seu valor equiparado com o de terminal, haja visto que o número de vidas iniciais de nosso personagem no jogo pode ser parametrizada manualmente.

Também é atribuído para last_lives o valor atual de ale.lives de info, da mesma forma processed_new_frame agora é atualizada com os valores oriundos de process_frame, por sua vez instanciando dados de sess e new_frame.

Em seguida para new_state, objeto que de acordo com sua nomenclatura,

está criando dados para o novo estado de nosso Agente, recebe os dados de toda matriz de state para processed_new_frame em seu eixo 2.

Finalmente, após cruzados e processados os dados dessa estrutura, é retornado os novos dados/valores para processed_new_frame, reward, terminal, terminal_life_lost e new_frame.

```
253    def clip_reward(reward):
254        if reward > 0:
255            return 1
256        elif reward == 0:
257            return 0
258        else:
259            return -1
260
```

Na sequência é criada a função clip_reward( ) que recebe uma recompensa via reward, porém, raciocine que essa função será o gatilho a ser acionado onde de acordo com o contexto, nosso Agente será recompensado ou penalizado.

Para isso, simplesmente criamos uma estrutura condicional onde se o valor de reward for maior que 0 é retornado o valor 1, se o valor de reward for igual a 0 é retornado o 0 e caso nenhuma das condições seja alcançada é retornado o valor -1. Esses três estados se dão porque nosso Agente de acordo com sua ação pode ser recompensado, penalizado ou pode simplesmente não acontecer nada com o mesmo caso ele realize alguma tomada de decisão computada como irrelevante para o contexto.

Apenas relembrando os conceitos teóricos iniciais, uma recompensa positiva se dará quando o agente der um passo em direção ao objetivo, quando o mesmo conseguir ficar mais próximo de seu objetivo. Da mesma forma uma penalização é aplicada sempre que de acordo com a ação ele acaba se afastando de seu objetivo, e existirão também situações onde a ação de nosso agente simplesmente não irá interferir nesse progresso.

```
261   tf.reset_default_graph()
262   MAX_EPISODE_LENGTH = 18000
263   EVAL_FREQUENCY = 200000
264   EVAL_STEPS = 10000
265   NETW_UPDATE_FREQ = 10000
266   DISCOUNT_FACTOR = 0.99
267   REPLAY_MEMORY_START_SIZE = 50000
268   MAX_FRAMES = 30000000
269   MEMORY_SIZE = 1000000
270   NO_OP_STEPS = 10
271   UPDATE_FREQ = 4
272   HIDDEN = 1024
273   LEARNING_RATE = 0.00001
274   BS = 32
275   PATH = "output/"
276   SUMMARIES = "summaries"
277   RUNID = 'run_1'
278   os.makedirs(PATH, exist_ok=True)
279   os.makedirs(os.path.join(SUMMARIES, RUNID), exist_ok=True)
280   SUMM_WRITER = tf.summary.FileWriter(os.path.join(SUMMARIES, RUNID))
281
```

Seguindo com nosso código podemos agora, caso você venha testando o mesmo etapa após etapa de criação, resetar as ações realizadas até o momento que muito provavelmente ainda estão carregadas na memória, para que possamos agora inserir alguns parâmetros específicos para o núcleo de nossa biblioteca gym. O reset em si é feito simplesmente rodando a função tf.reset_default_graph( ) e na sequência vamos começar a definir alguns parâmetros de controle para nosso Agente. Tais parâmetros podem serem alterados normalmente, inclusive é recomendado que se realizem testes com diferentes parâmetros para avaliar a performance do modelo.

Partindo para a parametrização, inicialmente temos MAX_EPISODE_LENGTH com valor atribuído 18000, o que em outras palavras nos diz que esse número de ciclos equivale a mais ou menos 5 minutos jogando; EVAL_FREQUENCY setado em 200000 define de quantos em quantos frames processados será realizada uma avaliação da performance do modelo; EVAL_STEPS com valor 10000 atribuído para si define o número de frames a serem usados para avaliação; NETW_UPDATE_FREQ com valor 10000 define um tamanho de dados das

ações usadas por nosso agente que serão levadas em conta para o processo de aprendizado.

Lembre-se que em outros momentos criamos as estruturas de código para isso e lá entendemos que como parte do processo de aprendizado é realizado essa verificação, onde de acordo com os padrões encontrados, os mais corretos e eficientes são guardados enquanto os incorretos aos poucos vão sendo descartados, de modo que a ideia é que nosso Agente domine uma determinada função quando o mesmo tiver uma bagagem de conhecimento acumulada as formas corretas de realizar tais feitos; DISCOUNT_FACTOR aqui definido como 0.99 nada mais é do que o gamma da equação e Bellman, equivalente a taxa de aprendizado de outros modelos de rede neural artificial convencional; REPLAY_MEMORY_START_SIZE com valor atribuído de 50000, aqui define o número de ações aleatórias iniciais que nosso Agente terá, lembre-se que em seu estado inicial o mesmo não tem nenhuma informação de aprendizado passado, começando totalmente de forma aleatória, em um processo de tentativa e erro; MAX_FRAMES definido como 30000000 delimita o tamanho de frames que nosso Agente será capaz de “ver” em seu estado inicial, tenha em mente que esse número alto inicial se dá porque aqui é levadas em consideração não somente as possibilidades de sua primeira ação, mas toda a árvore de possibilidades de todos os passos até chegar muito próximo ou alcançar seu objetivo, esse número é astronomicamente maior em sistemas de carro autônomo, por exemplo, onde as possibilidades são praticamente infinitas; MEMORY_SIZE, aqui com valor 1000000 define o número de informações que serão mantidas na memória de replay; NO_OP_STEPS = 10 delimita o número de ações iniciais que podem ser tomadas a cada ciclo de execução assim como de avaliação; UPDATE_FREQ aqui define que a cada 4 das 10 ações iniciais será realizado o reajuste dos pesos.

Lembre-se que estamos trabalhando sobre uma rede neural artificial intuitiva porém temos mecanismos equivalentes aos de outras redes, nesse contexto, esse parâmetro equivale aos ajustes dos pesos para atualização da descida do gradiente em modelos convencionais; HIDDEN aqui setado em 1024 simplesmente define o número de neurônios na primeira camada oculta da rede neural densa; LEARNING_RATE com valor 0.00001 define a taxa de aprendizado, que nesse contexto, está diretamente relacionado com UPDATE_FREQ; BS aqui simplesmente configura um valor de 32 para

batch; PATH especifica o caminho onde será salvo o gif; SUMMARIES especifica onde ficarão salvos os arquivos de sumário, que nesse caso registram todas as ações corretas tomadas por nosso Agente, desde o começo até o ponto onde alcança seu objetivo, em forma de todo o caminho de processamento realizado em um ciclo; RUNID está diretamente ligado com o parâmetro anterior, pois é possível, obviamente, salvar diferentes execuções para que se realizem comparações e avaliações das mesmas; Via makedirs( ), função da biblioteca os, são criados os respectivos diretórios.

```
282  atari = Atari(ENV_NAME, NO_OP_STEPS)
283  print(f'O ambiente gerado tem as respectivas {atari.env.action_space.n}')
284  print(f'de {atari.env.unwrapped.get_action_meanings()} ações.')
285
```

Por fim, o objeto atari, aqui instanciando a classe Atari, repassa para a mesma os dados de ENV_NAME e NO_OP_STEPS. Finalizando é exibida via console uma mensagem do estado inicial do Agente.

```
286  with tf.variable_scope('mainDQN'):
287      MAIN_DQN = DQN(atari.env.action_space.n, HIDDEN, LEARNING_RATE)
288  with tf.variable_scope('targetDQN'):
289      TARGET_DQN = DQN(atari.env.action_space.n, HIDDEN)
290  init = tf.global_variables_initializer()
291  saver = tf.train.Saver()
292  MAIN_DQN_VARS = tf.trainable_variables(scope='mainDQN')
293  TARGET_DQN_VARS = tf.trainable_variables(scope='targetDQN')
294
```

Anteriormente em determinados blocos de códigos de classes foram necessárias as criações de algumas sessões específicas, agora dando continuidade criaremos algumas no escopo global de nosso código. Porém antes disso precisamos criar algumas instancias no TensorFlow para que, literalmente, seus tensores saibam onde, em que ordem e quais dados buscar.

Sendo assim, via método variable_scope( ) parametrizado com ‘mainDQN’, chamamos da função homônima, nossa rede neural DQN, passando para ela os parâmetros para atari.env.action_space.n (o jogo em si), HIDDEN e LEARNING_RATE. Da mesma maneira para ‘targetDQN’ é repassado a instância do jogo e HIDDEN, repare que um escopo é para a rede base e outra para a rede que processará os dados alvo, os dados a serem atingidos com o cruzamento de todos aqueles objetos criados anteriormente

sob a ótica de sistema de dependência.

Logo após é criado o objeto init que chama a função global_variables_initializer( ), da biblioteca TensorFlow, sem parâmetros mesmo.

Também é criado o objeto saver, por sua vez chamando a função train.Saver( ), também sem nenhum parâmetro específico além dos parâmetros padrão.

Prosseguindo é instanciada a variável reservada da biblioteca gym MAIN_DQN_VARS que chama a função da biblioteca TensorFlow trainable_variables( ) alimentando a mesma com ‘mainDQN’. Note que essa função em particular equivale a função .fit( ) de alguns modelos convencionais. O mesmo é feito para TARGET_DQN_VARS com a respectiva ‘targetDQN’.

```
295   LAYER_IDS = ["conv1", "conv2", "conv3", "conv4", "denseAdvantage",
296                "denseAdvantageBias", "denseValue", "denseValueBias"]
297
```

Apenas em uma pequena codificação adicional, para LAYERS_IDS são repassados todos os identificadores de objetos onde existem camadas de frames processados, note que alguns deles definimos manualmente nas camadas da rede neural densa enquanto outros (Bias) antes ignorados agora são instanciados no processo.

```
298   def train():
299       my_replay_memory = ReplayMemory(size=MEMORY_SIZE, batch_size=BS)
300       update_networks = TargetNetworkUpdater(MAIN_DQN_VARS, TARGET_DQN_VARS)
301       explore_exploit_sched = ExplorationExploitationScheduler(
302           MAIN_DQN, atari.env.action_space.n,
303           replay_memory_start_size=REPLAY_MEMORY_START_SIZE,
304           max_frames=MAX_FRAMES)
305
```

Eis que finalmente chegamos no momento mais crítico de nosso código, o de criar a função treino de nosso Agente, que realizará a execução de todas as instancias, assim como o cruzamento de todos os seus dados nas respectivas sessões.

```
306     with tf.Session() as sess:
307         sess.run(init)
308         frame_number = 0
309         rewards = []
310         loss_list = []
311
```

Indentado para a função train( ) criamos uma estrutura de sessão via Session( ) referenciada como sess onde a sessão inicialmente é inicializada com os dados de init, passados como parâmetro para sess.run( ). Também é especificado o frame inicial a ser lido, neste caso, o frame 0 para o objeto frame_number. Também são instanciados os objetos rewards e loss_list com suas respectivas listas inicialmente vazias.

```
312         while frame_number < MAX_FRAMES:
313             epoch_frame = 0
314             while epoch_frame < EVAL_FREQUENCY:
315                 terminal_life_lost = atari.reset(sess)
316                 episode_reward_sum = 0
317                 for _ in range(MAX_EPISODE_LENGTH):
318                     action = explore_exploit_sched.get_action(sess, frame_number, atari.state)
319                     processed_new_frame, reward, terminal, terminal_life_lost, _ = atari.step(sess, action)
320                     frame_number += 1
321                     epoch_frame += 1
322                     episode_reward_sum += reward
323                     clipped_reward = clip_reward(reward)
324                     my_replay_memory.add_experience(action=action,
325                                                     frame=processed_new_frame[:, :, 0],
326                                                     reward=clipped_reward,
327                                                     terminal=terminal_life_lost)
328
```

Peço perdão pela proporção da imagem, a mesma acabou ficando para esse bloco muito reduzida, porém é necessário mostrar nesse ponto todos os espaços de endentação. Ao final desse capítulo o código inteiro estará disponível em outro formato onde será possível realizar uma melhor leitura deste bloco.

Dando sequência, indentado para a sessão criada anteriormente criamos uma estrutura condicional que define que enquanto frame_number tiver seu valor menor que o de MAX_FRAMES será realizado uma série de ações. Tenha em mente que este gatilho inicial simplesmente está definindo um limite de frames a serem processados nesse contexto. Logo, é criado um objeto epoch_frame que inicialmente tem seu valor nulo.

Em seguida é criada uma estrutura condicional dentro da condicional anterior que define que enquanto o valor de epoch_frame for menor do que o

de EVAL_FREQUENCY, uma série de ações serão tomadas. Aqui está um gatilho dos sistemas de dependência de nosso Agente.

Na estrutura dessa condicional é instanciado terminal_life_lost que por sua vez recebe como atributo a função reset( ) parametrizada com sess. Em outras palavras uma vez que o número de vidas chega a zero o jogo recomeça do zero. Note que em seguida é instanciado o objeto episode_reward_sum que nesse caso recebe como valor 0, pois pela lógica se nosso Agente perdeu o jogo, sua pontuação deve ser resetada também.

Na sequência é criado um laço de repetição que usa uma variável vazia para percorrer todos os dados em MAX_EPISODE_LENGTH, dentro dessa estrutura o objeto action agora recebe atribuído para si os dados de explore_exploit_sched.get_action( ) parametrizado com sess, frame_number e atari.state. Vamos entender o que está acontecendo aqui, a ação que nosso Agente toma é com base em um estado, no código isso se dá pela leitura dos dados da sessão atual, último frame processado e o estado atual de atari (que instancia a classe Atari( )).

Em seguida os objetos processed_new_frame, reward, terminal, terminal_life_lost e a variável vazia _ de nosso laço de repetição recebem como atributos os dados de atari.step( ) por sua vez lendo os dados de sess e de action. Como dito anteriormente, é de suma importância entender esses cruzamentos de dados assim como as dependências que estamos criando. Aqui, nesse momento estamos basicamente realizando a atualização dessas variáveis com os últimos dados da última etapa de processamento do estado anterior do Agente.

Também é definido que frame_number e epoch_frame recebem como atributo o valor atual deles acrescentados em 1. Da mesma forma episode_reward_sum recebe como atributo a soma do seu valor atual com o valor de reward, atualizando assim essas variáveis.

Seguindo a mesma lógica, clipped_reward recebe para si os valores de clip_reward( ) parametrizado com os valores de reward.

Em seguida é criada uma estrutura que atualiza também o estado de nosso Agente com base nos últimos cruzamentos de dados, raciocine que isso é feito porque nosso agente não somente terá seus estados atualizados quando for recompensado ou penalizado por uma ação, lembre-se que existe também

um terceiro estado onde a ação tomada surtiu um efeito nulo, e mesmo aqui não havendo qualquer alteração, recompensa ou penalização, esse estado tem que ser também atualizado. Para isso my_replay_memory chama a função add_experience( ) parrando alguns parâmetros como action=action, frame = o valor da matriz de processed_new_frame, reward = último valor atribuído para clipped_reward e terminal = o último valor lido para terminal_life_lost. Dessa forma é acionado os gatilhos de atualização de estado de nosso Agente mesmo quando o último estado é nulo.

```
329                 if frame_number % UPDATE_FREQ == 0 and frame_number > REPLAY_MEMORY_START_SIZE:
330                     loss = learn(sess, my_replay_memory, MAIN_DQN, TARGET_DQN,
331                                  BS, gamma = DISCOUNT_FACTOR)
332                     loss_list.append(loss)
333                 if frame_number % NETW_UPDATE_FREQ == 0 and frame_number > REPLAY_MEMORY_START_SIZE:
334                     update_networks(sess)
335 
336                 if terminal:
337                     terminal = False
338                     break
```

Continuando na mesma endentação é criada outra estrutura condicional onde se o resultado da divisão entre o valor de frame_number e UPDATE_FREQ for 0, ou seja, sem resto nesta divisão, e o valor de frame_number for maior que o de REPLAY_MEMORY_START_SIZE, o objeto loss chama a função learn( ) repassando como parâmetros sess, my_replay_memory, MAIN_QDN, TARGET_DQN, BS e gama com o valor de DISCOUNT_FACTOR atribuído para si.

Note que o que este código está fazendo é dando continuidade para a ação de nosso Agente com base em um último estado lido como nulo (ação sem recompensa ou penalidade / ação com resultado 0 visto anteriormente). Executado este bloco, loss_list por meio da função append( ) tem seus valores atualizados com os dados de loss.

Mais uma estrutura condicional é criada, agora verificando se a diferença da divisão dos valores de frame_number e NETW_UPDATE_FREQ foram igual a 0 e o valor de frame_number for maior que o de REPLAY_MEMORY_START_SIZE, a função update_networks é instanciada tendo sess como parâmetro.

Ainda nesse contexto, finalizando o mesmo é criada uma última estrutura condicional onde se terminal tiver seu estado definido como False, é interrompida a execução.

```
340                 rewards.append(episode_reward_sum)
341                 if len(rewards) % 10 == 0:
342                     if frame_number > REPLAY_MEMORY_START_SIZE:
343                         summ = sess.run(PERFORMANCE_SUMMARIES,
344                                         feed_dict={LOSS_PH:np.mean(loss_list),
345                                                    REWARD_PH:np.mean(rewards[-100:])})
346                         SUMM_WRITER.add_summary(summ, frame_number)
347                         loss_list = []
348                     summ_param = sess.run(PARAM_SUMMARIES)
349                     SUMM_WRITER.add_summary(summ_param, frame_number)
350                     print(len(rewards), frame_number, np.mean(rewards[-100:]))
351                     with open('rewards.dat', 'a') as reward_file:
352                         print(len(rewards), frame_number,
353                               np.mean(rewards[-100:]), file=reward_file)
```

Dando continuidade agora realizamos a codificação do sistema que atualiza as recompensas dadas para nosso Agente, inicialmente instanciando rewards que chama a função append( ) por meio dela repassando para si os valores de episode_reward_sum.

Em seguida é criada uma estrutura condicional que percorre o tamanho de rewards e se o resto da divisão deste valor por 10 for igual a zero, é criada uma estrutura condicional dentro dessa estrutura atual.

Nessa nova estrutura condicional é realizada a verificação onde se o valor de frame_number for maior que o valor de REPLAY_MEMORY_START_SIZE, o objeto summ cria e roda uma sessão onde é instanciado PERFORMANCE_SUMMARIES, também é realizada a atualização de nosso dicionário por meio de feed_dict que por sua vez recebe para suas chaves os dados de LOSS_PH e para seus valores o valor médio de loss_list por meio da função np.mean( ).

Também é criada outra camada em nosso dicionário com dados chave de REWARD_PH e com dados de valor a média dos últimos 100 valores de rewards. Lembre-se que alguns blocos atrás criamos essas estruturas, porém elas tinham apenas um tamanho delimitado, mas sem dados as preenchendo, uma vez que não haviam dados de estados anteriores inicialmente, e nessa construção esses valores devem não só alimentar esses espaços como constantemente os atualizar.

Para o objeto da biblioteca gym SUM_WRITER é executada a função add_summary( ) repassando para mesma os dados de summ e frame_number.

Na mesma lógica loss_list recebe como valor inicial uma lista vazia a ser

alimentada com os respectivos valores de loss a partir dos primeiros ciclos de execução de nosso Agente.

Na sequência summ_param recebe como atributo os dados da sessão que instancia e executa PARAM_SUMMARIES.

Complementar a isso SUMM_WRITER roda a função add_summary( ) com os dados de summ_param e frame_number.

Repare que todo esse processo é equivalente a quando em uma rede neural convencional guardamos os melhores dados dos pesos para reutilização, aqui, nesse contexto de uma rede neural artificial intuitiva, tais valores são salvos de maneira parecida, com a particularidade de que os mesmos são constantemente atualizados, salvos, lidos, em um ciclo praticamente interminável. O fato de ter tais dados salvos em algum estado faz com que possamos “desligar” nosso agente assim como “religar” o mesmo de forma que ele retome seu último estado sem necessidade de recomeçar do zero.

Em seguida é criada uma estrutura para mostrar a leitura destes números e exibir em console durante a execução.

```
354             terminal = True
355             gif = True
356             frames_for_gif = []
357             eval_rewards = []
358             evaluate_frame_number = 0
359
```

Para isso também é realizada uma pequena codificação adicional onde habilitamos a exibição do terminal e do gif que nos mostrará nosso agente “jogando” BreakOut. Para essa visualização também deve ser instanciado frames_for_gif e eval_rewards como listas vazias.

Por fim, como aqui estamos apenas visualizando a execução de nosso Agente, é necessário definir que evaluate_frame_number terá atribuído para si o valor 0, em outras palavras isso sincroniza o valor de console com o frame que você está vendo.

```
360            for _ in range(EVAL_STEPS):
361                if terminal:
362                    terminal_life_lost = atari.reset(sess, evaluation=True)
363                    episode_reward_sum = 0
364                    terminal = False
365                action = 1 if terminal_life_lost else explore_exploit_sched.get_action(sess, frame_number,
366                                                                                       atari.state,
367                                                                                       evaluation=True)
368                processed_new_frame, reward, terminal, terminal_life_lost, new_frame = atari.step(sess, action)
369                evaluate_frame_number += 1
370                episode_reward_sum += reward
371                if gif:
372                    frames_for_gif.append(new_frame)
373                if terminal:
374                    eval_rewards.append(episode_reward_sum)
375                    gif = False
376            print("Pontuação:\n", np.mean(eval_rewards))
```

Para alguns exemplos, de algumas versões da biblioteca gym é necessário criar uma codificação adicional para que de fato a visualização de nosso Agente jogando BreakOut se dê de forma correta, caso contrário pode ocorrer de o gif ficar com a imagem apenas do primeiro frame, erro que como mencionado anteriormente ocorre para alguns exemplos.

Sendo assim, é necessário criar um laço de repetição que usa uma variável vazia _ para percorrer os dados de EVAL_STEPS, onde para cada leitura é realizada uma verificação onde se estamos trabalhando fazendo o uso do terminal, em terminal o objeto terminal_life_lost recebe como atributo atari.reset( ) tendo agora sess e evaluation = True, o que em outras palavras nos diz que o que for apresentado em terminal começa realmente do estado zero de nosso Agente e do jogo em si, contornando um bug que faz com que o jogo fosse iniciado de forma aleatória, sem seus valores resetados.

Em seguida episode_reward_sum = 0 apenas condiz com a explicação do parágrafo anterior. Também é alterado o estado de terminal para False quando essa condição for alcançada. Note que aqui estamos deixando bem específico que estamos trabalhando aplicando essas condições apenas para os dados visualizados, todo o resto que é executado em segundo plano segue normalmente.

Agora muita atenção para esta etapa, repare que aqui o que estamos instanciando para action é um valor 1 seguido de uma estrutura condicional, onde este valor deve coincidir com o valor de terminal_life_lost, caso contrário por meio de explore_exploit_sched.get_action( ) é acionado o gatilho para que o mesmo tome alguma decisão. Para a função get_action( ) são repassados como parâmetros frame_number, atari.state e evaluation = True.

Em seguida os dados de processed_new_frame, reward, terminal, terminal_life_lost e new_frame são atualizados com os dados obtidos de atari.step( ) parametrizado com action.

Note que o que estamos fazendo aqui é contornando o fato de que nosso Agente precisa esperar a reação do ambiente para suas ações, cada vez que o mesmo dispara seu tiro em direção aos blocos a serem destruídos existe um tempo ocioso do mesmo esperando o projétil chegar até os blocos e retornar, onde em boa parte desse trajeto o mesmo pode simplesmente ficar parado, apenas realizando a leitura do ambiente. A partir de um certo ponto que o projétil está voltando aí sim ele precisa ir em direção ao projétil. Você verá que nas primeiras execuções ele fica se movimentando de forma aleatória durante esse período, à medida que ele aprende ele começa a "poupar energia" realizando ação apenas quando necessário.

Da mesma forma como as outras atualizações realizadas no passado para estas variáveis, evaluate_frame_number tem seu valor atualizado com a soma de seu valor atual somado de 1. Da mesma forma episode_reward_sum tem seu valor atualizado com seu valor somado ao valor de reward.

Logo após é crida mais uma estrutura condicional, dessa vez verificando nosso gif, para o mesmo não ficar estagnado exibindo apenas o primeiro frame, frames_for_gif é atualizado também acrescentando para si o valor de new_frame.

Do mesmo jeito em nosso terminal eval_rewards receber um novo valor oriundo de episode_reward_sum, gif tem seu estado alterado para False. Lembre-se que já que estamos realizando essa visualização, queremos acompanhar apenas em terminal o que é exibido em tela por nosso gif.

Finalizando, é criada uma simples mensagem exibindo também a pontuação alcançada no jogo.

```
377                 try:
378                     generate_gif(frame_number, frames_for_gif, eval_rewards[0], PATH)
379                 except IndexError:
380                     print("ERRO: Sem pontuação para esse jogo")
381                 saver.save(sess, PATH+'/my_model', global_step=frame_number)
382                 frames_for_gif = []
383                 summ = sess.run(EVAL_SCORE_SUMMARY, feed_dict={EVAL_SCORE_PH:np.mean(eval_rewards)})
384                 SUMM_WRITER.add_summary(summ, frame_number)
385                 with open('rewardsEval.dat', 'a') as eval_reward_file:
386                     print(frame_number, np.mean(eval_rewards), file=eval_reward_file)
387
```

Lembre-se que uma prática comum quando estamos criando qualquer software é criar sistemas de validação, onde iremos tentar prever todos erros ou exceções cometidas pelo usuário do programa, contornando-as evitando que o programa pare de funcional. Aqui nosso Agente é programado por um humano, logo, cometerá erros que não foram previstos no momento da codificação de suas estruturas de código.

Sendo assim, é interessante criar um sistema de validação também para nossa rede neural artificial intuitiva, como se bem sabe, isso em Python é feito por meio de try e except.

Para a tentativa de contornar a não exibição correta de nosso gif chamamos manualmente a função generate_gif( ) passando como parâmetro para a mesmo frame_number, frames_for_gif, eval_rewards em sua posição 0 e PATH. Isso irá forçar a criação e leitura do gif.

Caso não seja possível realizar a ação anterior, muito provavelmente o que está ocorrendo é que no cruzamento dos dados nenhum valor está sendo repassado ou atualizado para eval_rewards. Então é exibida uma mensagem em console elucidando esse erro. Na sequência serão tomadas uma série de medidas para forçar que as devidas ações sejam feitas.

Como estamos trabalhando com objetos reservados da biblioteca gym cruzando dados com nossos objetos criados para esse exemplo, não que seja comum, mas é perfeitamente possível haver alguma incompatibilidade entre o cruzamento desses dados. Agora iremos forçar o salvamento dos dados dos primeiros ciclos processados.

Para isso instanciando o objeto saver, executando a função save( ) parrando como parâmetro PATH concatenado com ‘/my_model’, também parametrizando manualmente global_step com frame_number.

Em seguida frames_for_gif é instanciado com uma nova lista vazia. Nos mesmos moldes summ novamente abre e executa uma sessão parametrizada com os dados de EVAL_SCORE_SUMMARY e o mesmo dicionário usado sempre, com os dados em chaves:valores de EVAL_SCORE_PH e o valor médio de eval_rewards.

Também é executado novamente todo o processo de SUMM_WRITER, atualizando o sumário com os dados de summ e frame_number.

Por fim, encerrando esse bloco validador, é aberto o arquivo criado pelo validador, ‘rewardsEval.dat’ como atributo de eval_reward_file, também é exibido em tela os dados de frame_number, média de eval_rewards e os dados lidos a partir de eval_reward_file, caso, obviamente, todo processo anterior tenha sido executado com sucesso.

```
388  TRAIN = train()
389  if TRAIN:
390      train()
391  if not TRAIN:
392      gif_path = "GIF/"
393      os.makedirs(gif_path, exist_ok=True)
394      trained_path, save_file = save_files_dict[ENV_NAME]
395      explore_exploit_sched = ExplorationExploitationScheduler(
396          MAIN_DQN, atari.env.action_space.n,
397          replay_memory_start_size=REPLAY_MEMORY_START_SIZE,
398          max_frames=MAX_FRAMES)
399
```

Atingindo a marca de 400 linhas de código, mas partindo para o fim do modelo de nossa rede artificial intuitiva, podemos criar o gatilho final que é por meio de TRAIN, finalmente instanciar e executar a função train( ). Caso não haja nenhum erro de sintaxe, a partir desse momento nosso agente é ativado e começa a trabalhar de forma contínua. Porém, vamos criar uma última estrutura que funcionará como validadora, serão apenas mais algumas linhas de código para quem já escreveu 400, que irão forçar todos os gatilhos serem acionados, da maneira correta, para que a inteligência artificial de nosso Agente funciona corretamente.

Para isso criamos uma estrutura condicional que verifica se train( ) está mesmo sendo executada por TRAIN, supondo que você está executando o código pela primeira vez, não havendo estados anteriores, não havendo nenhum dado ou valor de objeto nenhum alocado em memória, etc... Realmente aqui estamos criando o validador que é o gatilho inicial para que tudo comece a entrar em processamento.

Inicialmente gif_path recebe como atributo um diretório a ser criado caso ainda não exista um específico para nosso gif. Pela função do sistema makedirs( ) é validado o caminho de gif_path para que seja reconhecido.

Em seguida para trained_path e save_file são repassados os dados de

save_files_dict, neste caso, todas as instâncias de ‘BreakoutDeterministic-v3’.

Na sequência explore_exploit_sched recebe como atributo a classe ExplorationExploitationScheduler( ) que por sua vez instancia a rede neural MAIN_DQN, atari.env.action_space.n, que pré-carrega o jogo BreakOut a partir da biblioteca gym e define manualmente para esse estado atual que replay_memory_start_size obtém dados de REPLAY_MEMORY_START_SIZE, assim como max_frames obtém dados de MAX_FRAMES.

```
400     with tf.Session() as sess:
401         saver = tf.train.import_meta_graph(trained_path+save_file)
402         saver.restore(sess,tf.train.latest_checkpoint(trained_path))
403         frames_for_gif = []
404         terminal_life_lost = atari.reset(sess, evaluation = True)
405         episode_reward_sum = 0
406         while True:
407             atari.env.render()
408             action = 1 if terminal_life_lost else explore_exploit_sched.get_action(sess, 0, atari.state,
409                                                                                    evaluation = True)
410             processed_new_frame, reward, terminal, terminal_life_lost, new_frame = atari.step(sess, action)
411             episode_reward_sum += reward
412             frames_for_gif.append(new_frame)
413             if terminal == True:
414                 break
415
```

Seguindo com a parte final do código, note que agora abrimos uma sessão que englobará todas as outras em seu escopo, pois instanciando Session( ) repassando cada sessão de cada classe, por meio de sess, estamos agora de certa forma importando tudo para essa sessão em particular.

Dentro de sua estrutura saver recebe por meio da função import_meta_graph( ) o valor da soma dos dados de trained_path e save_file. Do mesmo modo via função restore( ) saver realiza a execução da função latest_checkpoint( ) parametrizada por sua vez com trained_path. Consegue perceber que agora também existe o cruzamento e verificação dos dados de diferentes escopos, finalmente temos uma cadeia de retroalimentação ativa para nossa inteligência artificial.

Novamente frames_for_gif tem seus dados atualizados com uma lista vazia, pois terminal_life_lost valida o estado inicial que o jogo deve ter a esse momento, o que pode ser confirmado no objeto seguinte pois episode_reward_sum volta a ter valor inicial 0.

Na sequência é criada mais uma estrutura condicional, onde, enquanto o

estado dessa sessão for True, é feito o processamento contínuo por meio de atari.env.render( ) de nosso Agente sobre o jogo carregado.

Também é reforçado que é necessária a realização da primeira ação para que sejam acionados todos gatilhos iniciais em seu efeito cascata. Isso é feito por meio de action, como feito anteriormente, executando uma ação ou pronto para execução de uma ação a qualquer momento, mesmo esperando o tempo de resposta do ambiente.

Mais uma vez é realizadas as atualizações de processed_new_frame, reward, terminal, terminal_life_lost, new_frame, episode_reward_sum e frames_for_gif como realizado anteriormente.

Por fim, é feita a verificação de que quando o conteúdo de terminal chega ao fim de processamento, essa sessão pode ficar suspensa.

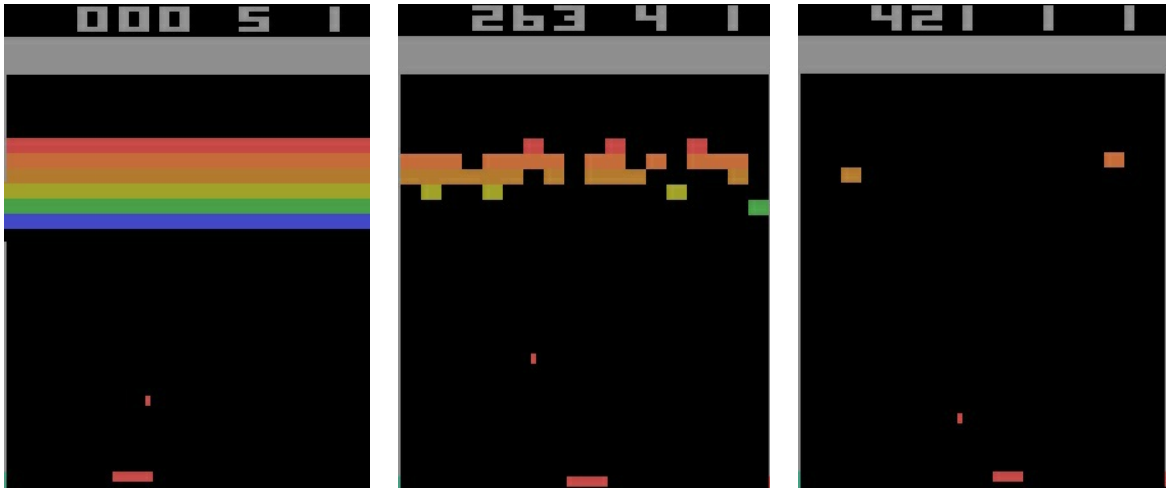
```
416         atari.env.close()
417         print("A recompensa total é: {}".format(episode_reward_sum))
418         print("Gerando GIF...")
419         generate_gif(0, frames_for_gif, episode_reward_sum, gif_path)
420         print("Gif criado, confira na pasta {}".format(gif_path))
421
```

Encerrando nosso exemplo, uma vez que nosso agente dominou o jogo por completo, terminando-o, o mesmo pode agora encerrar esse processo e partir para uma nova ação diferente de jogar BreakOut. Isso é feito encerrando o ambiente via atari.env.close( ).

Como de costume, podemos exibir em tela algumas mensagens nos dando o feedback de nosso Agente. Por fim é salvo no referente pasta um gif que mostra diferentes estados do progresso de nosso Agente.



Observando por alguns momentos é possível, como esperado, notar que nas primeiras execuções nosso Agente erra bastante seus movimentos, perdendo no jogo, porém à medida que o mesmo realiza os primeiros acertos é impressionante a velocidade como o mesmo evolui no jogo, não cometendo mais erros e dominando o mesmo.



À esquerda o estado inicial do jogo, ao centro mais ou menos metade dos blocos já destruídos e à direita uma imagem do final do jogo.

Apenas lembrando que o tempo de processamento está diretamente ligado com o seu hardware em questão, nos testes feitos para esse livro, o código acima foi executado em um Intel i7 3770k de 3.5Ghz, com 16gb de ram e uma placa de vídeo GeForce GTX 1060 de 6gb, tendo um tempo médio de 50 segundos para que fosse completada toda a tarefa.

Código Completo:

```
import os
import random
import gym
import tensorflow as tf
import numpy as np
import imageio
from skimage.transform import resize

ENV_NAME = 'BreakoutDeterministic-v3'

class FrameProcessor(object):
    def __init__(self, frame_height=84, frame_width=84):
        self.frame_height = frame_height
        self.frame_width = frame_width
        self.frame = tf.placeholder(shape=[210, 160, 3], dtype=tf.uint8)
        self.processed = tf.image.rgb_to_grayscale(self.frame)
        self.processed = tf.image.crop_to_bounding_box(self.processed, 34, 0, 160, 160)
        self.processed = tf.image.resize_images(self.processed,
                                                [self.frame_height, self.frame_width],
                                                method=tf.image.ResizeMethod.NEAREST_NEIGHBOR)

    def __call__(self, session, frame):
        return session.run(self.processed, feed_dict={self.frame:frame})
```

```
class DQN(object):
    def __init__(self, n_actions, hidden=1024, learning_rate=0.00001,
                 frame_height=84, frame_width=84, agent_history_length=4):
        self.n_actions = n_actions
        self.hidden = hidden
        self.learning_rate = learning_rate
        self.frame_height = frame_height
        self.frame_width = frame_width
        self.agent_history_length = agent_history_length
        self.input = tf.placeholder(shape=[None, self.frame_height,
                                           self.frame_width,
                                           self.agent_history_length],
                                           dtype=tf.float32)
        self.inputscaled = self.input/255
        self.conv1 = tf.layers.conv2d(inputs=self.inputscaled,
            filters=32, kernel_size=[8, 8], strides=4,
            kernel_initializer=tf.variance_scaling_initializer(scale=2),
            padding="valid", activation=tf.nn.relu, use_bias=False, name='conv1')
        self.conv2 = tf.layers.conv2d(
            inputs=self.conv1, filters=64, kernel_size=[4, 4], strides=2,
            kernel_initializer=tf.variance_scaling_initializer(scale=2),
            padding="valid", activation=tf.nn.relu, use_bias=False, name='conv2')
        self.conv3 = tf.layers.conv2d(
            inputs=self.conv2, filters=64, kernel_size=[3, 3], strides=1,
            kernel_initializer=tf.variance_scaling_initializer(scale=2),
            padding="valid", activation=tf.nn.relu, use_bias=False, name='conv3')
        self.conv4 = tf.layers.conv2d(
            inputs=self.conv3, filters=hidden, kernel_size=[7, 7], strides=1,
            kernel_initializer=tf.variance_scaling_initializer(scale=2),
            padding="valid", activation=tf.nn.relu, use_bias=False, name='conv4')
```

```
        self.valuestream, self.advantagestream = tf.split(self.conv4, 2, 3)
        self.valuestream = tf.layers.flatten(self.valuestream)
        self.advantagestream = tf.layers.flatten(self.advantagestream)
        self.advantage = tf.layers.dense(
            inputs=self.advantagestream, units=self.n_actions,
            kernel_initializer=tf.variance_scaling_initializer(scale=2),
            name="advantage")
        self.value = tf.layers.dense(
            inputs=self.valuestream, units=1,
            kernel_initializer=tf.variance_scaling_initializer(scale=2),
            name='value')
        self.q_values = self.value + tf.subtract(self.advantage,tf.reduce_mean(self.advantage,
                                                 axis=1,
                                                 keepdims=True))
        self.best_action = tf.argmax(self.q_values, 1)
        self.target_q = tf.placeholder(shape=[None], dtype=tf.float32)
        self.action = tf.placeholder(shape=[None], dtype=tf.int32)
        self.Q = tf.reduce_sum(tf.multiply(self.q_values, tf.one_hot(self.action,
                                           self.n_actions,
                                           dtype=tf.float32)),
                                           axis=1)
        self.loss = tf.reduce_mean(tf.losses.huber_loss(labels=self.target_q,
                                   predictions=self.Q))
        self.optimizer = tf.train.AdamOptimizer(learning_rate=self.learning_rate)
        self.update = self.optimizer.minimize(self.loss)

class ExplorationExploitationScheduler(object):
    def __init__(self, DQN, n_actions, eps_initial=1, eps_final=0.1, eps_final_frame=0.01,
                 eps_evaluation=0.0, eps_annealing_frames=1000000,
                 replay_memory_start_size=50000, max_frames=25000000):
        self.n_actions = n_actions
```

```
        self.eps_initial = eps_initial
        self.eps_final = eps_final
        self.eps_final_frame = eps_final_frame
        self.eps_evaluation = eps_evaluation
        self.eps_annealing_frames = eps_annealing_frames
        self.replay_memory_start_size = replay_memory_start_size
        self.max_frames = max_frames
        self.slope = -(self.eps_initial - self.eps_final)/self.eps_annealing_frames
        self.intercept = self.eps_initial - self.slope*self.replay_memory_start_size
        self.slope_2 = -(self.eps_final - self.eps_final_frame)/
                       (self.max_frames - self.eps_annealing_frames - self.replay_memory_start_size)
        self.intercept_2 = self.eps_final_frame - self.slope_2*self.max_frames
        self.DQN = DQN

    def get_action(self, session, frame_number, state, evaluation=False):
        if evaluation:
            eps = self.eps_evaluation
        elif frame_number < self.replay_memory_start_size:
            eps = self.eps_initial
        elif frame_number >= self.replay_memory_start_size and frame_number <
                             self.replay_memory_start_size + self.eps_annealing_frames:
            eps = self.slope*frame_number + self.intercept
        elif frame_number >= self.replay_memory_start_size + self.eps_annealing_frames:
            eps = self.slope_2*frame_number + self.intercept_2
        if np.random.rand(1) < eps:
            return np.random.randint(0, self.n_actions)
        return session.run(self.DQN.best_action, feed_dict={self.DQN.input:[state]})[0]

class ReplayMemory(object):
    def __init__(self, size=1000000, frame_height=84, frame_width=84,
                 agent_history_length=4, batch_size=32):
```

```
        self.size = size
        self.frame_height = frame_height
        self.frame_width = frame_width
        self.agent_history_length = agent_history_length
        self.batch_size = batch_size
        self.count = 0
        self.current = 0
        self.actions = np.empty(self.size, dtype=np.int32)
        self.rewards = np.empty(self.size, dtype=np.float32)
        self.frames = np.empty((self.size, self.frame_height,
                                self.frame_width), dtype=np.uint8)
        self.terminal_flags = np.empty(self.size, dtype=np.bool)
        self.states = np.empty((self.batch_size, self.agent_history_length,
                                self.frame_height, self.frame_width),
                               dtype=np.uint8)
        self.new_states = np.empty((self.batch_size, self.agent_history_length,
                                    self.frame_height, self.frame_width),
                                   dtype=np.uint8)
        self.indices = np.empty(self.batch_size, dtype=np.int32)

    def add_experience(self, action, frame, reward, terminal):
        if frame.shape != (self.frame_height, self.frame_width):
            raise ValueError('Dimensão do frame está incorreta!')
        self.actions[self.current] = action
        self.frames[self.current, ...] = frame
        self.rewards[self.current] = reward
        self.terminal_flags[self.current] = terminal
        self.count = max(self.count, self.current+1)
        self.current = (self.current + 1) % self.size

    def _get_state(self, index):
```

```
        if self.count is 0:
            raise ValueError("A memória de replay está vazia!")
        if index < self.agent_history_length - 1:
            raise ValueError("Indice mínimo deve ser 3")
        return self.frames[index-self.agent_history_length+1:index+1, ...]

    def _get_valid_indices(self):
        for i in range(self.batch_size):
            while True:
                index = random.randint(self.agent_history_length, self.count - 1)
                if index < self.agent_history_length:
                    continue
                if index >= self.current and index - self.agent_history_length <= self.current:
                    continue
                if self.terminal_flags[index - self.agent_history_length:index].any():
                    continue
                break
            self.indices[i] = index

    def get_minibatch(self):
        if self.count < self.agent_history_length:
            raise ValueError('ERRO: Não foi possível alocar espaço na memória.')

        self._get_valid_indices()
        for i, idx in enumerate(self.indices):
            self.states[i] = self._get_state(idx - 1)
            self.new_states[i] = self._get_state(idx)
        return np.transpose(self.states, axes=(0, 2, 3, 1)),
                            self.actions[self.indices],
                            self.rewards[self.indices],
                            np.transpose(self.new_states,
```

```
                    axes=(0, 2, 3, 1)),
                    self.terminal_flags[self.indices]

def learn(session, replay_memory, main_dqn, target_dqn, batch_size, gamma):
    states, actions, rewards, new_states, terminal_flags = replay_memory.get_minibatch()
    arg_q_max = session.run(main_dqn.best_action, feed_dict={main_dqn.input:new_states})
    q_vals = session.run(target_dqn.q_values, feed_dict={target_dqn.input:new_states})
    double_q = q_vals[range(batch_size), arg_q_max]
    target_q = rewards + (gamma*double_q * (1-terminal_flags))
    loss, _ = session.run([main_dqn.loss, main_dqn.update],
                          feed_dict={main_dqn.input:states,
                                     main_dqn.target_q:target_q,
                                     main_dqn.action:actions})
    return loss

class TargetNetworkUpdater(object):
    def __init__(self, main_dqn_vars, target_dqn_vars):
        self.main_dqn_vars = main_dqn_vars
        self.target_dqn_vars = target_dqn_vars

    def _update_target_vars(self):
        update_ops = []
        for i, var in enumerate(self.main_dqn_vars):
            copy_op = self.target_dqn_vars[i].assign(var.value())
            update_ops.append(copy_op)
        return update_ops

    def __call__(self, sess):
        update_ops = self._update_target_vars()
        for copy_op in update_ops:
            sess.run(copy_op)
```

```
def generate_gif(frame_number, frames_for_gif, reward, path):
    for idx, frame_idx in enumerate(frames_for_gif):
        frames_for_gif[idx] = resize(frame_idx, (420, 320, 3),
                                     preserve_range=True, order=0).astype(np.uint8)

    imageio.mimsave(f'{path}{"ATARI_frame_{0}_reward_{1}.gif".format(frame_number, reward)}',
                    frames_for_gif, duration=1/30)

class Atari(object):
    def __init__(self, envName, no_op_steps=10, agent_history_length=4):
        self.env = gym.make(envName)
        self.process_frame = FrameProcessor()
        self.state = None
        self.last_lives = 0
        self.no_op_steps = no_op_steps
        self.agent_history_length = agent_history_length

    def reset(self, sess, evaluation=False):
        frame = self.env.reset()
        self.last_lives = 0
        terminal_life_lost = True
        if evaluation:
            for _ in range(random.randint(1, self.no_op_steps)):
                frame, _, _, _ = self.env.step(1)
        processed_frame = self.process_frame(sess, frame)
        self.state = np.repeat(processed_frame, self.agent_history_length, axis=2)
        return terminal_life_lost

    def step(self, sess, action):
        new_frame, reward, terminal, info = self.env.step(action)
```

```
        if info['ale.lives'] < self.last_lives:
            terminal_life_lost = True
        else:
            terminal_life_lost = terminal
        self.last_lives = info['ale.lives']
        processed_new_frame = self.process_frame(sess, new_frame)
        new_state = np.append(self.state[:, :, 1:], processed_new_frame, axis=2)
        self.state = new_state
        return processed_new_frame, reward, terminal, terminal_life_lost, new_frame

def clip_reward(reward):
    if reward > 0:
        return 1
    elif reward == 0:
        return 0
    else:
        return -1

tf.reset_default_graph()
MAX_EPISODE_LENGTH = 18000
EVAL_FREQUENCY = 200000
EVAL_STEPS = 10000
NETW_UPDATE_FREQ = 10000
DISCOUNT_FACTOR = 0.99
REPLAY_MEMORY_START_SIZE = 50000
MAX_FRAMES = 30000000
MEMORY_SIZE = 1000000
NO_OP_STEPS = 10
UPDATE_FREQ = 4
HIDDEN = 1024
LEARNING_RATE = 0.00001
```

```
BS = 32
PATH = "output/"
SUMMARIES = "summaries"
RUNID = 'run_1'
os.makedirs(PATH, exist_ok=True)
os.makedirs(os.path.join(SUMMARIES, RUNID), exist_ok=True)
SUMM_WRITER = tf.summary.FileWriter(os.path.join(SUMMARIES, RUNID))
atari = Atari(ENV_NAME, NO_OP_STEPS)
print(f'O ambiente gerado tem as respectivas {atari.env.action_space.n} ações: {atari.env.unwrapped.ge

with tf.variable_scope('mainDQN'):
    MAIN_DQN = DQN(atari.env.action_space.n, HIDDEN, LEARNING_RATE)
with tf.variable_scope('targetDQN'):
    TARGET_DQN = DQN(atari.env.action_space.n, HIDDEN)
init = tf.global_variables_initializer()
saver = tf.train.Saver()
MAIN_DQN_VARS = tf.trainable_variables(scope='mainDQN')
TARGET_DQN_VARS = tf.trainable_variables(scope='targetDQN')

LAYER_IDS = ["conv1", "conv2", "conv3", "conv4", "denseAdvantage",
             "denseAdvantageBias", "denseValue", "denseValueBias"]
with tf.name_scope('Performance'):
    LOSS_PH = tf.placeholder(tf.float32, shape=None, name='loss_summary')
    LOSS_SUMMARY = tf.summary.scalar('loss', LOSS_PH)
    REWARD_PH = tf.placeholder(tf.float32, shape=None, name='reward_summary')
    REWARD_SUMMARY = tf.summary.scalar('reward', REWARD_PH)
    EVAL_SCORE_PH = tf.placeholder(tf.float32, shape=None, name='evaluation_summary')
    EVAL_SCORE_SUMMARY = tf.summary.scalar('evaluation_score', EVAL_SCORE_PH)

PERFORMANCE_SUMMARIES = tf.summary.merge([LOSS_SUMMARY, REWARD_SUMMARY
with tf.name_scope('Parameters'):
```

```
    ALL_PARAM_SUMMARIES = []
    for i, Id in enumerate(LAYER_IDS):
        with tf.name_scope('mainDQN/'):
            MAIN_DQN_KERNEL = tf.summary.histogram(Id, tf.reshape(MAIN_DQN_VARS[i], shape=[-1]))
        ALL_PARAM_SUMMARIES.extend([MAIN_DQN_KERNEL])
PARAM_SUMMARIES = tf.summary.merge(ALL_PARAM_SUMMARIES)

def train():
    my_replay_memory = ReplayMemory(size=MEMORY_SIZE, batch_size=BS)
    update_networks = TargetNetworkUpdater(MAIN_DQN_VARS, TARGET_DQN_VARS)
    explore_exploit_sched = ExplorationExploitationScheduler(
        MAIN_DQN, atari.env.action_space.n,
        replay_memory_start_size=REPLAY_MEMORY_START_SIZE,
        max_frames=MAX_FRAMES)

    with tf.Session() as sess:
        sess.run(init)
        frame_number = 0
        rewards = []
        loss_list = []

        while frame_number < MAX_FRAMES:
            epoch_frame = 0
            while epoch_frame < EVAL_FREQUENCY:
                terminal_life_lost = atari.reset(sess)
                episode_reward_sum = 0
                for _ in range(MAX_EPISODE_LENGTH):
                    action = explore_exploit_sched.get_action(sess, frame_number, atari.state)
                    processed_new_frame, reward, terminal, terminal_life_lost, _ = atari.step(sess, action)
                    frame_number += 1
```

```
            epoch_frame += 1
            episode_reward_sum += reward
            clipped_reward = clip_reward(reward)
            my_replay_memory.add_experience(action=action,
                                            frame=processed_new_frame[:, :, 0],
                                            reward=clipped_reward,
                                            terminal=terminal_life_lost)

            if frame_number % UPDATE_FREQ == 0 and frame_number > REPLAY_MEMORY_ST
              loss = learn(sess, my_replay_memory, MAIN_DQN, TARGET_DQN,
                           BS, gamma = DISCOUNT_FACTOR)
              loss_list.append(loss)
            if frame_number % NETW_UPDATE_FREQ == 0 and frame_number > REPLAY_MEMC
              update_networks(sess)

            if terminal:
              terminal = False
              break

          rewards.append(episode_reward_sum)
          if len(rewards) % 10 == 0:
            if frame_number > REPLAY_MEMORY_START_SIZE:
              summ = sess.run(PERFORMANCE_SUMMARIES,
                              feed_dict={LOSS_PH:np.mean(loss_list),
                                         REWARD_PH:np.mean(rewards[-100:])})
              SUMM_WRITER.add_summary(summ, frame_number)
              loss_list = []
            summ_param = sess.run(PARAM_SUMMARIES)
            SUMM_WRITER.add_summary(summ_param, frame_number)
            print(len(rewards), frame_number, np.mean(rewards[-100:]))
            with open('rewards.dat', 'a') as reward_file:
```

```
                print(len(rewards), frame_number,
                      np.mean(rewards[-100:]), file=reward_file)
        terminal = True
        gif = True
        frames_for_gif = []
        eval_rewards = []
        evaluate_frame_number = 0

        for _ in range(EVAL_STEPS):
            if terminal:
                terminal_life_lost = atari.reset(sess, evaluation=True)
                episode_reward_sum = 0
                terminal = False
            action = 1 if terminal_life_lost else explore_exploit_sched.get_action(sess, frame_number,
                                                    atari.state,
                                                    evaluation=True)
            processed_new_frame, reward, terminal, terminal_life_lost, new_frame = atari.step(sess, actio
            evaluate_frame_number += 1
            episode_reward_sum += reward
            if gif:
                frames_for_gif.append(new_frame)
            if terminal:
                eval_rewards.append(episode_reward_sum)
                gif = False
        print("Pontuação:\n", np.mean(eval_rewards))
        try:
            generate_gif(frame_number, frames_for_gif, eval_rewards[0], PATH)
        except IndexError:
            print("ERRO: Sem pontuação para esse jogo")
        saver.save(sess, PATH+'/my_model', global_step=frame_number)
        frames_for_gif = []
```

```
                summ = sess.run(EVAL_SCORE_SUMMARY, feed_dict={EVAL_SCORE_PH:np.mean(eval_rewards)})
            SUMM_WRITER.add_summary(summ, frame_number)
            with open('rewardsEval.dat', 'a') as eval_reward_file:
                print(frame_number, np.mean(eval_rewards), file=eval_reward_file)

TRAIN = train()
if TRAIN:
    train()
if not TRAIN:
    gif_path = "GIF/"
    os.makedirs(gif_path, exist_ok=True)
    trained_path, save_file = save_files_dict[ENV_NAME]
    explore_exploit_sched = ExplorationExploitationScheduler(
        MAIN_DQN, atari.env.action_space.n,
        replay_memory_start_size=REPLAY_MEMORY_START_SIZE,
        max_frames=MAX_FRAMES)

    with tf.Session() as sess:
        saver = tf.train.import_meta_graph(trained_path+save_file)
        saver.restore(sess,tf.train.latest_checkpoint(trained_path))
        frames_for_gif = []
        terminal_life_lost = atari.reset(sess, evaluation = True)
        episode_reward_sum = 0
        while True:
            atari.env.render()
            action = 1 if terminal_life_lost else explore_exploit_sched.get_action(sess, 0, atari.state,
                                                    evaluation = True)
            processed_new_frame, reward, terminal, terminal_life_lost, new_frame = atari.step(sess, action)
            episode_reward_sum += reward
            frames_for_gif.append(new_frame)
```

```
        if terminal == True:
            break

    atari.env.close()
    print("A recompensa total é: {}".format(episode_reward_sum))
    print("Gerando GIF...")
    generate_gif(0, frames_for_gif, episode_reward_sum, gif_path)
    print("Gif criado, confira na pasta {}".format(gif_path))
```

PARÂMETROS ADICIONAIS

Ao longo de nossos exemplos abordados nesse livro, foram passados os modelos mais comumente usados, assim como sua estrutura e parametrização, porém, este é apenas o passo inicial, consultando a documentação das bibliotecas usadas você verá que existe literalmente uma infinidade de parâmetros/argumentos que podem ser utilizados além dos defaults e dos que programamos manualmente. Segue abaixo alguns exemplos, dependendo muito das aplicações de suas redes neurais, da forma como você terá que adaptá-las a alguma particularidade de algum problema, vale a pena se aprofundar neste quesito, descobrir novos parâmetros e realizar testes com os mesmos. Lembre-se que aqui nossa abordagem foi a mais simples, direta e prática possível dentro de uma das áreas de maior complexidade na computação.

```
np.array()
```

```
Arguments
-----------------------------------------------------
array(object, dtype=None, copy=True, order='K',
      subok=False, ndmin=0)
```

```
pd.read_csv()
```

Arguments

```
parser_f(filepath_or_buffer, sep=sep,
         delimiter=None, # Column and Index
         Locations and Names header='infer',
         names=None, index_col=None, usecols=None,
         squeeze=False, prefix=None,
         mangle_dupe_cols=True, # General Parsing
         Configuration dtype=None, engine=None,
         converters=None, true_values=None,
         false_values=None,
         skipinitialspace=False, skiprows=None,
         skipfooter=0, nrows=None, # NA and
         Missing Data Handling na_values=None,
         keep_default_na=True, na_filter=True,
         verbose=False, skip_blank_lines=True, #
         Datetime Handling parse_dates=False,
         infer_datetime_format=False,
         keep_date_col=False, date_parser=None,
         dayfirst=False, # Iteration
         iterator=False, chunksize=None, #
         Quoting, Compression, and File Format
         compression='infer', thousands=None,
         decimal=b'.', lineterminator=None,
         quotechar='"', quoting=csv.QUOTE_MINIMAL,
         doublequote=True, escapechar=None,
         comment=None, encoding=None,
         dialect=None, tupleize_cols=None, # Error
         Handling error_bad_lines=True,
         warn_bad_lines=True, # Internal
         delim_whitespace=False, low_memory=_c_par
         ser_defaults['low_memory'],
```

```
Sequential()
```

Arguments

```
Sequential(self, layers=None, name=None)
```

```
.add(Dense())
```

Arguments

```
Dense(self, units, activation=None, use_bias=True,
      kernel_initializer='glorot_uniform',
      bias_initializer='zeros',
      kernel_regularizer=None,
      bias_regularizer=None,
      activity_regularizer=None,
      kernel_constraint=None,
      bias_constraint=None, **kwargs)
```

```
.fit()
```

Arguments

```
fit(self, x=None, y=None, batch_size=None,
    epochs=1, verbose=1, callbacks=None,
    validation_split=0., validation_data=None,
    shuffle=True, class_weight=None,
    sample_weight=None, initial_epoch=0,
    steps_per_epoch=None, validation_steps=None,
    **kwargs)
```

```
.add(LSTM()
```

Arguments

```
LSTM(self, units, activation='tanh',
     recurrent_activation='hard_sigmoid',
     use_bias=True,
     kernel_initializer='glorot_uniform',
     recurrent_initializer='orthogonal',
     bias_initializer='zeros',
     unit_forget_bias=True,
     kernel_regularizer=None,
     recurrent_regularizer=None,
     bias_regularizer=None,
     activity_regularizer=None,
     kernel_constraint=None,
     recurrent_constraint=None,
     bias_constraint=None, dropout=0.,
     recurrent_dropout=0., implementation=1,
     return_sequences=False, return_state=False,
     go_backwards=False, stateful=False,
     unroll=False, **kwargs)
```

```
.add(Dropout()
```

Arguments

```
Dropout(self, rate, noise_shape=None, seed=None,
        **kwargs)
```

```
ImageDataGenerator()
```

```
Arguments

ImageDataGenerator(self, featurewise_center=False,
                   samplewise_center=False, featur
                   ewise_std_normalization=False,
                   samplewise_std_normalization=Fa
                   lse, zca_whitening=False,
                   zca_epsilon=1e-6,
                   rotation_range=0,
                   width_shift_range=0.,
                   height_shift_range=0.,
                   brightness_range=None,
                   shear_range=0., zoom_range=0.,
                   channel_shift_range=0.,
                   fill_mode='nearest', cval=0.,
                   horizontal_flip=False,
                   vertical_flip=False,
                   rescale=None,
                   preprocessing_function=None,
                   data_format=None,
                   validation_split=0.0,
                   dtype=None)
```

Repare que em nossos exemplos contamos muito com estes parâmetros pré-configurados, que de modo geral funcionam muito bem, mas é interessante saber que sim existe toda uma documentação sobre cada mecanismo interno de cada ferramenta dessas, para que possamos evoluir nossos modelos adaptando-os a toda e qualquer particularidade.

CAPÍTULO RESUMO

Muito bem, foram muitas páginas, muitos exemplos, muitas linhas de código, mas acredito que neste momento você já tenha construído bases bastante sólidas sobre o que é ciência de dados e aprendizado de máquina.

Como nossa metodologia foi procedural, gradativamente fomos abordando tópicos que nos levaram desde uma tabela verdade até uma rede neural artificial convolucional generativa, algo que está enquadrado no que há de mais avançado dentro dessa área da computação.

Apenas agora dando uma breve resumida nos conceitos, desde o básico ao avançado, espero que você realmente não tenha dúvidas quanto a teoria e quanto a aplicação de tais recursos. Caso ainda haja, sempre é interessante consultar as documentações oficiais de cada biblioteca assim como buscar exemplos diferentes dos abordados neste livro. Se tratando de programação, conhecimento nunca será demais, sempre haverá algo novo a aprender ou ao menos aprimorar.

Pois bem, abaixo está a lista de cada ferramenta que usamos (por ordem de uso) com sua explicação resumida:

BIBLIOTECAS E MÓDULOS:

Numpy: Biblioteca de computação científica que suporta operações matemáticas para dados em forma vetorial ou matricial (arrays).

SKLearn: Biblioteca que oferece uma série de ferramentas para leitura e análise de dados.

Pandas: Biblioteca desenvolvida para manipulação e análise de dados, em especial, importadas do formato .csv

Keras: Biblioteca que oferece uma série de modelos estruturados para fácil construção de redes neurais artificiais.

MatPlotLib: Biblioteca integrada ao Numpy para exibição de dados em forma de gráficos.

KNeighborsClassifier: Módulo pré-configurado para aplicação do modelo estatístico KNN.

TrainTestSplit: Módulo pré-configurado para divisão de amostras em grupos, de forma a testá-los individualmente e avaliar a performance do algoritmo.

Metrics: Módulo que oferece ferramentas para avaliação de performance dos processos aplicados sobre uma rede neural.

LogisticRegression: Módulo que aplica a chamada regressão logística, onde se categorizam os dados para que a partir dos mesmos se possam realizar previsões.

LabelEncoder: Módulo que permite a alteração dos rótulos dos objetos/variáveis internamente de forma que se possa fazer o correto cruzamento dos dados.

Sequential: Módulo que permite facilmente criar estruturas de redes neurais sequenciais, onde os nós das camadas anteriores são ligados com os da camada subsequente.

Dense: Módulo que permite que se criem redes neurais densas, modelo onde todos neurônios estão (ou podem estar) interligados entre si.

KerasClassifier: Módulo que oferece um modelo de classificador de amostras pré-configurado e de alta eficiência a partir da biblioteca Keras.

NPUtils: Módulo que oferece uma série de ferramentas adicionais para se trabalhar e extrair maiores informações a partir de arrays.

Dropout: Módulo que funciona como filtro para redes que possuem muitos neurônios por camada em sua estrutura, ele aleatoriamente remove neurônios para que sejam testadas a integridade dos dados dos neurônios remanescentes.

GridSearchCV: Módulo que aplica técnicas de validação cruzada sobre matrizes, podendo fazer o uso de vários parâmetros e comparar qual o que resulta em dados mais íntegros.

OneHotEncoder: Módulo que permite a criação de variáveis do tipo dummy,

variáveis estas com indexação interna própria e para fácil normalização dos tipos de dados.

KerasRegressor: Módulo que oferece um modelo pré-configurado de regressor a partir da biblioteca Keras.

Conv2D: Módulo que permite a transformação de arrays tridimensionais para estruturas bidimensionais para se aplicar certas funções limitadas a 2D.

MaxPooling2D: Módulo que oferece ferramentas para que sejam filtrados dados fora de um padrão, a fim de aumentar a precisão dos resultados.

Flatten: Módulo que permite a redução de dados matriciais de duas ou mais dimensões para uma dimensão, muito usado para vetorizar dados de uma imagem transformando-os em neurônios de uma determinada camada.

BatchNormalization: Módulo que possui internamente ferramentas para que se encontre os melhores padrões de atualização de pesos e valores dos registros de uma rede neural.

ImageDataGenerator: Módulo que permite a conversão e extração de atributos numéricos a partir de imagens para geração de novas imagens.

MinMaxScaler: Módulo muito usado quando necessária a normalização do range de dados dentro de um intervalo pré-estabelecido (entre 0 e 1, por exemplo).

LSTM: Biblioteca que permite a criação de modelos de redes neurais recorrentes. Onde a chamada recorrência se dá pela atualização dos valores do próprio neurônio na própria camada após aplicadas algumas funções.

MiniSom: Biblioteca que possui ferramentas para fácil criação de mapas auto organizáveis.

PColor: Módulo que permite a inserção de marcadores coloridos em cima de gráficos.

ColorBar: Módulo acessório ao PColor que permite a inserção de uma barra de cores que gera um padrão mínimo/máximo de acordo com as características e proximidade dos dados.

RBM: Módulo que permite a criação de modelos de Boltzmann Machines, ou seja, de redes que funcionam como sistemas de recomendação.

Pandas DataReader: Biblioteca oriunda da biblioteca Pandas, oferecendo suporte a importação de dados diretamente de algum site específico.

TQDM: Biblioteca para plotagem de barras de progresso em console.

Gym: Biblioteca de estudos de inteligência artificial, com vários exemplos adaptados de forma a termos acesso ao seu código fonte para que possamos implementar uma inteligência artificial.

ImageIO: Biblioteca que oferece uma série de ferramentas para que se trabalhe diretamente sobre imagens.

SciKit-Image: Biblioteca oriunda da SciKit-Learn voltada para o processamento de dados a partir de imagens.

REFERENCIAL TEÓRICO:

Redes neurais artificiais: Arquitetura de código voltada ao reconhecimento de padrões a partir de processamento de dados de entrada, gerando saídas conforme a necessidade.

Machine Learning: Em tradução livre, aprendizado de máquina, método de cruzamento de dados de vetores/matrizes por meio de conexões e aplicação de funções em camadas de neurônios artificiais.

Deep Learning: Técnica de aprendizado de máquina onde são considerados um número maior de camadas de neurônios e suas respectivas funções, a fim de obter melhores dados a partir de grandes volumes de dados e/ou filtros para detecção de padrões de forma mais minuciosa.

Q-Learning: Técnica de aprendizado de máquina onde o chamado aprendizado por reforço é aplicado, de forma que temos uma rede neural que aprende por tentativa e erro, se retroalimenta e realiza processamento em tempo real.

Data Science: Em tradução livre, ciência de dados, onde grandes volumes de dados históricos, estatísticos ou probabilísticos são processados de forma a obter informação para possíveis classificações, regressões, prospectos ou

previsões a partir dos mesmos.

CONSIDERAÇÕES FINAIS

Como costumo dizer ao final de meus livros, tudo o que tem um começo tem um fim, tratando-se deste pequeno compêndio, apesar do volumoso número de páginas, avançamos alguns poucos passos na vasta gama de possibilidade que temos ao programar em Python.

Espero que de fato a sua leitura tenha sido tão prazerosa quanto foi para mim escrever este humilde livro. Mais importante que isso, espero que tenha contribuído com seus estudos e que tenha aprendido coisas novas a partir deste material.

Jamais pare de estudar, seja um eterno estudante de programação, especialize-se, e certamente estará à frente dos demais que por seus motivos ficam no entorno do básico.

Nos vemos em algum outro livro, curso ou treinamento, até lá...

Um forte abraço

Fernando Feltrin

BÔNUS

Revisão e Exemplos

Tipos de Dados

```
# INT - Número real inteiro, sem casas decimais.

num1 = 12

num2 = int(15)

print(type(num1))
print(type(num2))
```

```
# FLOAT - Número de ponto flutuante / com casas decimais.

num3 = 12.8

num4 = float(19.0)

print(type(num3))
print(type(num4))
```

```
# BOOL - Booleano / Binário (0 ou 1)

condicao1 = 0

sensor01 = True

print(type(condicao1))
print(type(sensor01))
```

```
# STRING - Texto composto de qualquer caractere alfanumérico.

palavra1 = 'palavra'

palavra2 = "marca d'água"

frase1 = 'Uma frase qualquer'

frase3 = 'Ela tinha 8 anos'

frase3 = str('Outra frase qualquer')

print(type(palavra2))
print(type(frase3))
```

```
# LIST - Listas de qualquer tipo.

lista = [2, 'Pedro', 15.9]

lista2 = list([1, 'Maria', 19.99])

print(type(lista))
print(type(lista2))
```

```
# DICT - Dicionários.

dicionario = {'Nome':'Fernando', 'Idade':32}

print(type(dicionario))
```

Comentários

```
# Comentário simples de até uma linha

'''Comentário onde não há limite de linhas,
   podendo ser usado para descrever blocos
   de código mais detalhadamente'''
```

```
# Comentando uma variável para desabilitar a mesma

usuario1 = 'Fernando'     #Legível pelo interpretador

#usuario1 = 'Fernando'    #Ignorado pelo interpretador
```

Variáveis

```
# Sintaxe Básica

nome_da_variavel = dado/valor/atributo

variavel1 = 20
variavel2 = 'Veronica'

print(variavel1)
print(variavel2)
```

```
# Tipos de Variáveis

var_tipo_string = 'Conjunto de caracteres alfanuméricos'
# Incluindo símbolos e caracteres especiais
```

```
var_tipo_int = 12
# Número inteiro

var_tipo_float = 15.3
# Número com casas decimais

var_tipo_bool = 1
# Booleano / Binário (0 ou 1)

var_tipo_lista = [1, 2, 'Paulo', 'Maria', True]
# Lista de dados/valores

var_tipo_dicionario = {'nome':'Fernando', 'idade':31}
# Conjunto de dados:valores

var_tipo_tupla = ('Ana', 'Fernando', 3)
# Lista de dados/valores porém imutável

var_tipo_comentario = """Um comentário atribuído a uma variável deixa de ser apenas
um comentário, vira um texto que pode ser incorporado ao código..."""
```

```
# Declarando uma variável quanto ao seu tipo

numero2 = 1987

ano_nasc = '1987'

print(type(numero2))
print(type(ano_nasc))
```

```
# Declarando múltiplas variáveis

nome, idade, sexo = 'Maria', 32, 'F'

'''
Mesmo que:
nome = 'Maria'
idade = 32
sexo = 'F'
'''

print(nome)
print(idade)
print(sexo)

print(nome, idade, sexo)
```

```
# Declarando múltiplas variáveis de mesmo tipo e dado/valor
```

```
num2 = x = a1 = 10

'''
Mesmo que:
num2 = 10
x = 10
a1 = 10
'''

print(num2)
print(x)
print(a1)
```

```
# Operações entre variáveis

a = 10
b = 5.2

print(a + b)   # Soma simples
```

```
# Interações entre variáveis

msg = 'Olá '
pessoa = 'Carlos'

print(msg + pessoa)   # Concatenação
```

```
# Sintaxe não permitida / não recomendada
# Formas de declarar uma variável que podem gerar erros de interpretação

Nome              Nome de variável iniciando em letra maiúscula
NOME              Nome de variável escrita totalmente em letras maiúsculas
8dados            Nome de variável iniciando com números
_numero           Nome de variável iniciando com caractere especial
minha variavel    Nome de variável com espaços
número = 12345    Nome de variável contendo acentos
(variavel2)       Nome de variável encapsulado em chaves, parênteses ou colchetes

# Sintaxe usual permitida

nome
nome2
nome_de_variavel
```

```
# Palavras reservadas ao sistema
# Palavras que não podem ser usadas como nome de uma variável
```

```
and       del       from       not       while
as        elif      global     or        with
assert    else      if         pass      yield
break     except    import     print
class     exec      in         raise
continue  finally   is         return
def       for       lambda     try
```

Funções Básicas de Entrada e de Saída

```
# Função de saída que exibe em tela ou em terminal algo para o usuário

print('Seja Bem-Vindo!!!')
```

```
# Exibindo em tela o conteúdo de uma variável

nome = 'Fernando'

print('Fernando')
print(nome)
```

```
# Realizando operações aritméticas dentro da função print( )

print(15 + 4)
print(15 - 4)
print(15 * 4)
print(15 / 4)
```

```
# Realizando operações entre dados/valores declarados e dados/valores atribuídos

numero3 = 30

print('O resultado da soma é: ', numero3 + 15)
```

```
# Realizando operações lógicas dentro da função print( )

numero = 9

print(numero > 3)
print(numero < 10)
print(numero == 11)
print(numero >= 12)
```

```
# Usando fStrings e máscaras de substituição dentro da função print( )

nome = 'Fernando'
```

```
print(f'Seja muito bem-vindo {nome}!!!')
```

```
# Usando operadores por meio de fStrings e suas máscaras de substituição

camisa = 19.90
calca = 39.90

print(f'A soma dos produtos do carrinho é: {camisa + calca}')
```

```
# Sintaxe antiga vs atual moderna

nome = 'Maria'

# Sintaxe funcional básica  //  Defasada
print('Bem-vindo', nome,'!!!')

# Sintaxe funcional  //  Defasada
print('Bem-vindo' + ' ' + nome + ' ' '!!!')

# Sintaxe antiga  //  Defasada
print('Bem-vindo %s !!!'%nome)

# Sintaxe usual  //  Defasada
print('Bem-vindo {} !!!'.format(nome))

# Sintaxe moderna atual
print(f'Bem-vindo {nome} !!!')

# Apesar da evolução da linguagem, é perfeitamente possível usar da sintaxe antiga
# de acordo com sua preferência.
```

```
# Função de entrada, interage com o usuário fazendo com que o mesmo insira dados
# Esta função por sua vez deve ser associada a uma variável
# A mesma só termina dua função no momento que o usuário aperta o botão ENTER

nome = input('Digite o seu nome: ')

print(nome)
```

```
nome = input('Digite o seu nome: ')

print(f'Bem-vindo {nome} !!!')
```

Operadores Aritméticos

```
# Soma
```

```
print(5 + 7)
```

```
# Subtração

print(12 - 3)
```

```
# Multiplicação

print(5 * 7)
```

```
# Divisão

print(120 / 6)
```

```
# Mais de uma operação simples

print(5 + 2 * 7)
```

```
# Operação composta

print((5 + 2) * 7)
# Primeiro será realizada a operação dentro dos parênteses, posteriormente o resto.
```

```
# Potenciação

print(3 ** 5) #3 elevado a 5ª potência
```

```
# Divisão exata

print(9.4 // 3)
```

```
# Módulo/Resto de uma divisão

print(10 % 3)
```

Operadores de Atribuição

```
# Atribuindo um dado ou valor a uma variável

nome = 'Fernando'
idade = 33
```

```
# Atribuição aditiva
```

```
valor = 4

valor = valor + 5

print(valor)
```

```
# Atribuição aditiva (método reduzido)

valor = 4

valor += 5

'''
Mesmo que:
valor = valor + 5
'''

print(valor)
```

```
# Atribuição subtrativa

valor = 4

valor -= 3

'''
Mesmo que:
valor = valor - 3
'''

print(valor)
```

```
# Atribuição multiplicativa

valor = 4

valor *= 3

'''
Mesmo que:
valor = valor * 3
'''

print(valor)
```

```
# Atribuição divisiva

valor = 12
```

```
valor /= 2

'''
Mesmo que:
valor = valor / 2
'''

print(valor)
```

```
# Módulo de (ou resto da divisão de)

valor = 12

valor %= 2

'''
Mesmo que:
valor = valor % 4
'''

print(valor)
```

```
# Divisão inteira

valor = 512

valor //= 256

'''
Mesmo que:
valor = valor // 256
'''

print(valor)
```

```
# Exponenciação

valor = 4

valor **= 8

'''
Mesmo que:
valor = valor ** 8
ou
valor = 4 * 4 * 4 * 4 * 4 * 4 * 4 * 4
'''
```

```
print(valor)
```

Operadores Lógicos

```
# Igualdade

print(5 == 6)
print(12 == 12)
```

```
# Diferença

print(7 != 3)
```

```
# Operadores lógicos compostos
# Todas as condições precisam ser verdadeiras para o retorno ser True

print(7 != 3 and 2 > 3)
```

```
# Outras expressões dependem de qual contexto serão aplicadas

num1 = 4
num2 = 9
num3 = 9

print(num1 == num2)
print(num1 is num2)

print(num2 is 9)
print(num2 is num3)
```

Operadores de Membro

```
# Verifica se um elemento é membro daquele tipo de dado

lista = [1, 2, 3, 'Ana', 'Maria']

print(2 in lista)
```

```
# Verifica se um elemento não faz parte daquele tipo de dado

lista = [1, 2, 3, 'Ana', 'Maria']

print('Maria' not in lista)
```

Operadores Relacionais

```
# Maior que

idade = 12

print(idade > 4)
# 3 é maior que 4?
```

```
# Igual ou maior que

num1 = 7

print(num1 >= 7)
# 7 é maior ou igual a 3?
```

```
# Operações entre valores atribuídos a variáveis

x = 2
z = 5

print(x > z)
# O valor de x é maior que o valor de z?
```

```
# Menor ou igual a

y = 7
z = 5

print(z <= y)
# O valor de z é igual ou menor que o valor de y?
```

```
# Duas condições lógicas

x = 2
y = 7

print(x is 2 and y != x)
# O valor de x é 2? E o valor de y é diferente do valor de x?
```

```
# Operadores em estruturas condicionais

idade = 21

if (idade > 18):
      print('Idade adulta.')
```

```
num1 = 2
```

```
num2 = 1

if num1 > num2:
   print('2 é maior que 1')
```

Operadores de Identidade

```
# O valor de uma variável é compatível com outra

aluguel = 250
energia = 250
agua = 65

print(aluguel is energia)
# O valor de aluguel é o mesmo valor de energia?

print(aluguel is agua)
# O valor de aluguel é o mesmo valor de agua?
```

Estruturas Condicionais

```
# if, elif, else
# Estrutura Condicional Simples

nome = input('Digite o seu nome:')

if nome == 'Fernando':
  print('Bem vindo de volta Fernando!!!')

print(f'Você é novo(a) aqui, olá {nome}!!!')

# se a condição for atingida, o bloco dela é executado, caso contrario é simplesmente ignorado
```

```
# Estrutura Condicional com argumento lógico

num = 51

if num < 50:
  print('Menor que 50')
else:
  print('Maior que 50')
```

```
# Estrutura Condicional Composta

nome = input('Digite o seu nome:')

if nome == 'Fernando':
  print('Bem vindo de volta Fernando!!!')
```

```
else:
  print('Você não é Fernando...')

print(f'Você é novo(a) aqui, olá {nome}!!!')
```

```
# Estruturas Condicionais Aninhadas (quando se tem mais de duas possibilidades)

num1 = 12

if num1 <= 50:
  print('Menor que 50')
elif num1 >= 50:
  print('Maior que 50')
else:
  print('Número Inválido')

# Se o valor de num1 for igual ou maior que 50, exibe a respectiva mensagem
# Caso a primeira condição não for verdadeira, é verificado se o valor de num1
# é maior ou igual a 50. Caso nenhuma condição seja verdadeira, é exibida a
# mensagem em else. Quando uma condição for verdadeira, nenhuma das seguintes
# será executada.
```

```
# 1 condição sendo verdadeira já encerra o processo

nome1 = 'Fernando'
nome2 = 'Maria'

if nome1 == 'Fernando':
  print('Bem-vindo Fernando!!!')
#if nome2 == 'Maria':
if nome2 == 'Maria':
  print('Bem-vinda Maria!!!')
else:
  print('Erro: Nome Desconhecido.')

# elif irá executar somente a primeira condição e encerrar o processo se ela
# for verdadeira. elif irá imprimir somente o primeiro print() substituir elif
# por if irá considerar mais de uma condição como verdadeira. if irá imprimir
# o primeiro e o segundo print por ambos serem verdadeiros.
```

```
# 2 ou mais condições sendo verdadeiras

num1 = 12
num2 = 44
nome1 = 'Fernando'
nome2 = 'Maria'

if num1 >= 10 and nome1 == 'Fernando':
```

```
  print('Número maior que 10 e o usuário é Fernando')
if num1 <= 10 and nome1 == 'Fernando':
  print('Número menor que 10 e o usuário é Fernando')
if num1 == num2 and nome2 == 'Maria':
  print('Número 1 e número 2 são iguais, assim como o usuário é Maria')
if num1 != num2 and nome2 == 'Maria':
  print('Número 1 e número 2 são diferentes, assim como o usuário é Maria')

# Operador and exige que as duas condições sejam verdadeiras (uma condição e outra).
```

```
# Operações dentro de estruturas condicionais simples

num1 = 1
num2 = 15

if (num1 + num2) >= 20:
  print('O resultado da soma é maior do que 20')
else:
  print('O resultado da soma é menor do que 20')
```

```
# Estruturas condicionais com interpolação e máscaras de substituição

nomes = ['Fernando', 'Maria', 'Carlos']
login = input('Digite o seu nome: ')

if login in nomes:
  print(f'Bem-vindo(a) {login}')
else:
  print('Usuário não cadastrado.')
```

```
# Operações dentro de estruturas condicionais compostas

num1 = 15
num2 = 40
num3 = 2
soma = num1 + num2 + num3

if (num1 + num2) >= 0:
  print('O número é positivo')
if (num1 + num2) > 50 and soma < 100:
  print('O número é maior que 50 e menor que 100')
```

```
# Estruturas Condicionais com Validadores Simples (em string)

nome = input('Digite o seu nome: ')

if nome == 'Fernando':
  print('Olá Fernando, você é o administraor do sistema!!!')
```

```
elif nome in 'Ana Bárbara Carlos José Maria Paulo Tatiana':
  print(f'Bem vindo(a) {nome}, você é um(a) usuário(a) registrado no sistema.')
else:
  print('Olá, você não está logado no sistema, suas permissões são restritas.')
```

```
# Mesmo exemplo que o anterior, mas diferenciando gêneros

nome = input('Digite o seu nome: ')

if nome == 'Fernando':
  print('Olá Fernando, você é o administraor do sistema!!!')
elif nome in 'Ana Bárbara Maria Tatiana':
  print(f'Bem vinda {nome}, você é uma usuária registrada no sistema.')
elif nome in 'Carlos José Paulo':
  print(f'Bem vindo {nome}, você é um usuário registrado no sistema')
else:
  print('Olá, você não está logado no sistema, suas permissões são restritas.')
```

```
# Mesmo exemplo anterior, aprimorado

nome = input('Digite o seu nome: ')

funcionarios_homens = ['Carlos', 'José', 'Paulo']
funcionarias_mulheres = ['Ana', 'Bárbara', 'Maria', 'Tatiana']

if nome == 'Fernando':
  print('Olá Fernando, você é o administrador do sistema!!!')
elif nome in funcionarias_mulheres:
  print(f'Bem vinda {nome}, você é uma usuária registrada no sistema.')
elif nome in funcionarios_homens:
  print(f'Bem vindo {nome}, você é um usuário registrado no sistema.')
else:
  print('Olá, você não está logado no sistema, suas permissões são restritas.')
```

```
# Estrutura condicional composta com or (ou) e and (e)

veiculo1 = 'Gol'
veiculo2 = 'Corsa'
veiculo3 = 'Onibus'
cor1 = 'Branco'
cor2 = 'Vermelho'

if veiculo1 == 'Gol' or veiculo2 == 'Celta':
  print('Carro')
if veiculo1 == 'Gol' and cor1 == 'Branco':
  print('Gol Branco')
if veiculo1 == 'Onibus' and cor2 == 'Vermelho':
  print('Onibus Vermelho')
```

```
# Condição em or apenas uma das condicionais precisa ser verdadeira
# Condicão em and as duas condicionais precisam serem verdadeiras.
```

Estruturas de Repetição

```
# Enquanto determinada condição for verdadeira, o processo se repete

x = 0
while x <= 5:
  print(x)
  x = x + 1
```

```
# Enquanto o usuário não digitar a palavra específica o código não é executado

validador = input('Digite "Matrix" para continuar:')

while validador.strip() != 'Matrix':
  print('Palavra-chave não confere, digite novamente:')
  validador = input('Digite "Matrix" para continuar:')
```

```
# Estruturas condicionais dentro de laços de repetição

num = 0
total = 10

while num < 10:
  print(num)
  num += 1
  if num == 5:
    print('50% computado')
  if num == 10:
    print('100%, processo encerrado')
```

```
# Reescrevendo o exemplo de palavra-chave específica, aprimorado com while True

while True:
  validador = input('Digite "Matrix" para continuar:')
  if validador.strip() == 'Matrix':
    break
  else:
    print('Palavra-chave não confere, digite novamente:')
```

```
# Laço de repetição simples

compras = ['Arroz', 'Feijão', 'Massa', 'Carne', 'Pão']
for i in compras:
```

```
  print(i)
```

```
# Laço de repetição que percorre cada elemento de uma string

nome = 'Alberto'

for i in 'Alberto':
  print(i)
```

```
# Laço de repetição dentro de um intervalo predefinido

for i in range(0, 10):
  print(i)
```

```
# Laço de repetição que atualiza valor de variável

vendas = [1000, 459, 911, 200, 838, 50]

total = 0

for i in vendas:
  total += i
print(total)
```

```
# Laço de repetição que separa elementos de uma lista

compras = [['Arroz', 'feijão'], ['Carne', 'Frango', 'Peixe'], ['Leite']]

for i in compras:
  print(i)
```

```
# Laço de repetição que separa elementos de um dicionário

cores = {'verde': 'green', 'azul': 'blue', 'vermelho': 'red'}

for i in cores:
  print(i, ':', cores[i])
```

```
# Laço for contando de 2 em 2 elementos

for i in range(0, 10, 2):
  print(i)
```

```
# Contagem Regressiva via laço for

for i in range(10, 0, -1): #terceiro parâmetro estipula que o laço será feito do fim para o começo
```

```
  print(i)
```

```
# Laço for que realiza contagem com base em número digitado pelo usuário

num = int(input('Digite o número limite: '))

for i in range(0, num+1):
  print(i)
```

```
# Laço de repetição com múltiplos parâmetros de múltiplas variáveis

repetir = 's'
fatura = []
total = 0

while repetir == 's':
  produto = input('Digite o nome do produto: ')
  preco = float(input('Digite o preço: '))
  fatura.append([produto, preco])
  total += preco
  repetir = input('Cadastrar mais algum item? (S ou N)').lower()

for i in fatura:
  print(i[0], ':', i[1])

print('O Total da fatura é: ', total)
```

```
# Usando de alternativas quando o laço for não for eficiente

# código convencional - percorrendo os elementos de uma string e exibindo-os junto ao seu índice
nome = 'Fernando'
n_indice = 0

for n_indice in range(len(nome)):
  print(n_indice, nome[n_indice])

# código otimizado via função enumerate( )

for n_indice, valor in enumerate(nome):
  print(n_indice, valor)
```

Strings

```
# Sintaxe básica

nome = 'atributo em formato alfanumérico'
```

```python
# Contar quantos caracteres compõe uma string

frase1 = 'Porto Alegre é uma cidade Brasileira.'

print(len(frase1))
```

```python
# Substituindo elementos de uma string

frase1 = 'Porto Alegre é uma cidade Brasileira.'
frase1 = frase1.replace('Porto Alegre', 'Curitiba')

print(frase1)
```

```python
# Contando quantos de um tipo específico de caractere estão na string

frase1 = 'Porto Alegre é uma cidade Brasileira.'

print(frase1.count('a'))
```

```python
# Exibindo a posição do índice de um determinado elemento da string

frase1 = 'Porto Alegre é uma cidade Brasileira.'

print(frase1.find('é'))
```

```python
# Realizando a leitura de um caractere pelo seu índice

frase1 = 'Porto Alegre é uma cidade Brasileira.'

print(frase1[16])
```

```python
# Desmembrando uma string, separando cada palavra da frase

frase1 = 'Porto Alegre é uma cidade Brasileira.'

print(frase1.split())

# split() sem parâmetros considerará os espaços entre as palavras, caso
# queira um delimitador diferente é necessário definir via parâmetro, por exemplo
# split(';'), dessa forma, apenas onde houver ; haverá a separação das strings.
```

```python
# Pegando dados de uma string especificando um intervalo de seu índice

frase1 = 'Porto Alegre é uma cidade Brasileira.'

print(frase1[26:37]) # intervalo específico
```

```
print(frase1[0:5]) # do início até uma posição i
print(frase1[-9]) # de trás para frente
```

```
# Concatenando strings por meio de variáveis

mensagem = 'Seja bem-vinda '
usuario = 'Ana Clara'

base = mensagem + usuario

print(base)
```

```
# Concatenando diferentes tipos de dados em uma string

nome = 'Ana Clara'
idade = 12

print(f'{nome} tem {idade} anos')

aviso = 'Atenção, geradores entrarão em manutenção às: ' + str(22) + ' horas!'

print(aviso)
```

```
# Operadores lógicos em uma string

aviso = 'Não é permitida a entrada de menores de 18 anos'

print(aviso)

print('anos' in aviso)

print('gestante' not in aviso)
```

```
# Letras maiúsculas ou minúsculas em uma string

alerta = 'Risco de Morte'

print(alerta.upper())

print(alerta.lower())
```

```
# Convertendo outro tipo de dado para string

num1 = 5623
num2 = str(num1)

print(type(num1))
```

```
print(type(num2))
```

```
# Removendo espaços no início e no fim de uma string

frase1 = '   Olá, você é o visitante nº 1000 '

print(frase1)
print(frase1.strip())
```

```
# Convertendo todas iniciais das palavras para maiúsculo, como em um título

tema = 'O diagnóstico por imagem como ferramenta para detecção de câncer'

tema = tema.title()

print(tema)
```

```
# Verificando se uma string é composta por letras e números

print('aa44'.isalnum()) # Qualquer caractere alfanumérico

print('aa44'.isalpha()) # Apenas letras

print('aa44'.isdigit()) # Apenas números
```

```
# Adicionando espaços antes e depois de uma string

frase1 = 'Alerta, Risco de Morte!!!'
frase1 = frase1.ljust(50)

print(frase1)

frase2 = 'Alerta, Risco de Morte!!!'
frase2 = frase2.rjust(50)

print(frase2)

frase3 = 'Alerta, Risco de Morte!!!'
frase3 = frase3.center(50)

print(frase3)
```

Listas

```
# Criando uma lista simples

lista1 = ['Ana Clara', 'Carlos', 'Maria', 'Michele', 'Fernando', 1987]
```

```
print(lista1)
```

```
# Descobrindo o elemento de uma lista através de seu número de índice
lista1 = ['Ana Clara', 'Carlos', 'Maria', 'Michele', 'Fernando', 1987]
print(lista1[3])
```

```
# Descobrindo o número de índice de um determinado elemento
lista1 = ['Ana Clara', 'Carlos', 'Maria', 'Michele', 'Fernando', 1987]
print(lista1.index('Michele'))
```

```
# Descobrindo o número de elementos de uma lista
lista1 = ['Ana Clara', 'Carlos', 'Maria', 'Michele', 'Fernando', 1987]
print(len(lista1))
```

```
# Adicionando um novo elemento a lista
lista1 = ['Ana Clara', 'Carlos', 'Maria', 'Michele', 'Fernando', 1987]
lista1.append('Paulo')
print(lista1)
```

```
# Adicionando um novo elemento na lista em uma posição específica
lista1 = ['Ana Clara', 'Carlos', 'Maria', 'Michele', 'Fernando', 1987]
lista1[2] = 'José'
print(lista1)
```

```
# Removendo um elemento de uma lista pelo seu número de índice
lista1 = ['Ana Clara', 'Carlos', 'Maria', 'Michele', 'Fernando', 1987]
lista1.remove(lista1[5])
print(lista1)
```

```
# Listas dentro de listas
```

```
cadastro = [[1, 2, 3, 4], ['Ana', 'Maria', 'Paulo', 'Roberto']]

print(cadastro)
```

Conjuntos Numéricos

```
# Criando um conjunto numérico (set) em Python

conjunto1 = {5, 10, 15, 20}

print(type(conjunto1))
```

```
# Operações entre conjuntos

cj1 = {1, 2, 3, 4, 5}
cj2 = {1, 3, 5, 7, 9}

print(cj2 - cj1)
```

```
# União de conjuntos numéricos

conjunto1 = {5, 10, 15, 20, 25}
conjunto2 = {1, 2, 3, 4, 5}

uniao = conjunto1.union(conjunto2)

print(uniao)

# Os dados que forem comum aos dois conjuntos não serão duplicados.
```

```
# Interseção de dois conjuntos

conjunto1 = {5, 10, 15, 20, 25}
conjunto2 = {1, 2, 3, 4, 5}

uniao = conjunto1.intersection(conjunto2)
#uniao = conjunto1.update(conjunto2)

print(uniao)
```

```
# Operadores lógicos em conjuntos numéricos

conjunto1 = {5, 10, 15, 20, 25}
conjunto2 = {1, 2, 3, 4, 5}
uniao = conjunto1.union(conjunto2)
```

```
print(uniao == conjunto1)
print(uniao >= conjunto2)
```

```
# Diferença entre conjuntos

c1 = {1, 2, 3, 4, 5} - {1, 2}

print(c1)
```

```
# Diferença entre conjuntos associados a variáveis

c1 = {1, 2, 3, 4, 5, 6, 7, 8, 9}
c2 = {1, 3, 5, 7, 9}

print(c1 - c2)
```

Interpolação

```
# Uso de máscaras de substituição

nome = 'Maria'
exp = 33

print(f'Olá {nome}, queremos te parabenizar por seus {exp} anos em nossa empresa.')
```

```
# Operações dentro de máscaras de substituição

nome = 'Maria'
idade = 30

print(f'{nome} tem 3 vezes minha idade, ela tem {3 * idade} anos!!!')
```

Dicionários

```
# Sintaxe básica

dicionario = {'chave': 'valor'}

print(dicionario)
```

```
# Adicionando novos elementos ao dicionário

dicionario1 = {'Nome1':'Paulo'}

print(dicionario1)

dicionario1['Nome2'] = 'Veronica'
```

```
print(dicionario1)
```

```
# Alterando o valor de uma chave do dicionário

dicionario1 = {'Nome1': 'Ana',
               'Nome2': 'Carla',
               'Nome3': 'Maria'}

print(dicionario1)

dicionario1['Nome2'] = 'Bárbara'

print(dicionario1)
```

```
# Acessando um elemento do dicionário

dicionario1 = {'Nome':'Fernando',
               'Idade':32,
               'Sexo':'Masculino',
               'Nacionalidade':'Brasileiro'}

print(dicionario1['Nome'])
print(dicionario1['Nacionalidade'])
```

```
# Usando o construtor de dicionários do Python

dicionario2 = dict(chave1 = 'valor da chave1',
                   chave2 = 'valor da chave 2')

print(dicionario2)
print(type(dicionario2))
```

```
# Consultando apenas as chaves de um dicionário

dicionario2 = dict(chave1 = 'valor da chave1',
                   chave2 = 'valor da chave 2')

dicionario2.keys()
```

```
# Consultando apenas os valores de um dicionário

dicionario2 = dict(chave1 = 'valor da chave1',
                   chave2 = 'valor da chave 2')

dicionario2.values()
```

```
# Verificando se uma chave ou valor consta em um dicionário

# Pesquisando pela chave, obtendo valor
d3 = {'1':'Ana', '2':'Maria', '3':'Paulo', '4':'Marcos'}

print(d3.get('5'))
print(d3.get('2'))

'''Mesmo que:
print('3' in d3)'''

# Pesquisando se o dado/valor consta em uma chave
print('2' in d3.keys())

# Pesquisando se o dado/valor consta em um valor
print('Ana' in d3.values())

# Pesquisando se o dado/valor de um valor de dicionário é o dado/valor que se espera ser
print(d3['2'] == 'Maria')   # o valor da chave '2' é 'Maria' ?
```

```
# Atualizando um elemento do dicionário

d4 = {'1':'A', '2':'B', '3':'C', '4':'D'}

print(d4)

d4.update({'2':'E'})

'''Mesmo que:
d4['3'] = 'F'
print(d4)'''

print(d4)
```

```
# Removendo um elemento do dicionário

d4 = {'1':'A', '2':'B', '3':'C', '4':'D'}

print(d4)

del d4['4']

print(d4)
```

```
# Pesquisando o tamanho de um dicionário / de quantos elementos ele é composto

d4 = {'1':'A', '2':'B', '3':'C', '4':'D', '5':'E', '6':'F'}
```

```
print(len(d4))
```

```
# Imprimindo só chaves e só valores

d4 = {'1':'A', '2':'B', '3':'C', '4':'D', '5':'E', '6':'F'}

print(d4.keys())
print(d4.values())
```

```
# Lendo as chaves ou valores por meio do laço for

d4 = {'1':'A', '2':'B', '3':'C'}
for i in d4.keys():   # somente as chaves
  print('Chaves:', i)

for j in d4.values():  # somente os valores
  print('Valores:', j)

for l in d4.items():   # chaves e valores
  print('Chaves : Valores =', l)

for m in d4.items():
  print('Chaves:',m[0],' - ','Valores:',m[1])

for n, o in d4.items():   # 2 variáveis temporárias n e o, o mesmo que desempacotamento de elementos
  print('Chaves:',n,' - ','Valores:',o)
  print(f'Chaves: {n}, Valores: {o}')   # trazendo para a sintaxe atual f'Strings
```

```
# Listas como valores de uma chave do dicionário

dict = {'almox':['Folha de Oficio', 'Caneta', 'Grampeador'],
        'cozinha':['Café', 'Açúcar']}

print(dict['almox'][0])
print(dict['cozinha'][1])
```

```
# Transformando chaves ou valores de um dicionário em tupla para fácil indexação

dicionario1 = {'Nome':'Fernando', 'Idade':32, 'Sexo':'Masculino', 'Nacionalidade':'Brasileiro'}

print(dicionario1.keys())

dict_chaves = tuple(dicionario1.keys())

print(dict_chaves)
print(dict_chaves[3])
```

```
# Dicionários dentro de dicionários

usuarios = {'João':{'Identificador':'0001','Cargo':'Porteiro','Salario':'2000'},
            'Maria':{'Identificador':'0003','Cargo':'Aux. Limpeza','Salario':'1900'},
            'José':{'Identificador':'0002','Cargo':'Técnico','Salario':'2500'}}

for i, j in usuarios.items():
  print(f'Funcionário: {i}')
  for k, l in j.items():
    print(f'\t {k} = {l}')
```

```
# Removendo elementos de um dicionário por meio da função pop( )

dicionario1 = {'Nome':'Fernando', 'Idade':32, 'Sexo':'Masculino', 'Nacionalidade':'Brasileiro'}

dicionario1.pop('Nacionalidade')

'''
Mesmo que:
del dicionário1['Nacionalidade']
porém del é uma palavra reservada ao sistema
'''

print(dicionario1)
```

```
# Usando do método get( ) para não gerar exceções quando um elemento não faz parte do dicionário

dict1 = {'1':'A', '2':'B', '3':'C'}

print(dict1['1'])   # ok
#print(dict1['4'])   # chave não existente, irá gerar um KeyError

dict1.get('4', 'nova chave') # '4' é um elemento não existe,te, será substituído por 'nova chave'
                             # caso instanciasse no primeiro parâmetro um elemento presente no
                             # dicionário iria retornar o mesmo normalmente, caso for um elemento
                             # faltante já irá tratar a exceção normalmente, inserindo o novo dado
```

Funções

```
# Sintaxe básica

def nome_da_funcao(parametros):
  "corpo da função"
```

```
# Definindo uma função sem parâmetros

def mensagem():
  print('Seja Bem-Vindo!!!')
```

```
  # poderia ser, dependendo o contexto, return 'Seja Bem-Vindo!!!'

print(mensagem())
```

```
# Chamando a função

def mensagem():
  print('Seja Bem Vindo!!!')

mensagem()
```

```
# Função associada a uma variável

def mensagem():
  print('Seja Bem Vindo!!!')

mensagem1 = mensagem()

print(mensagem1)
```

```
# Criando uma função que temporariamente não faz nada

def funcao():
  pass

var1 = funcao()   # Não acontecerá absolutamente nada

print(type(var1))   # Retornará NoneType
```

```
# Função interagindo com variável que interage com o usuário

usuario3 = input('Digite o seu nome:')

def mensagem(nome):
  print('Bem Vindo ' + nome + '!!!')

print(mensagem(usuario3))
```

```
# Passando o parâmetro ao chamar a função

def funcao(msg):
  print(msg)

funcao('Bem Vindo!!!')
```

```
# Passando o parâmetro ao chamar a função = exemplo 2
```

```python
def mensagem(nome):
  print(f'Bem Vindo(a) {nome}!!!')

usuario1 = mensagem('Fernando')

print(mensagem)
```

```python
# Passando mais de um parâmetro para uma função

def mensagem(nome, idade):
  print(f'{nome} tem {idade} anos...')

usuario1 = mensagem('Fernando', 33)   # respeitar a ordem dos parâmetros

print(usuario1)
```

```python
# Definindo parâmetros ''padrão'', para caso o usuário não passe nenhum parâmetro.

def funcao(msg='Olá', nome='usuário'):
  print(msg, nome)

funcao()
```

```python
# Passando apenas um parâmetro e recebendo o resto padrão

def funcao(msg='Olá', nome='usuário', msg2='Seja Bem Vindo!!!'):
  print(msg, nome, msg2)

funcao(nome='Fernando')
```

```python
# Passando parâmetro e interagindo com o usuário

def funcao(mensagem, nome):
  print(mensagem, nome)

funcao('Olá ', input('Digite o seu nome: '))
```

```python
# Interagindo com o usuário #2

def funcao(msg='Ola', nome='usuário', msg2='Seja Bem Vindo!!!'):
  nome = input('Digite o seu nome: ')
  print(msg, nome, msg2)

variavel1 = funcao()
```

```python
# Interagindo com o usuário #3

def funcao(msg='Ola', nome='usuário', msg2='Seja Bem Vindo!!!'):
  return f'{msg} {nome} {msg2}'

variavel1 = funcao(nome=input('Digite o seu nome: '))

print(variavel1)
```

```python
# Passando um parâmetro manualmente e fora da ordem padrão do índice

def msg(nome = 'Ana', idade = 25, prof = 'Cozinheira'):
  print(f'{nome}, {idade} anos, {prof}')

variavel5 = msg(prof = 'Costureira')
# assim não foi preciso declarar as prmeiras variáveis até chegar na prof

print(msg)
```

```python
# Fazendo operações dentro de uma função

# o código:
n1 = int(input('Digite o Primeiro Número:'))
n2 = int(input('Digite o Segundo Número:'))
soma = n1 + n2

print(soma)

# Pode ser reescrito por:
def soma(n1, n2):
  return n1 + n2

n1 = int(input('Digite o Primeiro Número:'))
n2 = int(input('Digite o Segundo Número:'))

print(soma(n1, n2))
```

```python
# Fazendo operações dentro de uma função #2

def aumento_percentual(valor, percentual):
  return(valor + (valor * percentual / 100))

num1 = int(input('Digite o valor: '))
num2 = int(input('Você deseja somar quantos % ? '))
calculo = aumento_percentual(num1, num2)

print('O valor será de: ',calculo,'%')
```

```python
# Estruturas condicionais dentro de funções #1

def repetidor(msg):
  contador = 0
  while contador < 5:
    print(msg)
    contador += 1

print(repetidor(msg=input('Digite algo para ser repetido 5 vezes: ')))
```

```python
# Estruturas condicionais dentro de funções #2

def divisao(n1, n2):
  if n1 == 0 or n2 == 0:
    return 'Operação Inválida'
  return n1 / n2

num1 = int(input('Digite o Primeiro Número:'))
num2 = int(input('Digite o Segundo Número:'))

print(divisao(num1, num2))
```

```python
# Estruturas condicionais dentro de funções #3 - Exercício Fizz Buzz

def fizz_buzz(num):
  if num % 3 == 0 and num % 5 == 0:
    return f'FizzBuzz, {num} é divisível por 3 e por 5'
  if num % 5 == 0:
    return f'Buzz, {num} é divisível por 5'
  if num % 3 == 0:
    return f'Fizz, {num} é divisível por 3)'
  return num

print(fizz_buzz(num=int(input('Digite um Número: '))))
```

```python
# Função com argumentos externos

def funcao(*args):
  for v in args:
    print(v)

argumentos = (1, 2, 3, 'Paulo', 'Ana')

print(funcao(argumentos))
```

```python
# Desempacotando uma lista para que os elementos dela virem argumentos da função

lista1 = ['nome', 'idade', 'sexo', 'nacionalidade']
```

```
def funcao(*args):
  print('Informações Necessárias:')
  print(args)

funcao(*lista1)
```

```
# Função com parâmetros baseados em *args e **kwargs

# Supondo que está cadastrando senhas e usuários em um sistema
def funcao(*args, **kwargs):
  print(args)
  print(kwargs)

senhas_padrao = [12345, 11111, 54321]

funcao(*senhas_padrao, usuario='user', administrador='admin')

# senhas_padrao substituirá args e o parametro nomeado aqui usuario substituirá kwargs
```

```
# Função com parâmetros baseados em *args e **kwargs

# Buscando dados do modelo anterior
def funcao(*args, **kwargs):
  nome = kwargs['usuario']
  nome2 = kwargs['administrador']
  senha1 = args[0]
  senha2 = args[1]

  print('O usuário comum é: ',nome)
  print('O administrador do sistema é: ', nome2)
  print('A senha padrão é: ', senha1)
  print('A senha alternativa é: ', senha2)

senhas_padrao = [12345, 11111]

funcao(*senhas_padrao, usuario='user', administrador='admin')

# senhas_padrao substituirá args e o parametro nomeado aqui usuario substituirá kwargs
```

```
# Função que recebe outra função como parâmetro

def msg_boas_vindas():
  return 'Seja Muito Bem Vindo!!!'

def mestre(funcao):
  return funcao()
```

```
exec = mestre(msg_boas_vindas)

print(exec)
```

```
# Justaposição - ordenados sequencialmente

#            1º    2º     3º
def pessoa1(nome, idade, funcao):
  print(f'{nome} tem {idade} anos, função: {funcao}')

p1 = pessoa1('Fernando', 32, funcao = 'gerente')
```

Expressões Lambda

```
variavel1 = lambda num1, num2: num1 * num2

num1 = int(input('Digite um número:'))
num2 = int(input('Digite outro número:'))

print(num1)
print(num2)

print(variavel1(num1,num2))
```

Escopo Global e Escopo Local

```
variavel1 = 'Paulo'
print('Exibindo direto da variável: ', variavel1)

def funcao1():
  print('Print de dentro da função 1: ', variavel1)

funcao1()

def funcao2():
  variavel1 = 'Paulo Silva'
  print('Print da variável local / função 2: ', variavel1)

funcao2()
print('Printando a variável global novamente: ', variavel1)

# Não confundir o nome da variável, apesar de igual, a variável1 que consta na  função 2 só funciona de
```

```
#  Modificando uma variável global de dentro de uma função

num1 = 100
```

```
print('Variável com seu valor inicial: ', num1)

def modificador():
  global num1
  num1 = 200
  print('Variável alterada dentro da função: ', num1)

modificador()

print('Variável atualizada pela função anterior: ',num1)
```

```
set1 = {1,2,3,4,5,6,7,8,9,0}
print(set1)

# A diferença de um set para um dicionário será que um set não tem aquele par de elementos chave:valo
```

```
# Criando um set via construtor

set1 = set()
set1.add('Ana')
set1.add(23)

print(set1)

set1.add(15.6)
set1.add('Maria')
set1.discard(23)

print(set1)
```

```
# Convertendo lista para set para que sejam removidos os elementos duplicados

lista1 = [1,1,2,3,4,4,4,4,4,5,6,7,8,9,'Luiz', 'Maria', 'Carlos', 'Luiz']
print(lista1)

lista1 = set(lista1) # Transformando em set
print(lista1)

lista1 = list(lista1) # Transformando em lista novamente
print(lista1)
```

List Comprehension

```
# Fazendo operações com elementos de uma lista usando 'comprehension'

lista1 = [1,2,3,4,5,6]
lista2 = [i * 2 for i in lista1]
```

```
# Variável temporária i que percorre cada elemento da lista1 e multiplica por 2

print(lista1)
print(lista2)
```

```
# Desmembrando uma string elemento por elemento

string = '12345678901234567890'
comprehension = [letra for letra in string]

print(comprehension)
```

```
# Pegando partes de uma string

string = '123456789012345678905925249529562956243482649287529843482548258922'

print(string[0:10])
print(string[10:20])
print(string[20:30])
```

```
# Trabalhando com partes de uma string

string = '012345678901234567890123456789012345678901234567890123456789'
n = 10
comp = [i for i in range(0, len(string), n)]
# Separando a string de 10 em 10 de acordo com a variável n

print(comp)

comp2 = [(i, i + n) for i in range(0, len(string), n)]
# Verificando de quanto a quanto é esse intervalo

print(comp2)

comp3 = [string[i:i + n] for i in range(0, len(string), n)]
# Realizando o fatiamento de nossa string

print(comp3)

lista = [string[i:i + n] for i in range(0, len(string), n)]
retorno = '.'.join(lista)
# Caractere "." usado como separador. join( ) para juntar novamente os elementos da lista

print(retorno)
```

```
# Aprimorando um código mudando sua função por uma list comprehension

carrinho = []
```

```
carrinho.append(('Item 1', 30))
carrinho.append(('Item 2', 45))
carrinho.append(('Item 3', 22))
total = 0

for produto in carrinho:
  total = total + produto[1]

print('Método tradicional', total)

# Mesmo que:
total2 = []
for produto in carrinho:
  total2.append(produto[1])
print('Usando funções internas', sum(total2))

# Usando list comprehension
total3 = sum([y for x, y in carrinho])

print('Usando list comprehension', total3)
```

Dictionary Comprehension

```
# Basicamente a mesma lógica de list comprehension, mas quando tempos chave:valor podemos transor

lista = [('chave1','chave2'), ('chave2','valor2'),('chave3','valor3')]

dicionario = {x:y for x, y in lista}
# Mesmo que dicionario = dict(lista)

print(dicionario)
```

```
# Realizando operações dentro do dicionário via comprehension

produtos = [('Caneta',1.99), ('Lápis',1.49), ('Caderno',8.99)]
produtos_com_imposto = {x:y * 1.6 for x, y in produtos}

print('Preços SEM Imposto: ', produtos)
print('Preços COM Imposto: ', produtos_com_imposto)
```

Geradores e Iteradores

```
# Um objeto iterável é aquele com que você consegue interagir com cada elemento de sua composição, 

lista = [1,2,3,4]
nome = 'Fernando'

for i in lista:
  print(i)
```

```
for j in nome:
  print(j)
```

```
# Gerador gera valores onde não existem para poupar tempo e codificação

lista_0_a_100 = list(range(101))

print(lista_0_a_100)
```

```
# Porém dados gerados com muitos elementos tem impacto sobre a memória

import sys

lista100 = list(range(101))
listamilhao = list(range(1000001))

print('Tamanho em bytes:', sys.getsizeof(lista100))
print('Tamanho em bytes:', sys.getsizeof(listamilhao))
```

```
# Guardando valores em uma variável, de forma que a cada vez que eu chame ela ela me retorne um val

import sys
import time

def variavel_iterada():
  variavel = 'Valor 1'
  yield variavel
  variavel = 'Valor 2'
  yield variavel
  variavel = 'Valor 3'
  yield variavel

var1 = variavel_iterada()

print(next(var1))
print(next(var1))
print(next(var1))
```

Zip

```
# Zip para iterar e unir duas listas

cidades = ['Porto Alegre', 'Curitiba', 'Salvador', 'Belo Horizonte']
estados = ['RS', 'PR', 'BH', 'MG']

cidades_estados = zip(cidades, estados)
```

```
for i in cidades_estados:
  print(i)
```

```
# Zip_longest para unir listas de diferentes tamanhos

from itertools import zip_longest

cidades = ['Porto Alegre', 'Curitiba', 'Salvador', 'Belo Horizonte', 'Rio de Janeiro', 'Goiânia']
estados = ['RS', 'PR', 'BH', 'MG']

cidades_estados = zip_longest(cidades, estados)
cidades_estados2 = zip_longest(cidades, estados, fillvalue='Desconhecido')

for i in cidades_estados:
  print(i)

for j in cidades_estados2:
  print(j)
```

Count

```
# Usando o contador do sistema

from itertools import count
contador = count()

for numero in contador:
  print(numero)
  if numero >= 10:
    break
```

```
# Contando dentro de um intervalo personalizado

from itertools import count
contador = count(start = 40)

for numero in contador:
  print(numero)
  if numero >= 50:
    break
```

```
# Personalizando de quantos em quantos números o contador irá trabalhar

from itertools import count
contador = count(start = 0, step = 2)

for numero in contador:
  print(numero)
```

```
  if numero >= 20:
    break
```

Map

```
# Map = função que serve para aplicar uma função em cada elemento ou linha por linha de uma lista/tup

produtos = [
        {'nome': 'item 1', 'preco': 32},
        {'nome': 'item 2', 'preco': 23},
        {'nome': 'item 3', 'preco': 12},
        {'nome': 'item 4', 'preco': 10},
        {'nome': 'item 5', 'preco': 55},
]

lista = [1,2,3,4,5,6,7,8,9,10]

nova_lista = map(lambda x: x * 2, lista)

# Mesmo que:
nova_lista2 = [x * 2 for x in lista]

print(lista)
print(list(nova_lista))
print(nova_lista2)
```

```
# Usando map( ) para extrair dados de um dicionário

produtos = [
        {'nome': 'item 1', 'preco': 32},
        {'nome': 'item 2', 'preco': 23},
        {'nome': 'item 3', 'preco': 12},
        {'nome': 'item 4', 'preco': 10},
        {'nome': 'item 5', 'preco': 55},
]

precos = map(lambda p: p['preco'], produtos)

for preco in precos:
  print(preco)
```

```
# Usando uma função ativa no lugar da lambra (que é uma função vazia)

def aumenta_precos(p):
  p['preco'] = p['preco'] * 1.05
  return p

precos2 = map(aumenta_precos, produtos)
```

```
for produto in precos2:
  print(produto)
```

Filter

```
# Mesma coisa que a map, só que trabalha com lógica condicional, servindo para aplicar funções ou extr

pessoas = [
        {'nome': 'Ana', 'idade': 22},
        {'nome': 'Maria', 'idade': 72},
        {'nome': 'Paulo', 'idade': 55},
        {'nome': 'Pedro', 'idade': 68},
        {'nome': 'Rafael', 'idade': 99},
        {'nome': 'Tania', 'idade': 18},
]

idosos = filter(lambda x: x['idade'] > 70, pessoas)

print(list(idosos))
```

```
produtos2 = [
        {'nome': 'item 1', 'preco': 32},
        {'nome': 'item 2', 'preco': 23},
        {'nome': 'item 3', 'preco': 62},
        {'nome': 'item 4', 'preco': 10},
        {'nome': 'item 5', 'preco': 55},
]

def filtra(p):
  if p['preco'] > 50:
    return True

nova_lista3 = filter(filtra, produtos2)
for produtox in nova_lista3:
  print(produtox)
```

Try, Except

```
# Tratando exceções para que não se interrompa a execução de um código

try:
  print(a)
except:
  pass

# Nesse caso nada acontecerá, mas também não acontecerá um erro ou interrupção em função de não co
```

```
# Exibindo que erro aconteceu
```

```
try:
  print(a)
except NameError as erro:
  print('Ocorreu um erro:',erro)

#Except genérico:
except Exception as erro:
  print('Ocorreu um erro inesperado')
```

```
# Exibindo que erro específico aconteceu

try:
  print(a)
except NameError as erro:
  print('Ocorreu um erro:',erro)
```

```
def conversor_num(num):
  try:
    num = int(num) # tentará converter para int
    return num
  except ValueError:
    try:
      num = float(num) # tentará converter para float
      return num
    except ValueError:
      pass # caso não consiga nenhuma opção anterior irá retornar None

num1 = conversor_num(input('Digite um número: '))

if num1 is not None:
  print(num1 + 100)
else:
  print('Operação inválida.')
```

```
# Raise

idade = int(input('Digite sua idade: '))

if idade <= 0:
  raise Exception('Idade inválida!!!')

print(f'Você tem {idade} anos!!!')

# Será gerado um erro Exception: Idade inválida!!!
```

Bibliotecas, Módulos e Pacotes

```
# Função dir( ) exibe quais módulos/pacotes/funções foram carregadas por padrão

dir()
```

```
# Explorando tudo o que vem importado por padrão no Python

import builtins

dir(builtins)
```

```
# Importando módulos que por padrão não são carregados junto ao Python em sua inicialização

import random

dir()

# Agora random( ) está carregada na memória e pronta para uso
```

```
# Verificando o que existe dentro de um módulo/pacote externo

dir(random)

# Todas linhas após '__spec__' são os nomes das funções da biblioteca random disponíveis para serem u
```

```
# Abreviando a nomenclatura de um módulo

import random as ra

for i in range(10):
  print(ra.randint(1,10))
```

```
# Importando apenas uma função específica de um módulo

from random import randint

for i in range(10):
  print(randint(0,10))
```

Criando Módulos Manualmente

```
# Arquivo soma.py

def soma(num1, num2):
    s = int(num1) + int(num2)
    return s
```

```
# Arquivo index.py

import soma

print(f'O resultado da soma é: {soma.soma(8,3)}')
```

Manipulando Arquivos

```
# Abrindo/criando um arquivo para leitura e escrita

arquivo = open('nomedoarquivo.txt', 'w+')
# w+ permite a leitura e escrita

arquivo.write('Prineira Linha \n')
arquivo.write('Segunda Linha \n')
arquivo.write('Terceira Linha \n')
arquivo.write('Quarta Linha \n')

arquivo.close()

arquivo.seek(0,0) # força a leitura do arquivo desde sua posição 0

print(arquivo.read())
# print(arquivo.readline()) linha por linha

arquivo.close()
```

```
# Apagando um arquivo via IDE

import os

os.remove('arquivo.txt')
```

Programação Orientada a Objetos

```
# Sintaxe básica

class Nome:
  objetos
```

```
# Criando uma classe vazia

class Pessoa:
  pass
```

```
# Criando objetos (atributos de classe) dentro de uma classe
```

```python
class Pessoa:
  pass

pessoa1 = Pessoa()

pessoa1.nome = 'Fernando'
pessoa1.idade = 32

print(pessoa1.nome)
```

```python
# Criando funções (métodos) dentro de uma classe

class Pessoa:
  def acao1(self):
    print('Ação 1 sendo executada...')

pessoa1 = Pessoa()

pessoa1.acao1()
```

```python
# Criando uma classe com construtor e variáveis internas

class Pessoa:
  def __init__(self, nome, idade, sexo, altura):
    self.nome = nome
    self.idade = idade
    self.sexo = sexo
    self.altura = altura

pessoa1 = Pessoa('Fernando', 32, 'M', 1.90)

print(pessoa1.nome, pessoa1.idade)
print(f'Bem vindo {pessoa1.nome}, parabéns pelos seus {pessoa1.idade} anos!!!')
```

```python
# Mais de uma função dentro de uma classe

class Pessoa:
  ano_atual = 2019

  def __init__(self, nome, idade):
    self.nome = nome
    self.idade = idade

  def ano_nascimento(self):
    ano_nasc = self.ano_atual - self.idade
    print(f'Seu ano de nascimento é {ano_nasc}')

pessoa1 = Pessoa('Fernando', 32)
```

```
print(pessoa1.ano_nascimento())

# Lembrando que nesse exemplo ano_atual é um atributo da classe, e está acessível a todas funções inter
```

```
# Métodos de classe
# Métodos que até interagem com funções internas, mas retorna dados para o escopo global da classe

class Pessoa:
  ano_atual = 2019

  def __init__(self, nome, idade):
    self.nome = nome
    self.idade = idade

  @classmethod
  def ano_nasc(cls, nome, ano_nascimento):
    idade = cls.ano_atual - ano_nascimento
    return cls(nome, idade)

pessoa2 = Pessoa.ano_nasc('Fernando', 1987)

print(pessoa2.idade)
```

```
# Métodos estáticos
# Aqueles que não possuem instâncias como atributos, funciona como uma função normal dentro da clas

from random import randint

class Pessoa:
  @staticmethod
  def gerador_id():
    gerador = randint(100, 999)
    return gerador

print(Pessoa.gerador_id())
```

```
# @property, Getters e Setters

class Produto:
  def __init__(self, nome, preco):
    self.nome = nome
    self.preco = preco

  def desconto(self, percentual):
    self.preco = self.preco - (self.preco * (percentual / 100))

  #Getter
```

```
  @property
  def preco(self):
    return self._preco

  #Setter
  @preco.setter
  def preco(self, valor):
    if isinstance(valor, str):
      valor = float(valor.replace('R$', ''))

    self._preco = valor

produto1 = Produto('Camiseta', 99)
produto1.desconto(5)
print(produto1.preco)

# Supondo que o usuário passou um valor que não pode ser processado porque foi passado em tipo de da
#    produto2 = Produto('Calça', 'R$59')
#    produto2.desconto(15)
#    print(produto2.preco)
# Irá gerar um erro pois R$59 não é um int, então é necessário criar uma estrutura de métodos que farão

produto2 = Produto('Calça', 'R$59')
produto2.desconto(15)
print(f'Produto2 após aplicação do desconto terá valor de {produto2.preco}')
```

```
# Atributos de classe // Variáveis de classe vão ser as variáveis que estarão declaradas dentro de uma cla

class Classe1:
  var1 = 101001

variavel1 = Classe1()

print(Classe1.var1)
print(variavel1.var1)
```

```
# Mudando um atributo de classe

class Classe1:
  var1 = 101001

variavel1 = Classe1()
Classe1.var1 = 111111

print(Classe1.var1)
```

Encapsulamento

```
# Serve para definir se alguma função de alguma classe será acessível ou não fora dela.
```

```python
# Muito pareceido com o public ou private em outras linguagens de programação.

class BaseDeDados:
  def __init__(self):
    self.dados = {}
    #self._dados = {} declarado como protegido (ainda acessível de fora da classe)
    #self.__dados = {} declarado como privado (inacessível e imutável de fora da classe)

base = BaseDeDados()

print(base.dados)
```

Recursividade

```python
def fatorial(num: int) -> int:
  if num == 1:
    return 1

  return num * fatorial(num - 1)

fator = fatorial(5)
print(fator)
```

Expressões Regulares

```python
import re

minha_string = 'Fernando chegará a meia noite.'
print(re.findall(r'Fernand.', minha_string))

minha_string2 = 'Fernanda chegará a meia noite.'
print(re.findall(r'Fernand.', minha_string2))
```

```python
minha_string = 'Fernando chegará a meia noite.'
print(re.findall(r'Fernando|Fernanda', minha_string))
```

```python
import re

# r'Strings

# Meta caractere " . "

texto1 = '''No dia trinta de março de dois mil e vinte
foi inaugurado o novo ginásio de esportes da escola
estatual Professor Annes Dias, na cidade de Cruz Alta,
no estado do Rio Grande do Sul. A obra inicialmente
orçada em um milhão de, reais acabou não utilizando de
todo o recurso, uma vez que dois grandes empresários da
```

```
cidade, João Fagundes e Maria Terres doaram juntos em
torno de duzentos mil reais. João Fagundes se manifestou
dizendo que apoiou a obra pois acredita no desenvolvimento
da cidade, assim investe regularmente para hospitais e
escolas da mesma. maria Terres não quis se pronunciar
sobre o assunto.'"

print(re.findall(r'.oão', texto1))
print(re.findall(r'.oão|.aria', texto1))
```

```
# Meta caractere " | " significa "ou"
# | significa ou, e tem suporte a número de palavras ilimitadas

print(re.findall(r'.oão|.aria', texto1))
print(re.findall(r'.oão|.aria|mil', texto1))
```

```
# Meta caractere " [] " significa "conjunto de caracteres"
# Usado para definir quais possíveis caracteres estarão numa determinada posição da string.

print(re.findall(r'[Jj]oão|.aria', texto1))
# Suponto que toos os caracteres possíveis no início de João são J maiúsculo e j minúsculo nesse caso ta
```

```
# pode conter quantos possíveis caracteres forem necessarios
print(re.findall(r'[AaBbCcDdEeFfGgHhIiJj]oão', texto1))

# otimizado com um intervalo predefinido
print(re.findall(r'[a-z]oão|[A-Z]oão', texto1))
print(re.findall(r'[a-zA-Z]oão', texto1))
```

```
# Alterando o comportamento de uma expressão regular via flags

print(re.findall(r'jOãO|maRIA', texto1, flags = re.I))
# flags serve para passar uma instrução local, nesse caso, re.I significa ignore case (não diferenciar letra
```

```
# Meta caracteres quantificadores

# * significa 0 ou ilimitado número de vezes
# + significa 1 ou ilimitado número de vezes
# ? significa 0 ou 1, apenas uma vez
# {} significa um número específico ou um intervalo específico de repetições {8} ou {0,15}

mensagem = 'Nãããããããããããooooooo esquecer de pagar a mensalidade, não deixar para depois'

print(re.findall(r'não', mensagem, flags = re.I))

print(re.findall(r'Nã+o', mensagem))
# o operador + ao lado de ã diz que esse caractere ali naquela posição pode se repetir uma ou mais vezes
```

```
print(re.findall(r'Nã+o+', mensagem))

print(re.findall(r'[Nn]ã+o+', mensagem))

print(re.findall(r'nã+o+', mensagem, flags = re.I))
```

```
# Meta caracteres aplicados a um intervalo de caracteres de uma expressão regular
#[A-Za-z0-9]+ significa que vai permitir qualquer caractere, quantas vezes o mesmo estiver repetido

chamando_nome = 'Fernnnnnnnannnnndo'

print(re.findall(r'fer[a-z]+a[Nn]+do', chamando_nome, flags = re.I))
```

```
# substituindo um elemento de uma string via expressão regular

texto1 = '''No dia trinta de março de dois mil e vinte
foi inaugurado o novo ginásio de esportes da escola
estatual Professor Annes Dias, na cidade de Cruz Alta,
no estado do Rio Grande do Sul. A obra inicialmente
orçada em um milhão de, reais acabou não utilizando de
todo o recurso, uma vez que dois grandes empresários da
cidade, João Fagundes e Maria Terres doaram juntos em
torno de duzentos mil reais. João Fagundes se manifestou
dizendo que apoiou a obra pois acredita no desenvolvimento
da cidade, assim investe regularmente para hospitais e
escolas da mesma. maria Terres não quis se pronunciar
sobre o assunto.'''

print(re.sub(r'jOãO', 'Carlos', texto1, flags = re.I))
```

```
minha_string = 'Fernando chegará a meia noite.'

print(re.match(r'Fernando', minha_string))

print(re.match(r'noite', minha_string))
```

```
minha_string2 = 'Fernando chegará, a meia noite.'

print(re.split(r',', minha_string2))
```

Expressões Ternárias

```
# Parecido com o "comprehension", mas iterando sobre estruturas condicionais e sobre expressões boole

num = 2
```

```
'Positivo' if num >= 0 else 'Negativo'
```

Map, Filter, Reduce + List Comprehension

```
estoque = [
    {'Item01': 'Camisa Nike', 'Preco':39.90},
    {'Item02': 'Camisa Adidas', 'Preco':37.90},
    {'Item03': 'Moletom 00', 'Preco':79.90},
    {'Item04': 'Calca Jeans', 'Preco': 69.90},
    {'Item05': 'Tenis AllStar', 'Preco': 59.90}
]

# Método convencional para extrair dados
precos01 = []
for preco in estoque:
    precos01.append(preco['Preco'])

print(f'Preços de estoque (normal) {precos01}')
```

```
# Extraindo dados via Map + Função Lambda
precos02 = list(map(lambda p: p['Preco'], estoque))
for preco in precos02:
    print(preco)

print(f'Preços de estoque (Map + Lambda) {precos02}')
#print(next(precos02)) retornaria apenas o primeiro valor, executando novamente apenas o segundo, e a
```

```
# Extraindo dados via List Comprehension
precos03 = [preco['Preco'] for preco in estoque]

print(f'Preços de estoque (List Comprehension) {precos03}')
```

```
# Extraindo dados via Filter
precos04 = list(filter(lambda p: p['Preco'] >= 60, estoque))
#precisa especificar um parâmetro a ser filtrado

print(f'Preços de estoque (Filter) {precos04}')
```

```
# Combinando Filter e List Comprehension
precos05 = [preco['Preco'] for preco in estoque if preco['Preco'] > 60]
#filtrados somente precos maiores que 60

print(f'Preços de estoque (Filter + List Comprehension) {precos05}')
```

```
# Combinando Map e Filter para extrair um dado
precos06 = list(map(lambda p: p['Preco'],
                    filter(lambda p: p['Preco'], estoque)))

print(f'Preços de estoque (Map + Filter) {precos06}')
```

```
# Extraindo dados via Reduce
from functools import reduce
def func_reduce(soma, valores):
    return soma + valores['Preco']
precos_total = reduce(func_reduce, estoque, 0)

print(f'Soma dos Preços (Reduce) {precos_total}')
```

```
# Extraindo dados via Reduce + List Comprehension
from functools import reduce
precos_total = reduce(lambda soma, valores: soma + valores['Preco'], estoque, 0)

print(f'Soma dos Preços (Reduce + List Comprehension) {precos_total}')
```

```
# Extraindo dados via Reduce (Método Otimizado)
from functools import reduce
precos_total = sum([preco['Preco'] for preco in estoque])

print(f'Soma dos Preços (Reduce Método Otimizado) {precos_total}')
```

Módulos Nativos

```
#import datetime

from datetime import datetime, date, time

data = datetime(2020, 6, 12, 15, 24, 59)

print(data.date())
print(data.time())

print(data.year)
print(data.month)
print(data.day)

print(data.hour)
print(data.minute)
print(data.second)
```

```
# uma datetime como string
```

```
print(data.strftime('%m/%d/%Y %H:%M'))
'''
%Y - Ano com 4 dígitos
%y - Ano com 2 dígitos
%m - Mês com 2 dígitos
%d - Dia com 2 dígitos
%w - Dia da semana (entre 0 (domingo) e 6 (sábado))
%H - Hora com notação de 24 horas
%h - Hora com notação de 12 horas
%M - Minuto com 2 dígitos
%S - Segundos com 2 dígitos
%W - Semana do ano (entre 00 e 53)
%z - Fuso horário'''
```

```
# convertendo uma string em datetime
data2 = datetime.strptime('20200101', '%Y%m%d')
print(data2)
```

```
# substituindo um valor de datetime
data3 = datetime(2020, 6, 12, 15, 24, 59)
#data3.year = 2001 retorna um erro porque o datetime inicial é imutavel
data3 = data3.replace(year = 2001)
print(data3)
```

```
# Diferença entre dois datetimes
delta = data3 - data2
print(delta)
```

Tópicos Avançados em POO

```
# Métodos que até interagem com funções internas, mas retorna dados para o escopo global da classe

class Pessoa:
  ano_atual = 2020

  def __init__(self, nome, idade):
    self.nome = nome
    self.idade = idade

  @classmethod
  def ano_nasc(cls, nome, ano_nascimento):
    idade = cls.ano_atual - ano_nascimento
    return cls(nome, idade)

pessoa2 = Pessoa.ano_nasc('Fernando', 1987)

print(pessoa2.idade)
```

```python
# Aqueles que não possuem instâncias como atributos, funciona como uma função normal dentro da cla

from random import randint

class Usuario:
  @staticmethod
  def gerador_id():
    gerador = randint(100, 999)
    return gerador

print(Usuario.gerador_id())
```

```python
class Usuario:

  # Método construtor
  def __init__(self, nome, identificador):
    self.nome = nome
    self.identificador = identificador

  @classmethod
  def autenticador(cls, identificador):

  @staticmethod
  def gerador_cod_seguranca():
```

```python
# Combinações

from itertools import combinations

lista10 = ['Ana', 'Bianca', 'Carla', 'Daniela', 'Franciele', 'Maria']

for combinacoes in combinations(lista10, 2):# possíveis combinações em grupos de 2
  print(combinacoes)

# a ordem não importa e não haverão combinações repetidas entre os elementos
```

```python
# Permutações

from itertools import permutations

lista10 = ['Ana', 'Bianca', 'Carla', 'Daniela', 'Franciele', 'Maria']

for combinacoes in permutations(lista10, 2): # possíveis combinações em grupos de 2
  print(combinacoes)

# Neste caso, as combinações podem se repetir, alterando a ordem dos elementos
# Ex: Ana e Bianca, Bianca e Ana...
```

```python
# Product

from itertools import product

lista10 = ['Ana', 'Bianca', 'Carla', 'Daniela', 'Franciele', 'Maria']

for combinacoes in product(lista10, repeat = 2): # possíveis combinações em grupos de 2
  print(combinacoes)

# Parecido com Permutations, mas aqui é permitido que um elemento se combine com ele mesmo
```

```python
# Associação de Classes

class Usuario:
    def __init__(self, nome):
        self.__nome = nome
        self.__logar = None
    @property
    def nome(self):
        return self.__nome
    @property
    def logar(self):
        return self.__logar
    @logar.setter
    def logar(self, logar):
        self.__logar = logar

class Identificador:
    def __init__(self, numero):
        self.__numero = numero
    @property
    def numero(self):
        return self.__numero
    def logar(self):
        print('Logando no sistema...')

usuario1 = Usuario('Fernando')
identificador1 = Identificador('0001')

usuario1.logar = identificador1
usuario1.logar.logar()
```

```python
# Agregação e Composição de Classes

class Contato:
    def __init__(self, residencial, celular):
        self.residencial = residencial
```

```
        self.celular = celular

class Cliente:
    def __init__(self, nome, idade, fone=None):
        self.nome = nome
        self.idade = idade
        self.fone = []

    def addFone(self, residencial, celular):
        self.fone.append(Contato(residencial, celular))
    def listaFone(self):
        for fone in self.fone:
            print(fone.residencial, fone.celular)

cliente1 = Cliente('Fernando', 32)
cliente1.addFone(33221766, 991357258)

print(cliente1.nome)

print(cliente1.listaFone())
```

```
# Herança Simples

class Corsa:
    def __init__(self, nome, ano):
        self.nome = nome
        self.ano = ano

class Gol:
    def __init__(self, nome, ano):
        self.nome = nome
        self.ano = ano

class Carro:
    def __init__(self, nome, ano):
        self.nome = nome
        self.ano = ano

class Corsa(Carro):
    pass

class Gol(Carro):
    pass
```

```
# Cadeia de Heranças

class Carro:
    def __init__(self, nome, ano):
        self.nome = nome
```

```
        self.ano = ano

class Gasolina(Carro):
    def __init__(self, tipogasolina=True, tipoalcool=False):
        self.tipogasolina = tipogasolina
        self.tipoalcool = tipoalcool

class Jeep(Gasolina):
    pass
```

```
# Mesmo que

class Jeep:
    def carro():
        def __init__(self, nome, ano):
            self.nome = nome
            self.ano = ano

    def gasolina(self, tipogasolina=True, tipoalcool=False):
        self.tipogasolina = tipogasolina
        self.tipoalcool = tipoalcool

jeep = Jeep()
```

```
# Código otimizado

class Carro:
    def __init__(self, nome, ano):
        self.nome = nome
        self.ano = ano

class Gasolina(Carro):
    def __init__(self, tipogasolina=True, tipoalcool=False):
        self.tipogasolina = tipogasolina
        self.tipoalcool = tipoalcool

class Jeep(Gasolina):
    pass

carro1 = Jeep()

print(carro1.tipogasolina)
```

```
# Herança Múltipla

class Mercadoria:
    def __init__(self, nome, preco):
        self.nome = nome
```

```
        self.preco = preco

class Carnes(Mercadoria):
    def __init__(self, tipo, peso):
        self.tipo = tipo
        self.peso = peso

class Utensilios:
    def __init__(self, espetos, carvao):
        self.espetos = espetos
        self.carvao = carvao

class KitChurrasco(Carnes, Utensilios):
    pass

pacote1 = KitChurrasco('carne', 14.90)
pacote1.tipo = 'costela'
pacote1.peso = 1
pacote1.espetos = 1
pacote1.carvao = 1
```

```
# Sobreposição de Membros

class Pessoa:
    def __init__(self, nome):
        self.nome = nome

    def Acao1(self):
        print(f'{self.nome} está dormindo')

class Jogador1(Pessoa):
    def Acao2(self):
        print(f'{self.nome} está comendo')

class SaveJogador1(Jogador1):
    pass

p1 = SaveJogador1('Fernando')
print(p1.nome)

p1.Acao1()
p1.Acao2()
```

```
class Pessoa:
    def __init__(self, nome):
        self.nome = nome

    def Acao1(self):
        print(f'{self.nome} está dormindo')
```

```python
class Jogador1(Pessoa):
    def Acao2(self):
        print(f'{self.nome} está comendo')

class SaveJogador1(Jogador1):
    def Acao1(self):
        super().Acao1()
        print(f'{self.nome} está acordado')

p1 = SaveJogador1('Fernando')
p1.Acao1()
p1.Acao2()
```

```python
class SaveJogador1(Jogador1):
    def Acao1(self):
        super().Acao1()
        print(f'{self.nome} está acordado')
```

```python
class Pessoa:
    def acao(self):
        print('Inicializando o sistema')

class Acao1(Pessoa):
    def acao(self):
        print('Sistema pronto para uso')

class Acao2(Pessoa):
    def acao(self):
        print('Desligando o sistema')

class SaveJogador1(Acao2, Acao1):
    pass

p1 = SaveJogador1()
p1.acao()
```

```python
# Classes Abstratas

from abc import ABC, abstractclassmethod

class Pessoa(ABC):
    @abstractclassmethod
    def logar(self):
        pass

class Usuario(Pessoa):
    def logar(self):
```

```
        print('Usuario logado no sistema')

#user1 = Pessoa()
#user1.logar()

user1 = Usuario()
user1.logar()
```

```
# Polimorfismo

from abc import ABC, abstractclassmethod

class Pessoa(ABC):
    @abstractclassmethod
    def logar(self, chaveseguranca):
        pass

class Usuario(Pessoa):
    def logar(self, chaveseguranca):
        print('Usuario logado no sistema')

class Bot(Pessoa):
    def logar(self, chaveseguranca):
        print('Sistema rodando em segundo plano')

user1 = Usuario()
user1.logar('I5m8b4y9')
```

```
# Sobrecarga de Operadores

class Caixa:
    def __init__(self, largura, altura):
        self.largura = largura
        self.altura = altura

caixa1 = Caixa(10,10)
caixa2 = Caixa(10,20)

print(caixa1 + caixa2)
```

```
class Caixa:
    def __init__(self, largura, altura):
        self.largura = largura
        self.altura = altura

    def __add__(self, other):
        largura1 = self.largura + other.largura
        altura1 = self.altura + other.altura
```

```
        return Caixa(largura1, altura1)

    def __repr__(self):
        return f"<class 'Caixa({self.largura}, {self.altura})'>"

caixa1 = Caixa(10,10)
caixa2 = Caixa(10,20)
caixa3 = caixa1 + caixa2

print(caixa3)
```

```
# Gerenciamento de filas via Nodes

from typing import Any

EMPTY_NODE_VALUE = '__EMPTY_NODE_VALUE__'

class Erro(Exception):
  ...

class Node:
  def __init__(self, value: Any) -> None:
    self.value = value
    self.next = Node

  def __repr__(self) -> str:
    return f'{self.value}'

  def __bool__(self) -> bool:
    return bool(self.value != EMPTY_NODE_VALUE)

class Queue:
  def __init__(self) -> None:
    self.first: Node = Node(EMPTY_NODE_VALUE)
    self.last: Node = Node(EMPTY_NODE_VALUE)
    self._count = 0

  def enqueue(self, node_value: Any) -> None:
    novo_node = Node(node_value)

    if not self.first:
      self.first = novo_node
    if not self.last:
      self.last = novo_node
    else:
      self.last.next = novo_node
      self.last = novo_node
    self._count += 1

  def pop(self) -> Node:
```

```
    if not self.first:
      raise Erro('Sem elementos na fila')
    first = self.first

    if hasattr(self.first, 'next'):
      self.first = self.first.next
    else:
      self.first = Node(EMPTY_NODE_VALUE)
    self._count += 1
    return first

  def peek(self) -> Node:
    return self.first

  def __len__(self) -> int:
    return - self._count

  def __bool__(self) -> bool:
    return bool(self._count)

  def __iter__(self) -> Queue:
    return self

  def __next__(self) -> Any:
    try:
      next_value = self.pop()
      return next_value
    except Erro:
      raise StopIteration

fila1 = Queue()
fila1.enqueue('Maria')
fila1.enqueue('Carlos')
fila1.pop()
```

```
# Gerenciamento de filas via Deque

from typing import Deque, Any
from collections import deque

fila2: Deque[Any] = deque()

fila2.append('Ana')
fila2.append('Carlos')
fila2.append('Fernando')
fila2.append('Maria')

for pessoas in fila2:
  print(pessoas)
```

```
fila2.popleft()

for pessoas in fila2:
  print(pessoas)

# Deque trabalha com índices, sendo assim, ao tentar remover um elemento que não consta no índice 0,
```

PYTHON + NUMPY

Instalação das dependências

```
pip install numpy
```

Importando a biblioteca numpy

```
import numpy as np
```

Criando uma array numpy

```
data = np.array([2, 4, 6, 8, 10])

print(data)
print(type(data))
print(data.shape)
```

Criando automaticamente uma array numpy com dados ordenados.

```
data = np.arange(15)

print(data)
```

Criando automaticamente uma array numpy com dados aleatórios (float entre 0 e 1)

```
data = np.random.rand(15)

print(data)
```

Criando uma array numpy com dados aleatórios (int entre 0 e o valor especificado).

```
data = np.random.randint(10, size = 10)
# números de 0 a 10, size define quantos elementos serão gerados
```

```
print(data)
```

Criando uma array numpy com dados aleatórios (float entre 0 e 1)

```
data = np.random.random((2, 2))

print(data)
```

Criando uma array numpy composta de zeros

```
data = np.zeros((3, 4))
# 3 linhas e 4 colunas

print(data)
```

Criando uma array numpy composta de números um

```
data = np.ones((3, 4))
# 3 linhas e 4 colunas

print(data)
```

Criando uma array numpy com valores vazios.

```
data = np.empty((3, 4))
# haverá um valor, representado em notação científica, algo muito próximo ao zero absoluto.

print(data)
```

Criando uma array numpy de números negativos.

```
data = np.arange(10) * -1

print(data)
```

Criando uma array numpy unidimensional

```
data = np.random.randint(5, size = 10)
# números de 0 a 5, size define quantos elementos serão gerados

print(data)
print(data.shape)
```

Criando uma array numpy bidimensional

```
data2 = np.random.randint(5, size = (3, 4))
# números de 0 a 5, distribuidos em 3 linhas e 4 colunas

print(data2)
print(data2.shape)
```

Criando uma array numpy tridimensional

```
data3 = np.random.randint(5, size = (5, 3, 4))
# números de 0 a 5, 5 camadas cada uma com 3 linhas e 4 colunas

print(data3)
print(data3.shape)
```

Convertendo uma array multidimensional para unidimensional

```
data4 = np.array([(1,2,3,4),(3,1,4,2)])

print(data4)

data4_flat = data4.flatten()

print(data4_flat)
```

Criando uma array numpy a partir de elementos de uma lista

```
data4 = np.array([[1, 2, 3, 4], [1, 3, 5, 7]])

print(data4)
```

Verificando o tipo de dado de uma array

```
print(type(data3))
print(data3.dtype)
```

Verificando o tamanho de uma array

```
data3 = np.random.randint(5, size = (5, 3, 4))

print(data3.ndim)
# número de dimensões

print(data3.shape)
# formato de cada dimensão

print(data3.size)
```

```
# tamanho total da matriz
```

Verificando o tamanho em bytes de cada item e de toda uma array

```
print(f'Cada elemento possui {data3.itemsize} bytes')

print(f'Toda a matriz possui {data3.nbytes} bytes')
```

Consultando o elemento de maior valor em uma array

```
data = np.arange(15)

print(data)
print(data.max())
```

Consultando um elemento através de seu índice (unidimensional)

```
data = np.arange(15)

print(data)
print(data[8])
```

```
data

print(data)
print(data[-5])
# 5º elemento a contar do fim para o começo
```

Consultando um elemento através de seu índice (multidimensional)

```
data2 = np.random.randint(5, size = (3, 4))

print(data2)
print(data2[0, 2])
# linha 0, coluna 2 (lembrando que o índice para coluna também começa em 0, 0-1-2-3)
```

Consultando um intervalo de elementos em uma array

```
data

print(data)

print(data[:5])
# apenas os 5 primeiros
```

```
print(data[3:])
# do terceiro elemento em diante (elemento, e não índice)

print(data[4:8])
# do quarto elemento até o oitavo (4º, 5º, 6º 7º e 8º elemento)

print(data[::2])
# de dois em dois elementos

print(data[3::])
# pula os 3 primeiros elementos e imprime o resto
```

Modificando dados/valores manualmente por meio de seu índice

```
data2

print(data2)

data2[0, 0] = 10
# elemento situado na posição 0x0 do índice passará a ter o valor atribuído 10

print(data2)
```

Inserindo elementos diferentes de int

```
data2

data2[1, 1] = 6.82945
# mesmo declarado como float será convertido em int

print(data2)
```

Definindo manualmente o tipo de dados de uma array

```
data2 = np.array([8, -3, 5, 9], dtype = 'float')
# todos elementos serão convertidos para float

print(data2)
print(type(data2[0]))
```

Criando uma array numpy de intervalos igualmente distribuídos

```
data = np.linspace(0, 1, 7)
# 7 números, gerados de 0 a 1, com intervalo igual entre cada número gerado

print(data)
```

Criando uma array numpy com formato definido manualmente

```
data5 = np.arange(8)

print(data5)

data5 = data5.reshape(2, 4)
# a multiplicação dessas dimensões deve ser o número de elementos. ex 2 x 4 = 8

print(data5)
```

Criando uma array numpy com tamanho predefinido em variável

```
tamanho = 10

data6 = np.random.permutation(tamanho)
# matriz de 10 elementos conforme variável tamanho define

print(data6)
```

Operadores lógicos em arrays numpy (todos elementos)

```
data7 = np.arange(10)

print(data7)
print(data7 > 3)
# irá verificar cada elemento da matriz, retornando true para os que forem maior que 3 e false para os que forem menor.
```

Operadores lógicos em arrays numpy (elemento específico)

```
data7 = np.arange(10)

print(data7)
print(data7[4] > 5)
# o elemento e é maior que 5?
```

Operadores aritméticos em arrays numpy

```
data8 = [1, 2, 3, 4, 5, 6, 7, 8]
print(data8)
```

```
data8 = np.array(data8)
# após convertido para array o operador + sempre fará a soma
```

```
print(data8 + 10)
```

Criando uma matriz diagonal

```
data9 = np.diag((1, 2, 3, 4, 5, 6, 7, 8))
# os elementos serão distribuídos em uma coluna diagonal

print(data9)
```

Criando padrões duplicados

```
data10 = np.tile(np.array([[9, 4], [3, 7]]), 4)
# 9, 4, elementos da primeira linha, 3, 7 da segunda, duplicados 4 vezes na mesma linha

print(data10)
```

```
data10 = np.tile(np.array([[9, 4], [3, 7]]), (2,2))
# mesmo exemplo que o anterior, mas duplicados linha e coluna

print(data10)
```

Somando um valor a cada elemento de uma array

```
data11 = np.arange(0, 15)

print(data11)

data11 = np.arange(0, 15) + 1
# irá somar 1 a cada elemento

print(data11)
```

```
data11 = np.arange(0, 30,3) + 3
# somando 3 ao valor de cada elemento, de 3 em 3 elementos

print(data11)
```

Realizando a soma de arrays

```
d1 = np.array([ 3,  6,  9, 12, 15])
d2 = np.array([ 1,  2,  3,  4,  5])

d3 = d1 + d2
```

```
print(d3)
```

Realizando a subtração de arrays

```
d1 = np.array([ 3,  6,  9, 12, 15])
d2 = np.array([ 1,  2,  3,  4,  5])

d3 = d1 - d2

print(d3)
```

Realizando a divisão de arrays

```
d1 = np.array([ 3,  6,  9, 12, 15])
d2 = np.array([ 1,  2,  3,  4,  5])

d3 = d1 / d2

print(d3)
```

Realizando a multiplicação de arrays

```
d1 = np.array([ 3,  6,  9, 12, 15, 18, 21, 24, 27, 30])
d2 = np.array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])

d3 = d1 * d2

print(d3)
```

Operações lógicas entre arrays

```
d1 = np.array([ 3,  6,  9, 12, 15, 18, 21])
d2 = np.array([ 1,  2,  3,  4,  5, 20, 25])

d3 = d2 > d1
# os elementos de d2 são maiores do que os de d1?

print(d3)
```

Transposição de arrays

```
arr1 = np.array([[1, 2, 3], [4, 5, 6]])

print(arr1)
print(arr1.shape)
```

```
arr1_transposta = arr1.transpose()
# realizará a conversão de linhas para colunas e vice-versa

print(arr1_transposta)
print(arr1_transposta.shape)
```

Salvando uma array no disco local

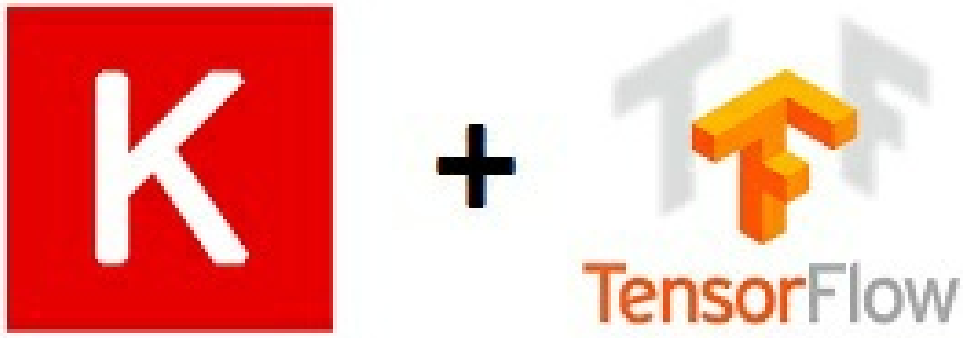
```
data = np.array([ 3,  6,  9, 12, 15])

np.save('minha_array', data)
# irá gerar um arquivo com extensão .npy
```

Carregando uma array do disco local

```
np.load('minha_array.npy')
```

Artigo - Classificador Multiclasse via TensorFlow e Keras



A partir do momento em que estamos buscando resolver algum tipo de problema computacional por meio de redes neurais artificiais, é de suma importância definirmos exatamente que tipo de problema estamos trabalhando, para escolhermos corretamente as ferramentas que serão utilizadas, assim como o modelo de rede neural que melhor se adapta para aquele fim. Uma vez que uma rede neural é um modelo estrutural de processamento de dados visando encontrar padrões e aprender a partir dos mesmos.

É bastante comum termos modelos específicos de redes neurais artificiais que realizam todo tipo de classificação, um desses modelos, chamado usualmente de rede neural convolucional, é um modelo especializado em resolver problemas de classificação a partir de imagens. Nesse sentido, existe toda uma área em machine learning e deep learning dedicada a realizar o processo chamado de convolução, onde as imagens, dadas as suas características de composição através de pixels e suas matrizes de cores, são convertidas para matrizes numéricas que por sua vez irão compor camadas de neurônios artificiais da rede neural para cruzamento dos dados.

Uma vez realizadas as devidas conversões, pixel a pixel, para

arrays de um dataframe, é possível treinar uma rede neural artificial para que encontre padrões específicos e característicos de uma imagem a fim de classificar a mesma quanto a sua forma.

A nível de rede neural artificial convolucional, podemos, utilizando de conceitos de machine learning, treinar uma rede para que diferencie, por exemplo, diferentes animais entre si, diferentes objetos quanto ao seu formato, diferentes pessoas quanto às suas características únicas que compõe seu rosto, e, como será exemplificado neste capítulo, diferenciar a partir de exames de imagem, com uma precisão muito maior do que a de um médico especialista, características de apresentação de uma anatomia normal ou patológica..

O exemplo que veremos trata justamente desse tipo de classificação, realizado via rede neural artificial convolucional. A área da radiologia, também conhecida como diagnóstico por imagem, é o ramo da medicina onde por meio de exames como ultrassonografia, raios-x, tomografia computadorizada, ressonância magnética, cintilografia, entre outros exames, conseguimos obter imagens fidedignas da anatomia para que sejam feitos os devidos estudos em busca de quaisquer tipo de alteração morfológica, fisiológica ou funcional. Nesta área em particular, além dos equipamentos distintos para cada tipo de exame, existe toda uma equipe multidisciplinar composta por técnicos e médicos radiologistas.

O médico radiologista, profissional este que se dedica exclusivamente a interpretar exames de imagem, possui formação médica somada a uma especialização que o treina a identificar características bastante minuciosas, como a diferença de densidade ou de atividade metabólica entre tecidos. O mesmo, com base em seu conhecimento da anatomia normal assim como a experiência

em clínica médica, identifica em exames de imagem características que possam apresentar alterações da anatomia indicando patologias. O ponto a se destacar é que, por melhor e mais experiente que este tipo de profissional seja, ele possui devido as suas limitações como ser humano, uma margem de erro a ser considerada em cada interpretação de exame.

É nesse momento que a coisa começa a ficar interessante, no sentido de que conseguimos elaborar e treinar uma rede neural artificial para que, a partir de uma base de dados previamente usada para aprendizado de máquina, possa realizar o processamento de novas imagens com nível de precisão maior do que a do próprio médico radiologista. Tenha em mente que toda subdivisão da área de inteligência artificial vem a somar, e muito, em várias áreas onde as mesmas estão sendo aplicadas, não longe disso, a área da medicina tem obtido bons e significativos progressos graças à usabilidade destas tecnologias e ferramentas. Tal tipo de tecnologia não visa substituir nenhum profissional desta área, mas servir como uma ferramenta que fornece um segundo viés de confirmação, para que sejam emitidos laudos mais precisos, e consequentemente tratamentos mais eficazes para os pacientes.

Nas páginas seguintes, estaremos elaborando e criando passo a passo, uma rede neural artificial convolucional dedicada a processar e interpretar radiografias de tórax, onde a anatomia principal demonstrada são os pulmões e a região do mediastino da caixa torácica, em busca de alguma patologia do trato respiratório como pneumonia, tuberculose e derrame pleural. Para isso, usaremos da linguagem Python assim como duas de suas mais importantes bibliotecas para estruturação de redes neurais densas chamadas TensorFlow e Keras.

Uma das características interessantes da biblioteca Keras é o fato da mesma, ter capacidade de criar redes neurais convolucionais partindo de amostras de imagens. Diferente de outros modelos onde é necessário aprendizado supervisionado, ou por reforço, ou conciliando labels sobre as imagens para que destaquem suas características, biblioteca Keras por meio de suas funções internas consegue fazer classificação de dois ou mais tipos de imagens simplesmente as separando em pastas de acordo com o seu tipo, convertendo tais tipos de dados para arrays que irão compor camadas de uma rede neural e passar por uma série de funções para os devidos reconhecimentos de padrões a partir destas imagens.

O que faremos, etapa por etapa, é a identificação do tipo de problema computacional, neste caso uma classificação “multiclasse” a partir de imagens, posteriormente preparar uma base de dados a ser utilizada e dando sequência teremos de fato a codificação da estrutura da rede neural artificial e por fim teremos um modelo treinado para realização de testes a partir do mesmo.

Temos um problema computacional idealizado, hora de estruturar o mesmo de forma lógica para que possamos construir um modelo. Para realizarmos a classificação a partir de radiografias de tórax, temos de preparar uma base de dados com imagens que serão processadas para o treinamento de nossa rede neural artificial convolucional. Para isso, podemos buscar datasets públicos de radiografias de tórax ou montar nossa própria base de dados, reunindo imagens separadas quanto às suas características. Nesse caso, é preciso construir uma base composta de imagens de exames sem alterações, e com as respectivas patologias separadas quanto ao seu tipo.

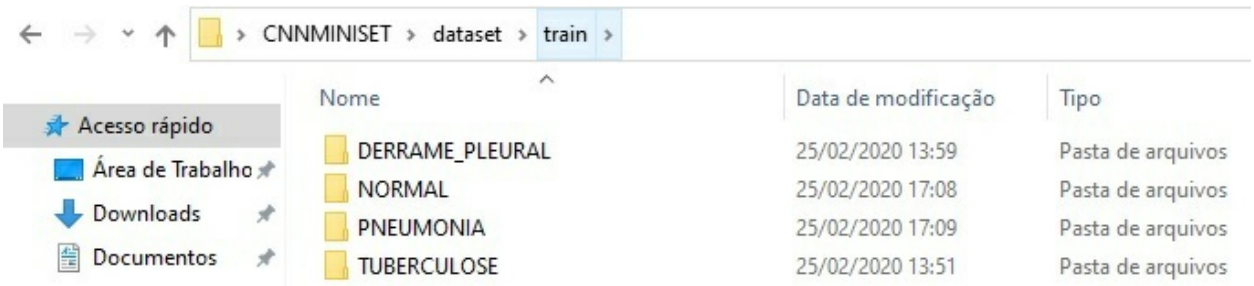


Imagem 01 - Estrutura de pastas para as amostras separando-as quanto ao seu tipo

Ao todo, montamos uma base com apenas 674 imagens com as respectivas características de um exame “normal” ou apresentando algum tipo de patologia (pneumonia, derrame pleural e tuberculose), separados entre si. Importante salientar que assim como toda rede neural artificial, muito da eficiência de nossa rede em seu processo de aprendizado de máquina está diretamente ligado a base de dados usada para esse processo. Principalmente no que se refere a classificação a partir de imagens, uma base de dados mais robusta, com maior número de amostras (dependendo o caso, milhares ou até mesmo milhões de amostras), torna a eficiência de nossa rede neural muito maior.

Uma base de dados pequena pode impactar negativamente o processo de aprendizado por reforço, assim como uma base de dados muito volumosa, se não preparada da maneira correta, pode gerar ruídos no processo de aprendizado de máquina. Note que aqui, especificamente, estamos usando uma base de dados de imagens, onde todas são muito parecidas entre si, ao menos aos olhos de um leigo, todas essas radiografias possuem basicamente as mesmas características, como extensão da caixa torácica, dois pulmões, mediastino, imagens em preto e branco, resolução parecida, etc…, essas características, para esse relativo volume pequeno de amostras, será o suficiente para nosso modelo inicial mas em uma aplicação real o ideal seria utilizarmos de uma base de

dados maior.

Base de dados pronta, com as devidas amostras separadas em pastas de acordo com seu tipo, assim como separadas em base de dados para treino e teste pela rede, podemos continuar nossa linha de raciocínio.

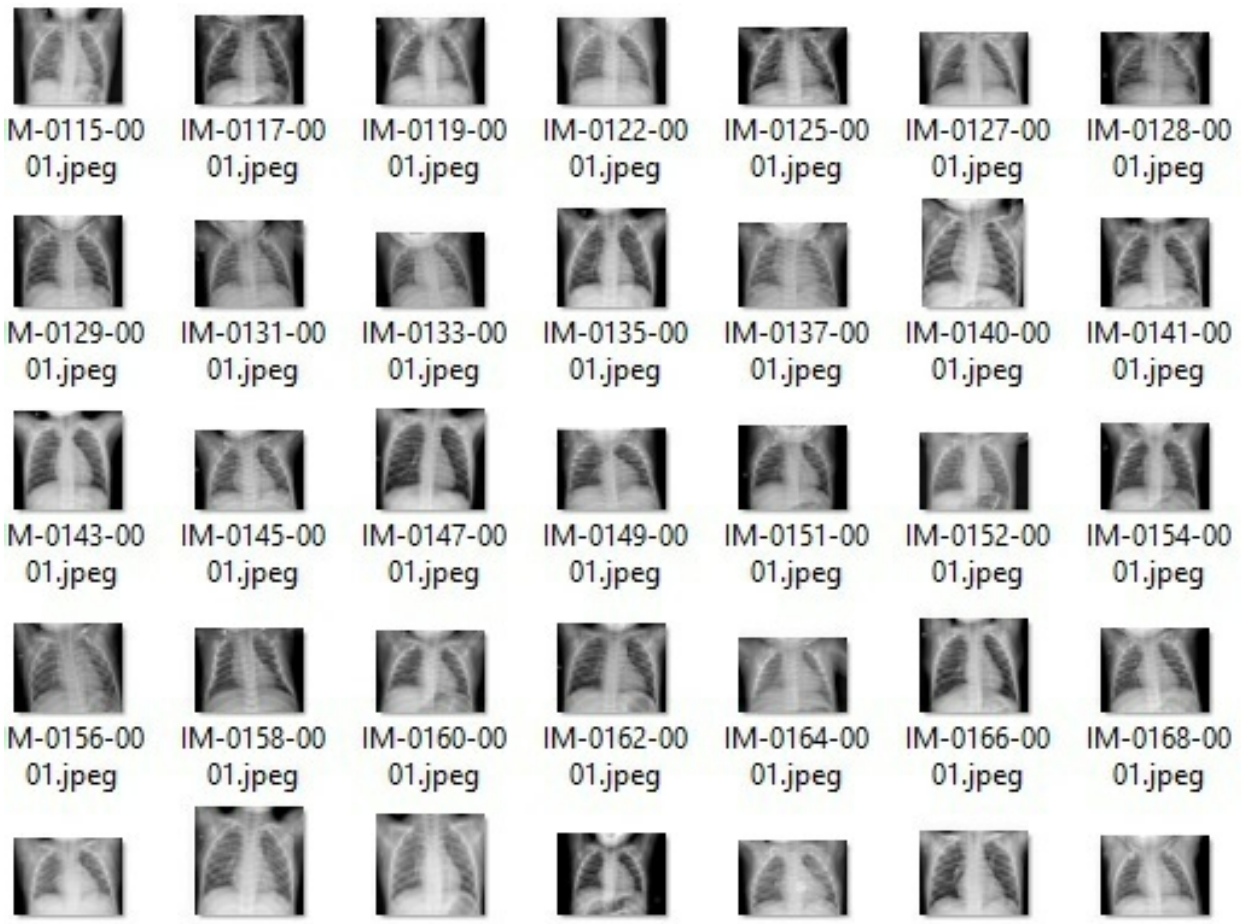


Imagem 02 - Amostras de radiografias de tórax

Para reforçar nosso entendimento, é interessante por hora entendermos o processo de convolução de uma imagem. O processo de conversão de uma imagem em toda sua forma e estrutura, para uma matriz numérica que alimentará perceptrons de camadas de nossa rede neural artificial convolucional é um processo dividido em várias etapas.

Inicialmente teremos uma imagem fonte, com uma determinada resolução dependendo o tipo de exame, que é convertida para preto e branco caso seja originalmente colorida, em seguida a imagem é mapeada e dividida em blocos menores, onde cada bloco será novamente mapeado em busca de características únicas e cada etapa desse mapeamento é indexada internamente como mapas de características. Na sequência, esse mapeamento inicial é replicado e novamente convertido em mapas indexados internamente, agora com as características de cada pixel, de cada bloco, de cada mapa, de cada posição da imagem original. Dando prosseguimento, cada mapa será escalonado e indexado algumas vezes até que por fim essas conversões resultarão em uma matriz de indexação com apenas uma coluna de dados numéricos. Essa coluna por sua vez, será normalizada em uma escala de 0 a 1 e dividida de forma que cada valor de cada linha da mesma se tornará um neurônio da camada de entrada na rede neural artificial. *Cada uma das imagens de nossa base de dados passará por esse processo ao alimentar nossa rede neural.

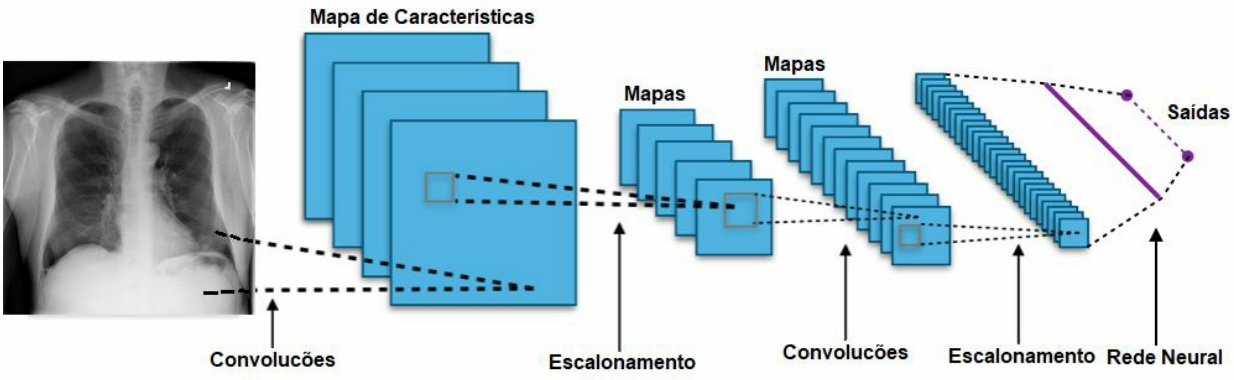


Imagem 03 - Representação das etapas do processo de convolução

Base de dados preparada, hora de partir para o código: Apenas lembrando que as bibliotecas TensorFlow e Keras não vem nativamente instaladas junto a sua distribuição do Python, porém a

instalação das mesmas é facilmente realizada via pip (pip install tensorflow, pip install keras) ou via conda, já que estaremos trabalhando em um ambiente virtual criado para este fim (conda install tensorflow, conda install keras). Uma vez instaladas as bibliotecas, podemos abrir nossa IDE, nesse caso em particular estarei usando a Spyder, ambiente de desenvolvimento muito usado para computação científica principalmente em função da sua maneira intuitiva de apresentar os dados das variáveis de forma visual, assim como a execução independente de cada bloco de código.

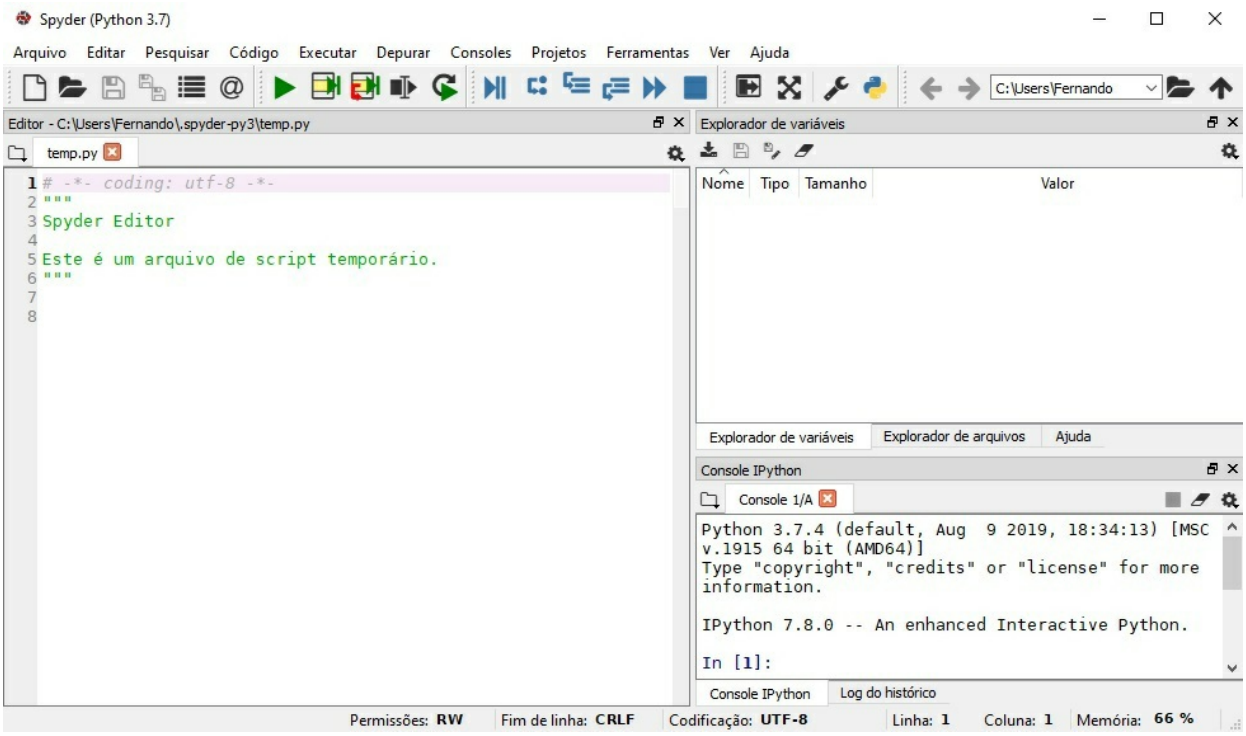


Imagem 04 - Ambiente da IDE Spyder

Apenas fazendo uma pequena observação, ao abrir o Spyder não é necessário realizar nenhuma configuração, a não ser verificar se estamos usando o kernel correto uma vez que estamos trabalhando em um ambiente virtual isolado.

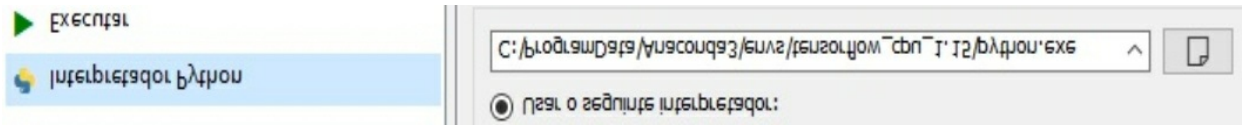


Imagem 05 - Seleção do kernel a ser utilizado para a execução da rede neural

Note que neste caso estarei utilizando a biblioteca TensorFlow em sua versão 1.15, porém o código apresentado é totalmente compatível com as versões a partir da 2.0 da mesma.

Partindo para o código:

Como sempre, o processo se dá iniciando com as importações das bibliotecas, módulos, pacotes e ferramentas que estaremos usando ao longo do código.

```
from keras.layers import Conv2D, MaxPooling2D, Activation, Dropout, Flatten, Dense
from keras.preprocessing import image
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential
from keras import backend as B
import numpy as np
```

Repare que, para fins de performance, importamos especificamente apenas os módulos das bibliotecas que de fato usaremos.

Sendo assim, inicialmente importamos do módulo **layers** da biblioteca **Keras** as funções empacotadas **Conv2D** e **MaxPooling2D** que respectivamente servirão para criar a estrutura de neurônios por camada assim como a conversão das matrizes

para duas dimensões; **Activation** será responsável pela camada de ativação e suas funções de ativação aplicadas sobre os neurônios; **Dropout** que é uma ferramenta que aleatoriamente seleciona e remove neurônios das camadas intermediárias, como uma espécie de filtro, para que seja testado o processamento sem os devidos neurônios, assim como seu impacto na performance dos ajustes dos pesos; **Flatten** que basicamente é uma ferramenta que converterá a última camada de matriz bidimensional para unidimensional, e dessa forma, cada perceptron empilhado destes se tornará um neurônio da camada de entrada da estrutura densa da rede; **Dense** que por sua vez permitirá criar camadas de neurônios conectadas entre si de forma a permitir, entre outras operações, a retroalimentação da rede e todo seu processo de aprendizado de máquina por meio dos ajustes dos pesos entre camadas; Da mesma forma, do módulo **preprocessing** importamos **image**, pacote que permitirá operações diretamente sobre arquivos de imagem, também importamos **ImageDataGenerator**, que possibilitará gerar uma base de dados aumentada, com amostras geradas pela própria rede neural a partir da base de dados inicial, para que o processo de reconhecimento de padrões a partir de imagens diretamente seja mais eficiente, haja visto que como mencionado anteriormente, temos uma base de dados bastante pequena para esse exemplo. Também importamos do módulo **models Sequential**, que nos permitirá a criação de camadas de neurônios subsequentes e conectadas de forma sequencial entre si. Por fim da biblioteca **Keras** importamos **backend** que nada mais é do que um pacote de funções de entrada e de comunicação com outras funções que podem ser oriundas de outras bibliotecas. Encerrando esse processo importamos a biblioteca **Numpy** que permitirá a utilização de dados em forma de matrizes numéricas, assim como a aplicação de funções lógicas e aritméticas sobre este tipo de dado em particular.

```
data_treino =
'C:/Users/Fernando/Desktop/CNNMINISET/dataset/train/'
data_teste =
'C:/Users/Fernando/Desktop/CNNMINISET/dataset/test/'
```

Em seguida criamos as variáveis que instanciação nossas imagens por meio de seus diretórios. Para isso, criamos **data_treino** e **data_teste** com seus respectivos diretórios absolutos (no caso de execução deste código em outra IDE é perfeitamente possível a utilização dos caminhos relativos ‘dataset/train’ e ‘dataset/test’)

```
if B.image_data_format() == 'channels_first':
        input_shape = (3, 150, 150)
else:
        input_shape = (150, 150, 3)
```

Na sequência criamos uma simples estrutura condicional que funcionará como uma estrutura de validação, pois na mesma estaremos realizando a conversão de formato da matriz gerada para que a mesma possa ser corretamente lida pela biblioteca **Numpy** e processada. Por meio da função **image_data_format( )** simplesmente realizamos a verificação se os dados de entrada estão no formato aceito em **‘channels_first’**, onde o primeiro formato (3, 150, 150) era lido pela biblioteca **Keras** usando o antigo núcleo **Theano**, e em versões mais recentes, a biblioteca **Keras** passou a ser incorporada ao **TensorFlow** e dessa forma tem de carregar as matrizes no formato aceito pelo núcleo **TensorFlow**, ou seja, (150, 150, 3).

```
data_gen_treino = ImageDataGenerator(rescale = 1. / 255,
                                     shear_range = 0.2,
                                     zoom_range = 0.2,
```

```
                                    horizontal_flip = True)

data_gen_teste = ImageDataGenerator(rescale = 1. / 255)
```

Seguindo com o código podemos criar o bloco responsável por realizar o aumento de nossa base de dados, o mesmo é feito atribuindo as variáveis **data_gen_treino** e **data_gen_teste** a função **ImageDataGenerator( )** que no primeiro caso recebe alguns parâmetros definidos manualmente. O primeiro deles, **rescale**, será responsável por escalonar as imagens carregadas para um formato proporcional e uniforme entre si; **shear_range** que definirá um nível de angulação (em graus) que será aplicado sobre as imagens; **zoom_range** define o nível de zoom aplicado a imagem e **horizontal_flip** faz o simples espelhamento das imagens. Para os dados de teste é feito apenas o escalonamento das imagens, mantendo o resto dos parâmetros em default.

```
gerador_treino =
data_gen_treino.flow_from_directory(data_treino, target_size =
(150, 150), batch_size = 20, class_mode = 'categorical')
```

Realizadas as parametrizações de como as imagens serão carregadas, criamos o gerador em si, com a variável **gerador_treino** que recebe os dados lidos pela função **flow_from_directory( )**, parametrizada com os dados de **data_treino**, escalonados para 150x150 pixels por meio do método **target_size**, assim como definimos que os ajustes serão realizados a cada 20 amostras por meio do método **batch_size** e por fim, já que estamos classificando múltiplos tipos de “objetos”, definimos **class_mode** como **‘categorical’**.

```
gerador_teste = data_gen_teste.flow_from_directory(data_teste,
target_size = (150, 150), batch_size = 20, class_mode =
'categorical')
```

Exatamente o mesmo processo é realizado, dessa vez, atribuindo os dados gerados a uma base aumentada preparada para ser usada em testes.

```
model = Sequential()
model.add(Conv2D(32, (3,3), input_shape = input_shape))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size = (2,2)))
model.add(Conv2D(32, (3,3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size = (2, 2)))
model.add(Conv2D(32, (3,3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size = (2, 2)))
model.add(Flatten())
model.add(Dense(64))
```

Uma vez que temos nossos dados preparados, podemos partir para criação da estrutura de rede neural em si. Sendo assim, criamos um objeto de nome **model** que receberá atribuído para si uma série de estruturas de código responsáveis pela codificação lógica da rede. Inicialmente atribuímos a **model** a função **Sequential( )**, sem parâmetros mesmo, que por sua vez nos permitirá a criação de uma rede onde as camadas de neurônios que forem adicionadas serão interconectadas entre si, lidas e processadas de forma sequencial. Em seguida, por meio de **Conv2D( )** criamos a primeira camada de convolução, onde

realizamos a transformação dos dados das imagens para uma matriz 3x3, dividida em 32 neurônios, a mesma recebe uma função de ativação ***relu*** (Rectified Linear Unit) sobre si. Em seguida usando da função **MaxPooling2D( )** parametrizada manualmente com valor para **pool_size** é realizada a etapa onde tal estrutura é redimensionada para uma matriz 2x2. Note que esta etapa se repete três vezes, criando camadas de neurônios operando dessa forma para que, de acordo com os reconhecimentos de padrões realizados, os mesmos se tornem estruturas de perceptrons com seus respectivos pesos reajustados. Na sequência é criada uma camada sob a função **Flatten( )** que por sua vez realiza a última conversão das matrizes processadas até então para uma escala unidimensional, que por sua vez se tornarão os neurônios de ativação da camada de entrada da rede densa (até o momento estávamos trabalhando apenas nas camadas de convolução). Também é aplicada uma função **Dropout( )** que removerá, de acordo com o parâmetro setado, 5% de amostras escolhidas aleatoriamente e, por fim, uma camada de saída com 4 saídas, referentes a quantidade de tipos de objetos em questão, que por sua vez passa pela função de ativação **softmax** que produz melhores resultados quando estamos classificando amostras em mais de 2 categorias.

```
model.summary( )
```

```
In [  ]: model.summary()
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 148, 148, 32)      896
_________________________________________________________________
activation_1 (Activation)    (None, 148, 148, 32)      0
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 74, 74, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 72, 72, 32)        9248
_________________________________________________________________
activation_2 (Activation)    (None, 72, 72, 32)        0
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 36, 36, 32)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 34, 34, 32)        9248
_________________________________________________________________
activation_3 (Activation)    (None, 34, 34, 32)        0
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 17, 17, 32)        0
_________________________________________________________________
flatten_1 (Flatten)          (None, 9248)              0
_________________________________________________________________
dense_1 (Dense)              (None, 64)                591936
_________________________________________________________________
activation_4 (Activation)    (None, 64)                0
_________________________________________________________________
dropout_1 (Dropout)          (None, 64)                0
_________________________________________________________________
dense_2 (Dense)              (None, 4)                 260
_________________________________________________________________
activation_5 (Activation)    (None, 4)                 0
=================================================================
Total params: 611,588
Trainable params: 611,588
Non-trainable params: 0
```

Imagem 06 - Sumário da estrutura da rede neural artificial, mostrando todas suas camadas e interconexões

Por meio da função **summary( )** é possível verificar de forma visual toda a estrutura interna da rede neural.

```
model.compile(loss = 'categorical_crossentropy',
        optimizer = 'rmsprop',
        metrics = ['accuracy'])
```

Dando continuidade, é criada a estrutura responsável pela fase de compilação da rede, por meio da função **compile( )** atribuída a model, parametrizada com **loss** (função de "perda") como **'categorical_crossentropy'** que por sua vez agrupará as amostras de acordo com seus respectivos tipos, a medida que a rede for reconhecendo seus padrões; **optimizer** definido como **'rmsprop'**, método que fará a chamada descida do gradiente, onde de acordo com uma taxa de aprendizado, irá conseguir realizar os devidos ajustes dos pesos em cada categoria de forma a "aprender" quais são os melhores e mais estáveis valores para cada padrão reconhecido; Por fim, **metrics** definido como **'accuracy'** estipula que os padrões devidamente reconhecidos deverão ser baseados em parâmetros de margem de precisão em acertos.

```
model.fit_generator(gerador_treino,
                    steps_per_epoch = 100,
                    epochs = 1000,
                    validation_data = gerador_teste,
                    validation_steps = 50)
```

Encerrando o bloco referente a estrutura da rede neural em si, definimos por meio da função **fit_generator( )** alguns parâmetros para a execução da rede. Primeiramente passamos os dados contidos em **gerador_teste**, seguido de **steps_per_epoch** que define quantas vezes serão realizados os devidos processamentos entre as camadas da rede; **epochs** estipula quantas vezes toda a rede neural será novamente executada (assumindo que a cada nova execução a mesma utiliza os melhores valores de pesos encontrados e parte para um novo processamento a fim de a cada rodada aprender melhor os padrões reconhecidos pela rede); **validation_data** recebe os dados gerados anteriormente e

atribuídos a **gerador_teste**; **validation_steps** por sua vez define um número de testes a serem realizados comparando os padrões encontrados na base de treino com os da base de teste. Importante salientar que aqui, apenas para fins de exemplo, estamos executando essa rede neural uma quantidade de vezes relativamente pequena, para melhores resultados você pode realizar testes aumentando tais valores.

```
In [ ]: model.fit_generator(gerador_treino,
   ...:                     steps_per_epoch=100,
   ...:                     epochs=1000,
   ...:                     validation_data=gerador_teste,
   ...:                     validation_steps=50)
Epoch 1/1000
 30/100 [========>.....................] - ETA: 55s - loss: 0.3284 -
accuracy: 0.8714
```

Imagem 07 - Início do processamento da rede neural artificial convolucional

Ao executar este bloco de código a rede começará a ser executada e você pode acompanhar o progresso do processamento da mesma via terminal.

```
model.save_weights('weights.h5')
model.save('model.h5')
```

Apenas realizando uma codificação adicional, uma vez a rede neural executada por completo, podemos salvar o estado da mesma, no que diz respeito aos valores dos pesos encontrados, para que sejam reutilizados. A ideia é que uma vez treinada a rede, poderemos realizar testes com a mesma, sem a necessidade de realizar todo o processo de aprendizado de máquina novamente. Isso é feito simplesmente por meio da função **save_weights( )** e **save( )** passando como parâmetro os nomes dos respectivos arquivos, nesse caso, **‘weights.h5’** e **‘model.h5’**.

```
Epoch 1000/1000
100/100 [==============================] - 101s 1s/step - loss: 0.0932 -
accuracy: 0.9675 - val_loss: 0.0446 - val_accuracy: 0.9803
```

Imagem 08 - Término do processamento da rede neural artificial convolucional

Terminado o processamento da rede neural podemos verificar os resultados e com base nos mesmos, saber se a mesma, de acordo com seus valores percentuais, atingiu uma margem satisfatória de acertos em seu treino (**loss** / **accuracy**) e em suas fases de teste (**val_loss** / **val_accuracy**). Note que inicialmente a rede havia começado com **loss: 0.3284** e **accuracy: 0874**, em outras palavras, uma margem de falsos positivos de 32% e uma margem de acertos de 87%. Ao final do processamento da mesma, **loss: 0.0932** e **accuracy: 0.9675**. Esses números melhoraram consideravelmente, loss inferior a 1% e margem de acertos por volta de 96%, validados com números bastante próximos a isso. Na verdade, esse resultado é muito bom se levarmos em consideração que havíamos uma base de dados tão pequena e que a rede foi executada um número relativamente baixo de vezes, possivelmente teríamos resultados ainda melhores se não fosse por esses dois fatores.

```
img_teste = image.load_img('dataset/amostra_teste/normal.jpeg',
target_size = (150, 150))

img_teste = image.img_to_array(img_teste)
img_teste = np.expand_dims(img_teste, axis = 0)
```

Rede neural devidamente treinada podemos começar a executar testes a partir das mesmas. A finalidade dessa rede em especial é ser a ferramenta onde, ao submetermos uma nova imagem para ser analisada, a mesma será processada e classificada de acordo com os padrões encontrados previamente pela rede.

Nesse contexto, criamos uma variável de nome **img_teste** que por sua vez recebe como atributo a função **load_img( )** parametrizada com o caminho onde temos uma imagem a ser submetida a teste, apenas definimos manualmente um segundo parâmetro chamado **target_size** onde a imagem ao ser carregada será escalonada, independentemente de seu tamanho original, para 150x150 pixels.

Em seguida por meio da função **img_to_array( )** realizamos a conversão da mesma para uma matriz numérica em forma de **array Numpy**. Por fim, chamamos a função **expand_dims( )** passando como parâmetro os dados de **img_teste** e estipulando que a mesma será convertida sobre seu eixo 0, ou seja, a imagem carregada será convertida em uma matriz que será empilhada para que esses dados se tornem aqueles neurônios que alimentarão a camada de entrada da rede.

```
resultado_teste = model.predict(img_teste)
print(resultado_teste)

resultado_final = resultado_teste
print(resultado_final[0,3])

if resultado_final[0,0].any() > 0.5:
    print('Esta radiografia apresenta as estruturas do Tórax sem
alterações significativas..')
elif resultado_final[0,1].any() > 0.5:
    print('Esta radiografia apresenta características compatíveis
com PNEUMONIA')
elif resultado_final[0,2].any() > 0.5:
    print('Esta radiografia apresenta características compatíveis
com DERRAME PLEURAL')
elif resultado_final[0,3].any() > 0.5:
```

```
    print('Esta radiografia apresenta características compatíveis
com TUBERCULOSE')
else:
    print('A imagem fornecida não cumpre os requisitos mínimos
para ser processada,')
    print('ou o resultado do processamento da mesma é
inconclusivo.')
```

```
[[0. 0. 0. 0.]]
Esta radiografia apresenta as estruturas do Torax sem alteracoes
significativas.
```

Na sequência criamos uma variável de nome **resultado_teste** que por sua vez, por meio da função **predict( )** realizará de fato o processamento e teste da imagem carregada. Também por meio da função **print( )** exibimos no terminal qual foi o resultado da classificação (0, 1, 2 ou 3). E para deixar este resultado mais interessante criamos uma simples estrutura condicional onde de acordo com o resultado obtido é exibida uma mensagem de acordo.

Imagem 09 - Resultado de teste exibida em terminal

Nesse caso, submetemos uma imagem que sabíamos ser uma radiografia normal e, como esperado, o processamento da rede encontrou essa característica, exibindo ao final do teste a referente mensagem para este fim.

Concluindo nosso raciocínio, apenas reforçando o que foi dito no início deste capítulo, com uma simples estrutura de rede neural artificial convolucional, baseada na biblioteca Keras, conseguimos criar uma excelente ferramenta que pode servir como um segundo viés de confirmação para os laudos do profissional médico radiologista, contribuindo e muito com a precisão dos diagnósticos,

consequentemente dos tratamentos dos pacientes que indiretamente se beneficiam do uso desta ferramenta.

Código completo disponível em:

https://github.com/fernandofeltrin/Ciencia-de-Dados-e-Aprendizado-de-Maquina/

Documentação da biblioteca Keras (em inglês):

https://keras.io/