PYTHON TOTAL

CONTEÚDO DE 12 LIVROS
MAIS DE 1500 PÁGINAS
TOTALMENTE EXEMPLIFICADO

FERNANDO FELTRIN

PYTHON TOTAL

CONTEÚDO DE 12 LIVROS
MAIS DE 1500 PÁGINAS
TOTALMENTE EXEMPLIFICADO

FERNANDO FELTRIN

PYTHON TOTAL

12 LIVROS 1500+ PÁGINAS

COLETÂNEA
PYTHON
TOTAL

FERNANDO FELTRIN

1.04

2021

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
- Tópicos Especiais em Python
- Arrays + Numpy Com Python
- Ciência de Dados e Aprendizado de Máquina
- Inteligência Artificial com Python
- Redes Neurais Artificiais
- Python + TensorFlow 2.0
- Visão Computacional em Python
- Python na Prática – Aprenda Python Através de Exercícios

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

SOBRE O AUTOR

Fernando Feltrin é Engenheiro da Computação com especializações na área de ciência de dados e inteligência artificial, Professor licenciado para docência de nível técnico e superior, Autor de mais de 10 livros sobre programação de computadores e responsável por desenvolvimento e implementação de ferramentas voltadas a modelos de redes neurais artificiais aplicadas à radiologia (diagnóstico por imagem).

REDES SOCIAIS

https://www.facebook.com/fernandofeltrinpython

https://github.com/fernandofeltrin

Contato direto: fernando2rad@gmail.com

SUMÁRIO

PYTHON TOTAL
AVISOS
SOBRE O AUTOR
REDES SOCIAIS
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
TÓPICOS ESPECIAIS EM PYTHON
Tipagem de Dados
- Tipagem dinâmica vs estática
- Tipo esperado e tipo retornado
- Tipando manualmente o tipo da variável e separadamente o tipo de seu conteúdo.
- Duck Typing
- Tipagem em Comentário
Tipos de Dados Especiais
- Tipo Literal
- Tipo Union
- Tipo Final
- Decorador final
- TypedDict
- Tipo Protocol
Listas
- Métodos inclusos nos built-ins
Strings
- Métodos inclusos nos built-ins
- Particularidades envolvendo o tamanho de uma string
- Tratando os espaços desnecessários de uma string
Dicionários
- Métodos inclusos nos built-ins

- Dicionário ordenado
- DefaultDict
Bool
Sets
- Métodos inclusos nos built-ins
Função print( )
- Quebrando em linhas o conteúdo de uma string parâmetro da função print( )
- Exibindo em tela o nome e o conteúdo de uma variável
Funções
- Parâmetros por Justaposição ou Nomeados
- *args e **kwargs
- Empacotamento e Desempacotamento
- Expressões Lambda
- Parâmetro/argumento somente posicional
- Parâmetro/argumento obrigatoriamente nomeado
- Regrando a obrigatoriedade de tipo de parâmetro da função
f'Strings
- Notação de dicionário
- f'string chamando função do corpo do código
- Operações matemáticas especiais via f'strings
- Saída de múltiplas linhas via print( ) + f'string
- f'string como retorno do método de classe __repr__
Contêineres de Dados
- Iterando sobre contêineres
- Migrando dados de contêineres
Orientação à Objetos
- setattr e getattr
Operador Walrus
Citações em Python
- Citações dentro de strings
Docstrings
- Docstrings de bibliotecas/módulos de terceiros

- Criando uma docstring personalizada
Any e All
- any( )
- all( )
Asserções
Funções Embutidas / Built-ins
- abs( )
- ascii( )
- bytes( )
- callable( )
- eval( )
- exec( )
- format( )
- id( )
- iter( )
- open( )
- slice( )
Funções Matemáticas Específicas
- Distância entre espaço 2D
- Distância entre espaço 3D
- Cálculo de hipotenusa
- Média de valores (números reais)
- Média geométrica
- Valor mais frequente
Metadados
Erros
- SintaxError
- NameError
- TypeError
- IndentationError
- IndexError
- KeyError

- ZeroDivisionError
- FileNotFoundError
- RunTimeError
- OSError
Pip
Gerenciamento de Memória
- Verificando o tempo de execução de um código
Threads
- Execução sequencial Vs paralela
- Single-thread vs multi-thread
- Multi-processing
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

Classificação por Recorrência de Dados – Imdb 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
PYTHON + TENSORFLOW 2.X
TensorFlow
Instalação do TensorFlow 2
CONCEITOS BÁSICOS
Tensores
Constantes e Variáveis
Sessões
PRÁTICA COM TENSORFLOW 2
Ajuda Inline
Operações Básicas com Tensores
Operações Matemáticas com Tensores
Operações Lógicas com Tensores
Strings Como Tensores
Trabalhando com Sessões
REDES NEURAIS ARTIFICIAIS
Classificação Binária
Classificação Multiclasse
ANÁLISE FINANCEIRA COM PYTHON
Visualização de Dados
Rede Neural Artificial Recorrente Aplicada a Séries Temporais
VISÃO COMPUTACIONAL
O que é Visão Computacional?

Processo de Formação de Imagem
Visão Computacional
Algoritmos de Detecção e Rastreamento
Redes Neurais Artificiais Convolucionais
INSTALAÇÃO DAS DEPENDÊNCIAS
Instalação do Python
Instalação do OpenCV
Demais instalações
COMANDOS BÁSICOS DO OPENCV
Ajuda Inline
PROCESSAMENTO DE IMAGENS
Histograma de uma imagem
Equalização de Histograma
TRANSFORMAÇÕES GEOMÉTRICAS EM IMAGENS
Rotacionando uma Imagem
Translação de uma Imagem
Escalonando uma Imagem
Recortando uma Imagem
Espelhando uma Imagem
OPERAÇÕES ARITMÉTICAS EM IMAGENS
Soma de Imagens
Subtração de Imagens
Sobreposição de Imagens
OPERAÇÕES MORFOLÓGICAS
Elementos Estruturantes
Aplicando Erosão
Aplicando Dilatação
Gerando um Gradiente Morfológico
TRATAMENTO DE IMAGENS
Filtro de Suavização por Borramento
Filtro Gaussiano
Filtro Mediano Não-linear

Filtro Manual de Suavização
Filtro de Realce Sobel
Filtro Espacial Laplaciano
Filtro Canny
DETECÇÃO DE OBJETOS
Segmentação Por Binarização
Binarização Adaptativa
Binarização Por OTSU
Transformada de Hough
EXTRAÇÃO DE CARACTERÍSTICAS
Detecção de Cantos
Template Matching
DETECÇÃO DE OBJETOS
Haar Cascades Para Objetos
Detecção de Carros
Detecção de Objetos Personalizados
DETECÇÃO DE FACES
Haar Cascades Para Faces
Detecção a Partir de Imagens
Detecção de Olhos
Detecção a Partir de Webcam
Landmarks de Rostos
Detecção via Dlib + CNN
RASTREAMENTO DE OBJETOS
Rastreamento de Um Objeto Personalizado
Rastreamento de Múltiplos Objetos
Detecção + Rastreamento de um Objeto
Rastreamento de Objetos via Webcam
Rastreando o Deslocamento de Objetos
RECONHECIMENTO DE FACES VIA CNN
REDES NEURAIS ARTIFICIAIS CONVOLUCIONAIS
Reconhecimento de Dígitos Manuscritos

Classificação de Dígitos Manuscritos

Classificação de Imagens de Animais

Classificação de Imagens via TensorFlow

DETECÇÃO DE OBJETOS COM YOLO + DARKNET

EXERCÍCIOS RESOLVIDOS E COMENTADOS

1 - Crie três variáveis com três tipos de dados diferentes, respeitando sua sintaxe:

2 - Crie um comentário de no máximo uma linha:

3 - Crie um comentário de mais de uma linha:

4 - Escreva um programa que mostra em tela a mensagem: Olá Mundo!!!

5 - Crie uma variável nome e atribua para a mesma um nome digitado pelo usuário:

6 - Exiba em tela o valor e o tipo de dado da variável num1: Sendo num1 = 1987.

7 - Peça para que o usuário digite um número, em seguida exiba em tela o número digitado.

8 - Peça para que o usuário digite um número, em seguida o converta para float, exibindo em tala tanto o número em si quanto seu tipo de dado.

9 - Crie uma lista com 5 nomes de pessoas:

10 - Mostre o tamanho da lista nomes / o número de elementos da lista nomes:

Mostre separadamente apenas o terceiro elemento dessa lista:

11 - Some os valores das variáveis num1 e num2: Sendo num1 = 52 e num2 = 106. Por fim exiba em tela o resultado da soma.

12 - Some os valores das variáveis num1 e num2, atribuindo o resultado da soma a uma nova variável homônima. Exiba em tela o conteúdo dessa variável.

13 - Subtraia os valores de num1 e num2:

14 - Realize as operações de multiplicação e de divisão entre os valores das variáveis num1 e num2:

15 - Eleve o valor de num1 a oitava potência, sendo num1 = 51:

16 - Escreva um programa que pede que o usuário dê entrada em dois valores, em seguida, exiba em tela o resultado da soma, subtração, multiplicação e divisão desses números:

17 - Dadas duas variáveis num1 e num2 com valores 100 e 89, respectivamente, verifique se o valor de num1 é maior que o de num2:

18 - Verifique se os valores de num1 e de num2 são iguais:

19 - Verifique se os valores de num1 e de num2 são diferentes:

20 - Verifique se o valor de num1 é igual ou menor que 100:

21 - Verifique se os valores de num1 e de num1 são iguais ou menores que 100.

22 - Verifique se os valores de num1 ou de num2 são iguais ou maiores que 100:

23 - Verifique se o valor de num1 consta nos elementos de lista1. Sendo num1 = 100 e lista1 = [10, 100, 1000, 10000, 100000].

24 - Crie duas variáveis com dois valores numéricos inteiros digitados pelo usuário, caso o valor do primeiro número for maior que o do segundo, exiba em tela uma mensagem de acordo, caso contrário, exiba em tela uma mensagem dizendo que o primeiro valor digitado é menor que o segundo:

25 - Peça para que o usuário digite um número, em seguida exiba em tela uma mensagem dizendo se tal número é PAR ou se é ÍMPAR:

26 - Crie uma variável com valor inicial 0, enquanto o valor dessa variável for igual ou menos que 10, exiba em tela o próprio valor da variável. A cada execução a mesma deve ter seu valor atualizado, incrementado em 1 unidade.

27 - Crie uma estrutura de repetição que percorre a string ‘Nikola Tesla’, exibindo em tela letra por letra desse nome:

28 - Crie uma lista com 8 elementos de uma lista de compras de supermercado, por meio de um laço de repetição for liste individualmente cada um dos itens dessa lista.

29 - Crie um programa que lê um valor de início e um valor de fim, exibindo em tela a contagem dos números dentro desse intervalo.

30 - Crie um programa que realiza a contagem de 0 a 20, exibindo apenas os números pares:

31 - Crie um programa que realiza a Progressão Aritmética de 20 elementos, com primeiro termo e razão definidos pelo usuário:

32 - Crie um programa que exibe em tela a tabuada de um determinado número fornecido pelo usuário:

33 - Crie um programa que realiza a contagem regressiva de 20 segundos:

34 - Crie um programa que realiza a contagem de 1 até 100, usando apenas de números ímpares, ao final do processo exiba em tela quantos números ímpares foram encontrados nesse intervalo, assim como a soma dos mesmos:

35 - Crie um programa que pede ao usuário que o mesmo digite um número qualquer, em seguida retorne se esse número é primo ou não, caso não, retorne também quantas vezes esse número é divisível:

36 - Crie um programa que pede que o usuário digite um nome ou uma frase, verifique se esse conteúdo digitado é um palíndromo ou não, exibindo em tela esse resultado.

37 - Declare uma variável que por sua vez recebe um nome digitado pelo usuário, em seguida, exiba em tela uma mensagem de boas vindas que incorpora o nome anteriormente digitado, fazendo uso de f'strings.

38 - Peça para que o usuário digite um número, diretamente dentro da função print( ) eleve esse número ao quadrado, exibindo o resultado incorporado a uma mensagem:

39 - Dada a seguinte lista: nomes = ['Ana', 'Carlos', 'Daiane', 'Fernando', 'Maria'], substitua o terceiro elemento da lista por 'Jamile':

40 - Adicione o elemento 'Paulo' na lista nomes:

41 - Adicione o elemento 'Eliana' na lista nomes, especificamente na terceira posição da lista:

42 - Remova o elemento 'Carlos' da lista nomes:

43 - Mostre o segundo, terceiro e quarto elemento da lista nomes. Separadamente, mostre apenas o último elemento da lista nomes:

44 – Crie um dicionário via método construtor dict( ), atribuindo para o mesmo ao menos 5 conjuntos de chaves e valores representando objetos e seus respectivos preços:

45 – Crie um dicionário usando o construtor de dicionários do Python, alimente os valores do mesmo com os dados de duas listas:

46 - Crie uma simples estrutura de dados simulando um cadastro para uma loja. Nesse cadastro deve conter informações como nome, idade, sexo, estado civil, nacionalidade, faixa de renda, etc... Exiba em tela tais dados:

47 - Crie um programa que recebe dados de um aluno como nome e suas notas em supostos 3 trimestres de aula, retornando um novo dicionário com o nome do aluno e a média de suas notas:

48 - Crie um sistema de perguntas e respostas que interage com o usuário, pedindo que o mesmo insira uma resposta. Caso a primeira questão esteja correta, exiba em tela uma mensagem de acerto e parta para a próxima pergunta, caso incorreta, exiba uma mensagem de erro e pule para próxima pergunta:

49 - Crie uma função de nome funcao1, que por sua vez não realiza nenhuma ação:

50 - Atribua a função funcao1 a uma variável:

51 - Crie uma função que retorna um valor padrão:

52 - Crie uma função que exibe em tela uma mensagem de boas-vindas:

53 - Crie uma função que recebe um nome como parâmetro e exibe em tela uma mensagem de boas-vindas. O nome deve ser fornecido pelo usuário, incorporado na mensagem de boas-vindas da função:

54 - Crie uma função que recebe um valor digitado pelo usuário e eleva esse valor ao quadrado:

55 - Crie uma função com dois parâmetros relacionados ao nome e sobrenome de uma pessoa, a função deve retornar uma mensagem de boas-vindas e esses dados devem ser digitados pelo usuário:

56 - Crie uma função com dois parâmetros, sendo um deles com um dado/valor predeterminado:

57 - Crie uma função com três parâmetros, sendo dois deles com dados/valores padrão, alterando o terceiro deles contornando o paradigma da justaposição de argumentos:

58 - Crie uma função que pode conter dois ou mais parâmetros, porém sem um número definido e declarado de parâmetros:

59 - Crie uma função de número de parâmetros indefinido, que realiza a soma dos números repassados como parâmetro, independentemente da quantidade de números:

60 - Crie uma função que recebe parâmetros tanto por justaposição (*args) quanto nomeados (**kwargs):

61 - Escreva um programa que retorna o número de Fibonacci: Sendo o número de Fibonacci um valor iniciado em 0 ou em 1 onde cada termo subsequente corresponde à soma dos dois anteriores.

62 - Crie um programa modularizado, onde em um arquivo teremos uma lista de médicos fictícios a serem consultados, em outro arquivo, teremos a estrutura principal do programa, que por sua vez realiza o agendamento de uma consulta médica com base na interação com o usuário.

63 - Aprimore o exemplo anterior, incluindo um módulo simulando o cadastro de usuários em um plano de saúde, apenas permitindo o agendamento de consulta caso o usuário que está interagindo com o programa conste no cadastro:

64 - Crie uma função que recebe parâmetros tanto por justaposição quanto nomeados a partir de uma lista e de um dicionário, desempacotando os elementos e reorganizando os mesmos como parâmetros da função:

65 - Crie uma classe de nome Carro e lhe dê três atributos: nome, ano e cor.

66 - Crie uma classe Pessoa, instancie a mesma por meio de uma variável e crie alguns atributos de classe dando características a essa pessoa. Por fim exiba em tela alguma mensagem que incorpore os atributos de classe criados:

67 - Crie uma classe que armazena algumas características de um carro, em seguida crie dois carros distintos, de características diferentes, usando da classe para construção de seus objetos/variáveis.

68 - Crie uma classe Pessoa com método inicializador e alguns objetos de classe vazios dentro da mesma que representem características de uma pessoa:

69 - Crie uma classe Inventario com os atributos de classe pré-definidos item1 e item2, a serem cadastrados manualmente pelo usuário, simulando um simples carrinho de compras:

70 - Crie uma classe Biblioteca que possui uma estrutura molde básica para cadastro de um livro de acordo com seu título, porém que espera a inclusão de um número não definido de títulos. Em seguida cadastre ao menos 5 livros nessa biblioteca:

71 - Crie uma calculadora simples de 4 operações (soma, subtração, multiplicação e divisão) usando apenas de estrutura de código orientada a objetos:

72 - Mostre via terminal a string 'Bem vindo ao mundo da programação!!!' de trás para frente usando indexação:

73 - Escreva um programa que encontre todos os números que são divisíveis por 7, mas que não são múltiplos de 5, entre 2000 e 2200 (ambos incluídos). Os números obtidos devem ser impressos em sequência, separados por vírgula e em uma única linha:

74 - Escreva um programa, uma calculadora simples de 4 operações, onde o usuário escolherá entre uma das 4 operações (soma, subtração, multiplicação e divisão). Lembrando que o usuário digitará apenas dois valores, e escolherá apenas uma operação matemática do menu.

75 - Crie uma função que recebe um número e retorna o mesmo dividido em duas metades, sendo cada metade um elemento de uma lista:

76 - Crie um programa que gera um dicionário a partir de um valor digitado pelo usuário, de modo que serão exibidos todos valores antecessores a este número multiplicados por eles mesmos. Supondo que o usuário tenha digitado 4, a saída deve ser {1: 1, 2: 4, 3: 9, 4: 16} :

77 - Defina uma função que pode aceitar duas strings como entrada, exibindo em tela apenas a string de maior tamanho/comprimeito. Caso as duas strings tenham mesmo tamanho, exiba em tela as duas:

78 - Escreva um programa que recebe um texto do usuário e o converte para código Morse, exibindo em tela o texto em formato Morse, seguindo a padronização " . - " (ponto, traço).

79 - Escreva um programa que recebe do usuário um número de 0 a 100 e transcreve o mesmo por extenso. Por exemplo o usuário digita 49, então é retornara a string 'quarenta e nove'.

80 - Crie uma função que recebe um nome e um sobrenome do usuário, retornando os mesmos no padrão americano, ou seja, sobrenome primeiro, seguido do nome:

81 - Crie um programa que gera uma senha aleatória, com um tamanho definido pelo usuário:

82 - Crie uma função que exibe em tela tanto o conteúdo de uma variável local quanto de uma variável global, sendo as variáveis de mesmo nome, porém uma não substituindo a outra:

83 – Crie um programa que recebe um número digitado pelo usuário, convertendo o mesmo para algarismo de número Romano, exibindo em tela esse dado já convertido:

84 – Crie um programa que mostra a data atual, formatada para dia-mês-ano, hora:minuto:segundo.

85 – Crie um programa que exibe a versão atual do Python instalada no sistema:

86 – A partir de uma lista composta apenas de dados numéricos, gere outra lista usando de list comprehension usando como base a lista anterior, compondo a nova com os valores dos elementos originais elevados ao cubo:

87 – Dada uma estrutura inicial de um carrinho de compras com 5 itens e seus respectivos valores, assim como uma função que soma os valores dos itens do carrinho, retornando um valor total. Aprimore a função deste código usando de list comprehension:

88 – Escreva uma função que recebe do usuário um número qualquer e retorna para o mesmo tal número elevado ao quadrado. Crie uma documentação para essa função que possa ser acessada pelo usuário diretamente via IDE.

89 – Escreva um programa que recebe uma frase qualquer, mapeando a mesma e retornando ao usuário cada palavra com a frequência com que a mesma aparece na frase em questão.

90 – Crie um programa que recebe do usuário uma sequência de números aleatórios separados por vírgula, armazene os números um a um, em formato de texto, como elementos ordenados de uma lista. Mostre em tela a lista com seus respectivos elementos após ordenados.

91 – Escreva um programa, da forma mais reduzida o possível, que recebe do usuário uma série de nomes, separando os mesmos e os organizando em ordem alfabética. Em seguida, exiba em tela os nomes já ordenados.

92 – Escreva um simples programa que recebe do usuário um número qualquer, retornando ao mesmo se este número digitado é um número perfeito.

93 – Escreva uma função que recebe uma lista de elementos totalmente aleatórios e os ordena de forma crescente de acordo com seu valor numérico:

94 – Crie uma estrutura toda orientada a objetos que recebe do usuário uma string qualquer, retornando a mesma com todas suas letras convertidas para letra

maiúscula. Os métodos de classe para cada funcionalidade devem ser independentes entre si, porém trabalhar no escopo geral da classe.

95 – Escreva de forma reduzida um programa que recebe do usuário um nome e duas notas, salvando tais dados como um elemento de uma lista. Exiba em tela o conteúdo dessa lista.

96 – Crie um programa que gera o valor de salário de funcionários considerando apenas horas trabalhadas e horas extras, sendo o valor fixo da hora trabalhada R$29,11 e da hora extra R$5,00. Crie uma regra onde o funcionário só tem direito a receber horas extras a partir de 40 horas trabalhadas da forma convencional.

97 – Reescreva o código anterior adicionando um mecanismo simples de validação que verifica se os dados inseridos pelo usuário são de tipos numéricos, caso não sejam, que o processo seja encerrado.

98 – Crie um programa que recebe uma nota entre 0 e 1.0, classificando de acordo com a nota se um aluno fictício está aprovado ou em recuperação de acordo com sua nota. A média para aprovação deve ser 0.6 ou mais, e o programa deve realizar as devidas validações para caso o usuário digite a nota em um formato inválido.

99 – Crie uma estrutura molde (orientada a objetos) para cadastro de veículos, tendo como características que os descrevem sua marca, modelo, ano, cor e valor. Cadastre ao menos três veículos, revelando seu número identificador de objeto alocado em memória, assim como o retorno esperado pelo usuário quando o mesmo consultar tal veículo.

100 – Crie um programa que recebe do usuário três números diferentes, retornando para o mesmo qual destes números é o de maior valor. Usar de funções aninhadas para realizar as devidas comparações entre os valores dos números:

101 – Crie um programa que atua como mecanismo controlador de um sistema direcional, registrando a direção e o número de passos executado. Ao final do processo, exiba em tela o número de referência para um plano cartesiano. Ex: Se o usuário digitar CIMA 5, BAIXO 3, ESQUERDA 3 e DIREITA 2, o resultado será a coordenada 2.

102 – Crie uma estrutura orientada a objetos híbrida, para cadastro de usuários, que permita ser instanciada por uma variável de modo que a mesma pode ou não repassar argumentos para seus objetos de classe na hora em que instancia tal classe. Para isso, a classe deve conter atributos/objetos de classe padrão e instanciáveis de fora da classe.

103 – Gere uma lista com 50 elementos ordenados de forma crescente. Inicialmente usando do método convencional (e do construtor de listas padrão), posteriormente reescrevendo o código usando de list comprehension.

104 – Reescreva a estrutura de código abaixo, de um simples programa que gera uma lista com as possíveis combinações entre duas outras listas [1,2,3] e [4,5,6], reduzindo e otimizando a mesma via list comprehension:

CONSIDERAÇÕES FINAIS

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.

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 vasta gama 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 de nossa “informática”, um dos nichos diferenciais é a área da programação, tanto pela sua complexidade quanto por sua enorme 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, seja ela em linguagens generalistas ou especialistas. No âmbito acadêmico existem diversos cursos, que vão de análise e desenvolvimento de sistemas até engenharia da computação, e observando a 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 a C ou uma de suas vertentes, isso se dá pelo fato de C ser uma linguagem ainda hoje bastante popular (se não a mais popular em aplicações), também podendo servir de base para muitas 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 de aplicações específicas. Poderíamos dedicar um capítulo inteiro mostrando apenas as principais e suas características, mas ao invés disso vamos nos focar logo em Python.

Hoje com tecnologias como blockchain, internet das coisas, data science, machine learning, visão computacional, além é claro da criação de aplicações web, aplicativos mobile, softwares, jogos e sistemas comerciais, mais do que nunca foi preciso profissionais da área que possuam ao menos noções de 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 graças a suas ferramentas e capacidade de se adaptar para qualquer propósito.

Python pode ser a sua linguagem de programação definitiva, ou abrir muitas portas para aprender outras mais, já que nesta área 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, linha por linha, 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, pois haverá uma sintaxe e uma sequência lógica de argumentos a se respeitar a fim de que seus comandos façam sentido e sejam interpretados pelo núcleo da linguagem, assim como em uma linguagem natural e escrita, organizamos sílabas e fonemas de modo a reproduzir ideias a serem compreendidas por outra pessoa.

Com certeza quando você iniciou seu processo de alfabetização, começou a aprender letras, sílabas, palavras, regras gramaticais, etc... tudo para usar da linguagem uma ferramenta de comunicação e interação com o mundo ao redor. Ao buscar aprender outro idioma, inglês por exemplo, também começou pelo básico de sua gramática, o verbo to be, e posteriormente foi incrementando com novos conceitos e vocabulário, até ter certo domínio sobre tal língua.

Em uma linguagem de programação não é muito diferente, há uma sintaxe, equivalente a gramática, 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 para executar uma determinada ação ou gerar um 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 nossos 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 de escrita, e seu potencial de fazer mais com menos no que diz respeito a suas ferramentas nativas. Não desmerecendo outras linguagens de programação, mas é fato que Python foi desenvolvida para ser descomplicada e ao mesmo tempo suprir necessidades de resolução de problemas computacionais com facilidade e eficiência.

Aqui cabe facilmente a pergunta:

- E por quê fazer as coisas da forma mais difícil se existem ferramentas para torná-las mais fáceis?

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

“- A vida é curta demais para programar em outra linguagem de programação, 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

```
1   #include <stdio.h>
2
3   int main(void) {
4     printf("Hello World\n");
5     return 0;
6   }
7
```

Hello World!!! em JAVA

```
1   class Main {
2     public static void main(String[] args) {
3       System.out.println("Hello world!");
4     }
5   }
6
```

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 sobre outras linguagens 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 você conhecerá este padrão irá se repetir. Em Python será muito mais fácil, mais rápido, e com menos linhas de código para se atingir o objetivo final.

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.

Python preza por uma 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 algum pequeno erro de escrita (um ponto ou vírgula que ficou sobrando ou faltando no código), tipo de situação que não ocorre em Python.

Python por ser uma linguagem interpretada deveria sofrer ainda mais com esses problemas, mas o que ocorre é justamente o contrário, o interpretador foca nos comandos e seus parâmetros, os pequenos erros de sintaxe não farão com que o código não funcione, apenas serão emitidos alertas para que em tempo real se corrija o(s) erro(s).

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 processando volumes enormes de dados.

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

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 da linguagem C, de modo que 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 considerável comunidade de entusiastas dedicada a aprimorá-la. Somente em 1994 foi lançada sua versão 1.0, ou seja, sua primeira versão totalmente funcional, 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 para a mesma tornando-a cada vez 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 graças a seus feitos e contribuições para a comunidade.

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 pelo seu núcleo de desenvolvedores quanto pela comunidade, e ainda hoje Guido continua supervisionando o desenvolvimento da linguagem Python, tomando certas 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 e esforços na linguagem Python. Em suma, programar em Python deve ser simples, de fácil aprendizado, com código de fácil leitura, enxuto, mas inteligível, eficiente e 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 ou a linguagem R, específica para ciência de dados. Já linguagens generalistas são aquelas que tem sua aplicabilidade em todo e qualquer propósito, e nem por isso ser inferior às específicas, apenas possuindo recursos e ferramentas facilmente adaptáveis a qualquer implementação.

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, graças a sua versatilidade, é possível programar para qualquer sistema operacional, programação web, mobile, data science, machine learning, blockchain, visão computacional, etc... coisa que outras linguagens de forma nativa não são suficientes ou ao menos práticas para o programador, necessitando uma série de gambiarras para realizar sua codificação, tornando o processo mais difícil ou menos eficiente.

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 incrível animações e filmes compostos de 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 e suas bibliotecas.

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, assim como computação distribuída em sistemas como blockchain, 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, muito pelo contrário, uma vez que todas tecnologias citadas acima podem ser facilmente implementadas via Python.

A versão 3 do Python é por si só bastante robusta e consegue de forma nativa trabalhar com tais tecnologias, haja visto que devido a enorme comunidade que desenvolve ou aprimora ferramentas para Python, podemos esperar que para o futuro certamente haverão novas versões da linguagem implementando novos recursos de forma nativa. Será muito difícil vermos surgir outra linguagem “do zero” ou que tenha usabilidade parecida com Python, uma vez que a mesma é bastante consolidada no mercado graças a sua robustez.

Se você olhar para trás no tempo, verá uma série de linguagens que foram descontinuadas ou que simplesmente foram perdendo seu espaço, mesmo seus desenvolvedores insistindo e injetando tempo e dinheiro em seu desenvolvimento, muitas delas não eram modernas o suficiente ou não acompanharam a rápida evolução deste nicho da computação. Do meu ponto de vista não consigo ver um cenário do futuro ao qual Python não consiga se adaptar ou até mesmo cenário onde Python não seja destaque entre as demais.

Entenda que 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, coisa que Python nos proporciona.

Num futuro próximo nosso diferencial como profissionais da área de tecnologia será ter conhecimento sobre linguagens de programação que se adaptem as necessidades do mercado. 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 (interface que interage com o usuário final).

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, basicamente 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 linguísticos e conceitos normalmente vindos do inglês e que o usuário pode pegar qualquer bloco de código de modo que 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 próprio programados. Em outras palavras, enquanto em alto nível temos códigos que abstraem ideias de forma legível, em baixo nível o código em execução terá instruções para o processamento de dados alocados em memória e em threads no próprio processador.

Quando estamos programando em Python sempre estaremos num ambiente de linguagem de alto nível, onde usaremos expressões em inglês e uma sintaxe fácil para por fim dar nossas instruções ao computador de que dados deverão ser processados ou de que ações deverão serem executadas a partir de nosso código. Esta linguagem posteriormente será convertida para camadas de linguagem de baixo nível que por fim se tornarão sequências de instruções para registradores, portas lógicas ou até mesmo chaveamentos de circuitos dependendo a usabilidade.

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-o 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.

Já na era da computação, 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 de um 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 em nosso sistema operacional temos a frontend do sistema, ou seja, a camada 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 tendo que emular estruturas de acesso ao hardware, assim como se programava separado de onde se compilava, separado de onde se debugava e por fim separado de onde se rodava o programa finalmente vendo o ver usando dos recursos do sistema operacional e executando suas devidas funcionalidades. Apenas entenda que as IDEs unificam todas camadas necessárias emulando tudo o que é necessário para que possamos nos concentrar em apenas escrever nossos códigos e executá-los ao final desse processo.

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

Principais IDEs

Como mencionado no tópico anterior, IDEs buscam unificar as ferramentas necessárias para facilitar a vida do programador, claro que é possível programar a partir de qualquer editor de texto, 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 para fins de estudo, sendo bastante intuitiva e fácil de aprender a usar seus recursos, por fim, ela oferece um ambiente com suporte em tempo real ao uso de console/terminal próprio, assim como capacidade de gerir ambientes virtualizados isolados para cada projeto, sendo assim possível a qualquer momento executar testes nos seus blocos de código independentemente se os mesmos usam apenas recursos nativos da plataforma ou se usam de bibliotecas instaladas exclusivamente para o projeto.

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, como já mencionado diversas vezes, é uma linguagem de programação 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 de, é claro, do próprio ambiente integrado de programação padrão, que neste caso é o VisualStudioCode, entre outras ferramentas todas pré-configuradas para interação entre si.

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

Importante salientar também que, é perfeitamente normal e possível programar direto a partir de um terminal, independentemente do sistema operacional, ou em terminais oferecidos pelas próprias IDEs, a escolha de usar um ou outro é de particularidade do próprio programador.

Como dito anteriormente, existem diversas IDEs e ferramentas que facilitarão sua vida como programador, a verdade é que é interessante você dedicar um tempinho a testar algumas IDEs e ver qual você se adapta melhor tanto pela facilidade de uso quanto pelas ferramentas disponibilizadas de acordo com sua implementação.

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 chamamos de linguagem os meios que criamos para conseguir passar comandos a serem interpretados e executados em um computador.

Quando falamos sobre um determinado assunto, automaticamente em nossa língua falada nativa geramos uma sentença que possui uma certa sintaxe gramatical 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 corretamente realizado.

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

Algoritmos

Todo estudante de computação no início de seu curso recebe uma boa carga horária de algoritmos. Algoritmos em suma nada mais são do que uma sequência de passos ordenados 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 e em que quantidade 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 funcionalidade, temos que passar as instruções passo a passo para que o interpretador consiga as executar e chegar ao fim de um determinado 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 toda a mistura já pronta de ingredientes, bastando apenas adicionar leite e colocar para assar (neste caso, não temos as entradas de forma explícita, 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 acondicionados em um invólucro todo o necessário para produzir o bolo, 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 (de forma explícita ou implícita para o usuário) de acordo com as entradas necessárias e as saídas esperadas para resolver algum tipo de problema computacional específico.

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:

```
1  print('5' + '2')
2
52
```

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:

```
1  print('Seja bem vindo!')
2  print('Você é' + 3 * ' muito ' + 'legal!')
3
```

```
Seja bem vindo!
Você é muito  muito  muito legal!
```

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:

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

```
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:

```
1  print('Olá Amigo')
2  print('Tudo bem?')
3
```

```
Olá Amigo
Tudo bem?
```

Código com vício de linguagem:

```
1  print('Olá Amigo');
2  print('Tudo bem?')
3
```

```
Olá Amigo
Tudo bem?
```

Código extrapolado:

```
1  print('Olá Amigo'); print('Tudo bem?')
2
```

```
Olá Amigo
Tudo bem?
```

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:

```
1  print('Olá Mundo!!!')
```

```
Olá Mundo!!!
```

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!!!"""
```

```
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 nº {ap}')
5
```

```
Digite o seu nome: Fernando
Digite o número do seu apartamento: 134
Seja bem vindo(a) Fernando, morador(a) do ap nº 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:

```python
num1 = int(5)
num2 = int(8.2)

print(f'O resultado da soma é: {num1 + num2}')

```

```
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 variável 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:

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

```
1
1
```

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

O resultado será 1.

Atribuição Multiplicativa:

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

```
8
8
```

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

O resultado será: 8

Atribuição Divisiva:

```
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
```

```
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):

```python
variavel1 = 4
resultado = variavel1 % 4

print(resultado)

# Mesmo que:
variavel2 = 4
variavel2 %= 4

print(variavel2)

```

```
0
0
```

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

O resultado será: 0

Exponenciação:

```python
variavel1 = 4
resultado = variavel1 ** 8

print(resultado)

# Mesmo que:
variavel2 = 4
variavel2 **= 8

print(variavel2)

```

```
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:

```
1   variavel1 = 512
2   resultado = variavel1 // 256
3
4   print(resultado)
5
6   # Mesmo que:
7   variavel2 = 512
8   variavel2 //= 256
9
10  print(variavel2)
11
```

```
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 %.

```
1  print(10 % 3)
2
```

```
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:

```
1  numero1 = 12
2  numero2 = 3
3
4  print(numero1 + numero2)
5
6  # Mesmo que:
7  print('O resultado da soma é:', numero1 + numero2)
8
9  # Que pode ser aprimorado para:
10 print(f'O resultado da soma é: {numero1 + numero2}')
11
```

```
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:

```python
x = 5
y = 8
z = 13.2

print(x + y)
print(x - y)
print(x ** z)
print(z // y)
print(z / y)

```

```
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:

Global frame
Cor
cor_atual
2
Cor class
hide attributes
azul
3
branco
4
preto
5
verde
2
vermelho
1

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:

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

```
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

7

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\Exercici
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

```
Global frame
frase1   "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.

```
1  fernando = {'nome': 'Fernando Feltrin',
2              'idade': 31,
3              'formacao': ['Engenheiro da Computação',
4                           'Técnico em TI']}
5
```

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:

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

```
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:

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

```
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:

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

```
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:

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

```
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:

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

```
{'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:

```
c1 = {1, 2}
c2 = {2, 3}

c1.union(c2)
#c1 união com c2, matematicamente juntar 2 conjuntos.

{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:

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

```
Digite o seu nome:
```

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:

```
1  nome = 'Maria'
2  idade = 30
3
4  print(f'Nome: {nome}, Idade: {idade}')
5
6  print(f'{nome} tem 3 vezes minha idade, ela tem {3 * idade} anos')
7
```

```
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:

Global frame
Usuario
usuario1
__init__
self
nome
"Fernando"
idade
"31"
Return value
None
boas_vindas
self
Return value
None
Usuario class
hide attributes
__init__
function
__init__(self, nome, idade)
boas_vindas
function
boas_vindas(self)
Usuario instance
idade
"31"
nome
"Fernando"

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 2
Return value None
imprimir
maximo 10
atual 3
Return value None
imprimir
maximo 10
atual 4
Return value None
imprimir
maximo 10
atual 5
Return value None
imprimir
maximo 10
atual 6
Return value None
imprimir
maximo 10
atual 7
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.

```
1  class Pessoa:
2      nome = 'Fernando'
3
4  pessoa1 = Pessoa()
5
6  print(pessoa1.nome)
7
```

```
Fernando
```

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.

```python
class Pessoa:
    nome = 'Padrão'

pessoa1 = Pessoa()
pessoa2 = Pessoa()
pessoa3 = Pessoa()

Pessoa.nome = 'Fernando'

print(pessoa1.nome)
print(pessoa2.nome)
print(pessoa3.nome)

```

```
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( ).

```
1  class Pessoa:
2    def acao1(self):
3      print('Ação 1 sendo executada...')
4
5  pessoa1 = Pessoa()
6
7  print(pessoa1.acao1())
8
```

```
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.

Frames
Objects
Global frame
Pessoa
pessoa1
Pessoa class
hide attributes
acao1
function
acao1(self)
acao1
self
Return value
None
Pessoa instance

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.

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.

```
1  class Pessoa:
2      nome = 'Padrão'
3
4      def __init__(self):
5          self.nome = 'Administrador'
6
7  usuario1 = Pessoa()
8
9  print(usuario1.nome)
10 print(Pessoa.nome)
11
```

```
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.

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.

```
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           if self.login:
9               print(f'{self.nome} já está logado no sistema')
10              return
11
12          print(f'Bem vindo {self.nome}, você está logado no sistema.')
13          self.login = True
14
15  usuario = Pessoa('Fernando')
16  usuario.logar()
17  usuario.logar()
18
```

```
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.

```python
from random import randint

class Pessoa:
    @staticmethod
    def gerador_id():
        gerador = randint(100, 999)
        return gerador

print(Pessoa.gerador_id())

```

```
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.

```
1   class Produto:
2       def __init__(self, nome, preco):
3           self.nome = nome
4           self.preco = preco
5
6       @property #Getter
7       def preco(self):
8           return self.preco_valido
9
10      @preco.setter #Setter
11      def preco(self, valor):
12          if isinstance(valor, str):
13              valor = float(valor.replace('R$',''))
14          self.preco_valido = valor
15
16      def desconto(self, percentual):
17          self.preco = self.preco - (self.preco*(percentual/100))
18
19  produto1 = Produto('Processador', 370)
20  produto1.desconto(15)
21
22  produto2 = Produto('Placa Mãe', 'R$280')
23  produto2.desconto(20)
24
25  print(produto1.preco)
26  print(produto2.preco)
27
```

```
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.

```
    self.dados = {}

base = BaseDeDados()
print(base.)
```

dados
mro
__base__
__bases__
__class__
__delattr__
__dict__
__dir__
__eq__
__format__
__ge__
__getattribute__

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.

```
    self._dados = {}

base = BaseDeDados()
print(base.)
```

```
mro
__base__
__bases__
__class__
__delattr__
__dict__
__dir__
__eq__
__format__
__ge__
__getattribute__
__gt__
```

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.

```
    self.__dados = {}

base = BaseDeDados()
print(base.)
```

```
mro
__base__
__bases__
__class__
__delattr__
__dict__
__dir__
__eq__
__format__
__ge__
__getattribute__
__gt__
```

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.

```
print(relClientes.)
                  apagar
                  base
                  inserir
                  listar
```

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.

```
print(relClientes.)
                  apagar
                  inserir
                  listar
```

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.

```
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.

```
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
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.

```
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
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
24  usuario1 = Usuario('Fernando')
25  identificador1 = Identificador('0001')
26
27  usuario1.logar = identificador1
28  usuario1.logar.logar()
29
```

```
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.

```
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
18  cliente1 = Cliente('Fernando', 32)
19  cliente1.addFone(33221766, 991357258)
20  print(cliente1.nome)
21  print(cliente1.listaFone())
22
```

```
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.

```
1   class Carro:
2       def __init__(self, nome, ano):
3           self.nome = nome
4           self.ano = ano
5
6   class Corsa(Carro):
7       pass
8
9   class Gol(Carro):
10      pass
11
```

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:

```
1  class Jeep:
2      def carro():
3          def __init__(self, nome, ano):
4              self.nome = nome
5              self.ano = ano
6
7      def gasolina(self, tipogasolina=True, tipoalcool=False):
8          self.tipogasolina = tipogasolina
9          self.tipoalcool = tipoalcool
10
11 jeep = Jeep()
12
```

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.

```
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
14 carro1 = Jeep()
15 print(carro1.tipogasolina)
16
17
```

```
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.

```python
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
```

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.

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. 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.

```
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
15  p1 = SaveJogador1('Fernando')
16  print(p1.nome)
17
```

```
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

```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')

```

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.

```
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          super().Acao1()
15          print(f'{self.nome} está acordado')
16
17  p1 = SaveJogador1('Fernando')
18  p1.Acao1()
19  p1.Acao2()
20  class SaveJogador1(Jogador1):
21      def Acao1(self):
22          super().Acao1()
23          print(f'{self.nome} está acordado')
24
```

```
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:

```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(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

```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()

```

```
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:

```
1   from abc import ABC, abstractclassmethod
2
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( ).

```
1   from abc import ABC, abstractclassmethod
2
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
12  user1 = Pessoa()
13  user1.logar()
14
```

```
---------------------------------------------------------------------------
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 à 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 um 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
__init__
self
largura 10
altura 10
Return value None
__init__
self
largura 10
altura 20
__init__
self
largura 10
altura 20
Return value None
__add__
self
other
largura1 20
altura1 30
Return value
__init__
self
largura 20
altura 30
Return value None
Caixa class
hide attributes
__add__ function __add__(self, other)
__init__ function __init__(self, largura, altura)
__repr__ function __repr__(self)
Caixa instance
<class 'Caixa(10, 10)'>
altura 10
largura 10
Caixa instance
<class 'Caixa(10, 20)'>
altura 20
largura 10
Caixa instance
<class 'Caixa(10, 20)'>
altura 20
largura 10
Caixa instance
<class 'Caixa(20, 30)'>
altura 30
largura 20

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 1 2 3 4
33 1987 2020 19.9 100000000
dict
"Nome" "Fernando"
"Profissão" "Engenheiro"
function
identificacao(*args, **kwargs)
tuple
0 1 2 3 4
33 1987 2020 19.9 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.

```
fator = fatorial(12)
print(fator)

```

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:

```
[ ]
def fatorial(num: int) -> int:
    if num == 1:
        return 1

    return num * fatorial(num - 1)

fator = fatorial(12)
print(fator)

```

```
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.

```
fator = fatorial(500)
print(fator)

1220136825991110068701238785423046926253574342803192842192413588 3
```

Apenas como exemplo, fatorando o número 500 é gerado um retorno com mais de 1000 dígitos.

```
fator = fatorial(1000)
print(fator)

---------------------------------------------------------------------------
RecursionError                            Traceback (most recent call
<ipython-input-3-85bcf81c3d0e> in <module>()
      5   return num * fatorial(num - 1)
      6
----> 7 fator = fatorial(1000)
      8 print(fator)

                              1 frames
... last 1 frames repeated, from the frame below ...

<ipython-input-3-85bcf81c3d0e> in fatorial(num)
      3     return 1
      4
----> 5   return num * fatorial(num - 1)
      6
      7 fator = fatorial(1000)

RecursionError: maximum recursion depth exceeded in comparison

SEARCH STACK OVERFLOW
```

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 metacaractere . 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.

```
4  print(re.findall(r'.oão|.aria', texto1))
5  print(re.findall(r'.oão|.aria|mil', texto1))
6
```

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.

```
1  print(re.findall(r'[Jj]oão|.aria', texto1))
2
```

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 = 'Nããããããããããããoooooooo 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 'Nããããããããããããoooooooo'.

```
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 ‘Nãããããããããoooooo’ 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']
['Nããããããããããão']
```

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']
['Nããããããããããão']
['Nãããããããããããoooooooo']
```

O resultado, como esperado, é finalmente toda a string ‘Nãããããããããoooooooo’, 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']
['Nãããããããããããoooooo']
['Nãããããããããããoooooo', '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']
['Nãããããããããããoooooo']
['Nãããããããããããoooooo', 'não']
['Nãããããããããããoooooo', '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 = '0123456789012345678901234567890 12'
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.

```
1   carrinho = []
2   carrinho.append(('Item 1', 30))
3   carrinho.append(('Item 2', 45))
4   carrinho.append(('Item 3', 22))
5   total = 0
6
7   for produto in carrinho:
8     total = total + produto[1]
9
10  print('Método tradicional', total)
11
```

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.

```
1   total2 = []
2   for produto in carrinho:
3     total2.append(produto[1])
4   print('Usando funções internas', sum(total2))
5
```

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
tuple
"Item 1"
30
tuple
"Item 2"
45
list
30
45
22
<listcomp>
.0
x "Item 3"
y 22
Return value
tuple
"Item 3"
22
list_iterator instance
list
30
45
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.

```
minha_lista = [1,2,3,4]
minha_lista = iter(minha_lista)

```

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)
10
```

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.

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.

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

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

```
1   produtos = [
2               {'nome': 'item 1', 'preco': 32},
3               {'nome': 'item 2', 'preco': 23},
4               {'nome': 'item 3', 'preco': 12},
5               {'nome': 'item 4', 'preco': 10},
6               {'nome': 'item 5', 'preco': 55},
7   ]
8
9   precos = map(lambda p: p['preco'], produtos)
10
11  for preco in precos:
12    print(preco)
13
```

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.

```
32
23
12
10
55
```

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
dict
"nome"
"item 1"
"preco"
32
dict
"nome"
"item 2"
"preco"
23
dict
"nome"
"item 3"
"preco"
12
dict
"nome"
"item 4"
"preco"
10
dict
"nome"
"item 5"
"preco"
55
λ
p
Return value
32
λ
p
Return value
23
λ
p
Return value
12
λ
p
Return value
10
λ
p
Return value
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}
```

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

```
1   produtos2 = [
2               {'nome': 'item 1', 'preco': 32},
3               {'nome': 'item 2', 'preco': 23},
4               {'nome': 'item 3', 'preco': 62},
5               {'nome': 'item 4', 'preco': 10},
6               {'nome': 'item 5', 'preco': 55},
7   ]
8
9   def filtra(p):
10    if p['preco'] > 50:
11      return True
12
13  nova_lista3 = filter(filtra, produtos2)
14  for produtox in nova_lista3:
15    print(produtox)
16
```

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]
```

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
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
False
λ
p
Return value
False
λ
p
Return value
True
λ
p
Return value
True
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]
```

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

```
52  # Extraindo dados via Reduce + List Comprehension
53  from functools import reduce
54  precos_total = reduce(lambda soma, valores: soma + valores['Preco'], estoque, 0)
55  print(f'Soma dos Preços (Reduce + List Comprehension) {precos_total}')
56
```

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.

```
Preços de estoque (normal) [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]
Preços de estoque (List Comprehension) [39.9, 37.9, 79.9, 69.9, 59
Preços de estoque (Filter) [{'Item03': 'Moletom 00', 'Preco': 79.9
Preços de estoque (Filter + List Comprehension) [79.9, 69.9]
Preços de estoque (Map + Filter) [39.9, 37.9, 79.9, 69.9, 59.9]
Soma dos Preços (Reduce) 287.5
Soma dos Preços (Reduce + List Comprehension) 287.5
Soma dos Preços (Reduce Método Otimizado) 287.5
```

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.

```
PS C:\Users\Fernando> & "C:/Program Files/Python38/python.exe"
name 'a' is not defined
PS C:\Users\Fernando>
Python 3.8.1 64-bit  ⊗ 1 ⚠ 0
```

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.

```
1  except NameError as erro:
2    print('Ocorreu um erro:',erro)
3
```

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.

```
1  def conversor_num(num):
2    try:
3      num = int(num)
4      return num
5    except ValueError:
6      try:
7        num = float(num)
8        return num
9      except ValueError:
10       pass
11
```

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.

```
Digite um número: f
Você digitou um caractere que não pode ser convertido
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-5-d1ad55ce221e> in conversor_num(num)
      4   try:
----> 5     num = int(num)
      6     return num

ValueError: invalid literal for int() with base 10: 'f'

During handling of the above exception, another exception occurred:

ValueError                                Traceback (most recent call last)
 ---------------------------------------- 1 frames ----------------------------------------
<ipython-input-5-d1ad55ce221e> in conversor_num(num)
      6     return num
      7   except ValueError:
----> 8       num = float(num)
      9       return num
     10   finally:

ValueError: could not convert string to float: 'f'
```

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.

```
'_itertools',
'_log',
'_pi',
'_random',
'_sha512',
'_sin',
'_sqrt',
'_test',
'_test_generator',
'_urandom',
'_warn',
'betavariate',
'choice',
'choices',
'expovariate',
'gammavariate',
'gauss',
'getrandbits',
'getstate',
'lognormvariate',
'normalvariate',
'paretovariate',
'randint',
'random',
'randrange',
'sample',
```

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.

À 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 daquele 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.

```
EXPLORER ···   main.py ×   soma.py
> OPEN EDITORS
v MODULARIZAÇÃO
  main.py
  soma.py
main.py > ...
1  import soma
2
3  print(f'O resultado da soma é: {soma.soma(8,3)}')
4
```

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.

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')
```

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 primeiras 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.

Print output (drag lower right corner to resize)
32
Frames
Objects
Global frame
Pessoa
pessoa2
ano_nasc
cls
nome "Fernando"
ano_nascimento 1987
idade 32
Return value
__init__
self
nome "Fernando"
idade 32
Return value None
Pessoa class
hide attributes
__init__ function __init__(self, nome, idade)
ano_atual 2019
ano_nasc classmethod instance
Pessoa instance
idade 32
nome "Fernando"

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.

```
1   from random import randint
2
3   class Usuario:
4     @staticmethod
5     def gerador_id():
6       gerador = randint(100, 999)
7       return gerador
8
9   print(Usuario.gerador_id())
10
```

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
__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
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"

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:

```
1  class Jeep:
2      def carro():
3          def __init__(self, nome, ano):
4              self.nome = nome
5              self.ano = ano
6
7      def gasolina(self, tipogasolina=True, tipoalcool=False):
8          self.tipogasolina = tipogasolina
9          self.tipoalcool = tipoalcool
10
11 jeep = Jeep()
12
```

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.

```
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
14 carro1 = Jeep()
15 print(carro1.tipogasolina)
16
```

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.

```
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.

```
17 p1 = SaveJogador1('Fernando')
18 p1.Acao1()
19 p1.Acao2()
20
```

```
Fernando está acordado
Fernando está comendo
```

Finalmente temos como retorno o resultado apresentado na imagem acima.

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.

```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(Acao1, Acao2):
    pass

p1 = SaveJogador1()
p1.acao()

```

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.

```
12  #user1 = Pessoa()
13  #user1.logar()
14
15  user1 = Usuario()
16  user1.logar()
17
```

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.

```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”.

```
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

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 4 por exemplo, é automaticamente interpretada 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
__init__
self
largura 10
altura 10
Return value None
__init__
self
largura 10
altura 20
__init__
self
largura 10
altura 20
Return value None
__add__
self
other
largura1 20
altura1 30
Return value
__init__
self
largura 20
altura 30
Return value None
Caixa class
hide attributes
__add__ function __add__(self, other)
__init__ function __init__(self, largura, altura)
__repr__ function __repr__(self)
Caixa instance
<class 'Caixa(10, 10)'>
altura 10
largura 10
Caixa instance
<class 'Caixa(10, 20)'>
altura 20
largura 10
Caixa instance
<class 'Caixa(10, 20)'>
altura 20
largura 10
Caixa instance
<class 'Caixa(20, 30)'>
altura 30
largura 20

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.

TÓPICOS ESPECIAIS EM PYTHON

Tipagem de Dados

Quando estamos falando especificamente dos tipos de dados suportados pela linguagem Python temos uma vasta gama, definidos de acordo com seu propósito quando o mesmo for usado por uma estrutura de código, desde guardar dados/valores até realizar determinadas ações no código.

Cada tipo de dado possui uma sintaxe bastante específica que torna fácil a declaração e a identificação visual do mesmo, uma vez que uma das propostas base da linguagem Python é prezar pela simplicidade e legibilidade do código.

Uma particularidade da linguagem Python em relação a muitas outras de alto nível é a tipagem de seus dados, no que diz respeito a forma com que dados podem ter um tipo específico, mas não fixo.

Em outras palavras, sempre que necessário é possível alterar o tipo de um dado, seja alterando sua sintaxe, seja criando máscaras e modificadores que o farão serem reconhecidos e forma diferente pelo interpretador, assim como mudar seus comportamentos de acordo com o tipo de dado definido por último.

- Tipagem dinâmica vs estática

Dentro da lógica referenciada anteriormente, podemos manipular o tipo de dado em questão de uma variável/objeto simplesmente alterando sua sintaxe.

```
1   numero = 1987
2
3   print(type(numero))
4
5   numero = '1987'
6
7   print(type(numero))
8
```

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

Declarada a variável numero, atribuindo para a mesma o valor 1987, nesta forma escrita, o interpretador automaticamente reconhece este dado como do tipo numérico, sem casas decimais, logo aqui temos uma variável do tipo int.

Na linha 5 do código podemos ver a mesma variável numero sendo novamente declarada, agora possuindo como atributo o dado ‘1987’, que pela sintaxe, automaticamente será identificado e reconhecido como uma string, haja visto que todo e qualquer dado entre aspas para o núcleo Python é esperado que seja um texto.

Por meio da função print( ), por sua vez parametrizada pela função type( ), parametrizada com a variável numero, ao executar este código será retornado o tipo de dado desta variável.

Note que pela leitura léxica, que se faz sequencial, linha após linha, a variável numero criada logo na primeira linha é do tipo int, ao declarar novamente esta variável na linha 5, não estamos criando uma nova variável de mesmo nome (e isso nem mesmo é possível em Python), mas estamos instanciando e atualizando o dado/valor da variável, nesse caso, de acordo com a sintaxe, de int para string.

Por fim, vale lembrar que pela ordem da leitura léxica o último dado/valor associado a esta variável irá definir seu tipo, até que por ventura algo a modifique novamente. Em nosso exemplo inicial, o último dado/valor associado a numero era do tipo string.

```
1   numero = int(1987)
2
3   print(type(numero))
4
5   numero = int('1987')
6
7   print(type(numero))
8
```

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

Para que possamos determinar o tipo de um dado manualmente, no momento da atribuição do dado/valor a sua respectiva variável é possível usar de um método, inserido como prefixo na declaração do dado/valor.

Dessa forma, de acordo com a leitura léxica, que lê cada linha de código da esquerda para direita, o interpretador irá identificar o tipo de dado antes mesmo de chegar no dado/valor em questão, associando de forma permanente o tipo de dado.

No exemplo, para nossa variável numero, como atributo temos int( ), método interno que define o tipo de dado aqui repassado como parâmetro, nesse caso, 1987.

Repare que na linha 5, ao realizar novamente a declaração da variável numero, agora com o prefixo int( ), mesmo seu conteúdo estando em forma de string (entre aspas), o dado em si respeitará o tipo manualmente definido.

Via função print( ), exibindo o conteúdo da variável numero da linha 1 e da linha 5, temos para ambas que o tipo de dado é int.

```python
nomes: list = ['Alberto', 'Fernando', 'Tania']

print(nomes)
print(type(nomes))

idades: tuple = (74, 33, 61)

print(idades)
print(type(idades))

```

```
['Alberto', 'Fernando', 'Tania']
<class 'list'>
(74, 33, 61)
<class 'tuple'>
```

Uma forma um pouco mais avançada de definir o tipo de um dado manualmente é fazendo o uso do gerador daquele tipo de variável/objeto em Python.

Repare na declaração da variável nome, a mesma possui uma notação um pouco diferente do usual, sendo nome: list, antes mesmo de usar do operador de atribuição “ = “ e do dado/valor atribuído em si.

Dessa forma, no momento da leitura léxica o interpretador irá identificar já o tipo de dado em questão pela própria variável, antes mesmo de qualquer prefixo de atributo.

Da mesma forma criamos a variável idades, por sua vez já definida como do tipo tupla pela notação idades: tuple, seguido do dado/valor atribuído para a mesma.

Por meio da função print( ), exibimos o conteúdo de ambas variáveis assim como seus respectivos tipos de dados.

O retorno, como esperado, é tipo list para a variável nomes, assim como tipo tuple para a variável idades.

```python
var1: set = {}
var2: list = []
var3: tuple = ()
var4: dict = {}

print(__annotations__)

```

```
{'var1': <class 'set'>, 'var2': <class 'list'>, 'var3': <class 'tuple'>, 'var4':
```

Usando desse método é possível definir manualmente o tipo de dado que uma determinada variável receberá. Lembrando que a tipagem de dados em Python é dinâmica, podemos com os métodos explicados anteriormente modificar a tipagem de uma variável/objeto, porém, caso alguma incompatibilidade surja, o interpretador irá contornar tentando ler o dado/valor pela sua forma mais básica, ou seja, pela sua forma sintática, antes mesmo de gerar algum erro ou exceção.

Finalizando nosso raciocínio neste ponto, apenas como exemplo, declaradas 4 variáveis var1, var2, var3 e var4, com diferentes tipos de dados, por meio da função print( ) podemos consultar dos metadados deste código o metadado __annotations__ que retorna todos objetos criados e prontos para uso, com seus respectivos tipos de dado, onde podemos ver como esperado os dados de acordo com seu identificador.

- Tipo esperado e tipo retornado

Se tratando da tipagem de um determinado dado, uma possibilidade que temos, muito comumente usada, é usar de tipagem para os dados que serão repassados como parâmetros de uma função.

```python
def dobro_valor(num: int) -> float:
    return num * 2

```

Como exemplo, note que criamos a função dobro_valor( ), que como esperado, recebe um valor numérico e retorna o seu valor multiplicado por 2.

O ponto aqui é que, assim como mostrado anteriormente, usando da notação na própria criação da variável é possível definir seu tipo. Nesse caso, no campo onde serão definidos os parâmetros dessa função temos a variável num, que pela notação num: int, deve receber um dado/valor do tipo int.

Em seguida é usado do apontador “ -> “ seguido de float, dessa forma definimos que o tipo de dado a ser retornado por essa função será do tipo float.

Novamente, caso os dados usados nesta função não respeitem essas notações, o interpretador antes de gerar algum erro tentará contornar qualquer incompatibilidade adotando a forma sintática original do dado sendo repassado para a função.

- Tipando manualmente o tipo da variável e separadamente o tipo de seu conteúdo.

```python
from typing import List

digitos1: List[int] = [2, 4, 6, 8]
digitos2: List[str] = ['1', '3', '5', '7']

print(type(digitos1))
print(type(digitos1[0]))

print(type(digitos2))
print(type(digitos2[0]))

print(__annotations__)

```

```
<class 'list'>
<class 'int'>
<class 'list'>
<class 'str'>
{'nomes': <class 'list'>, 'idades': <class 'tuple'>, 'var1': <class 'set'>, 'var
```

Avançando um pouco, outra possibilidade que temos se tratando de tipagem de dados é separadamente definir o tipo de dado identificado para a variável em si, e o tipo de dado a ser identificado para o conteúdo da mesma, no intuito de evitar erros internos e/ou exceções em nossos programas.

Para isso, inicialmente da biblioteca typing importamos List, módulo que possui algumas funções internas especiais para tratamento de dados elementos de uma lista.

No exemplo, criamos a variável digitos1, pela notação sendo do tipo List (List este que importamos no início do código, e não list tipo de dado básico padrão), que por sua vez será composto de elementos do tipo int.

Do mesmo modo criamos a variável digitos2, que pela notação será “uma lista de strings”.

Verificando o tipo de dado de digitos1 é retornado list, verificando o primeiro elemento da lista associada a digitos1 temos, como esperado, tipo int.

- Duck Typing

```
1   lista2 = [1, 2, 3, 4, 5, 6, 7, 8]
2   nome2 = 'Fernando Feltrin'
3
4   print(len(lista2))
5   print(len(nome2))
6
```

```
8
16
```

A esta altura você certamente já está olhando para as mesmas variáveis velhas conhecidas com outros olhos, e essa é a intenção. Dando continuidade, ainda temos algumas considerações a se fazer quanto a tipagem de dados.

Dos conceitos teóricos da linguagem Python é possível encontrar uma terminologia chamada Duck Typing, que trata de um conceito definido pelo núcleo idealizador da linguagem.

Duck Typing, literalmente é um conceito onde se algo voa como um pato, tem formato de pato, se comporta como pato, etc... podemos deduzir que é um pato! Em outras palavras, no quesito da linguagem Python possuir seus dados dinamicamente tipados, dependendo do contexto mudando de tipo de dado, podemos associar que todo tipo de variável que guarda em seu conteúdo múltiplos dados/valores é um contêiner de dados, logo, devem suportar os mesmos métodos aplicados para si.

Em nosso exemplo, a variável lista2 é uma lista, e a variável nome2 é uma string. Aos olhos do interpretador, tanto lista2 quanto nome2 são contêineres de dados, de modo que as mesmas funções que podemos usar em um se aplicam ao outro.

Nas linhas 8 e 9 descritas duas funções print( ), usamos do método len( ) para descobrir o tamanho, ou número de elementos que compõe tanto lista2 quanto nome2, uma vez que ambas podem ser desmembradas para que possamos iterar sobre cada elemento.

```
1   nascimento = 1987
2
3   print(len(nascimento))
4
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-12-7d812111c847> in <module>()
      1 nascimento = 1987
      2
----> 3 print(len(nascimento))
      4 # Irá gerar um erro pois o método len não se aplica a esse tipo de dado

TypeError: object of type 'int' has no len()
```

No mesmo contexto, criada a variável nascimento, tendo atribuída para si o dado/valor 1987, ao tentar usar do método len( ) sobre a mesma é gerado justamente um erro de tipo, pois nascimento não tem o mesmo comportamento das variáveis lista2 e nome2.

```
1   class Pessoa:
2     def __len__(self):
3       return 1987
4
5   pessoa1 = Pessoa()
6   print(len(pessoa1))
7
```

```
1987
```

Transformando nosso dado original como parte de um contêiner, tendo método (__len__( )) e atributo (1987), é possível retornar essa iteração antes não permitida para um dado do tipo int.

Agora declarando uma variável de nome pessoa1, que instancia a classe Pessoa, podemos iterar sobre seus atributos de classe.

- Tipagem em Comentário

```python
import math

def circunferencia(num):
    # type: (int) -> float
    return 2 * math.pi * num

calculo = circunferencia(8)

print(calculo)
print(type(calculo))

```

```
50.26548245743669
<class 'float'>
```

Outra particularidade interessante, se tratando da tipagem de dados, é que podemos criar marcadores que servirão de instrução para nosso interpretador.

Muito provavelmente você no início de seus estudos da linguagem Python deve ter ouvido o jargão de que o interpretador não lê comentários. Na verdade o mesmo lê, e ignora seu conteúdo, a não ser que neste comentário de forma implícita exista uma instrução de interpretação.

Para nosso exemplo, repare que definida a função circunferencia( ), na primeira linha de seu bloco de código indentado existe em forma de comentário # type: (int) -> float, que especifica que é esperado um dado/valor do tipo int, onde a função retornará seu resultado em formato float.

Novamente no escopo global do código é declarada uma variável de nome calculo, que chama a função circunferencia( ) parametrizando a mesma com o número 8 (tipo int).

Por meio da função print( ) verificamos o valor associado a variável calculo (valor retornado da função), assim como por meio do método type( ) por sua vez parametrizado com a variável calculo podemos ver que de fato o tipo de dado retornado é float.

Apenas fazendo um pequeno adendo, no que diz respeito ao tipo de dado esperado e retornado, é perfeitamente possível definir

mais de um tipo de dado esperado, assim como mais de um tipo de dado retornado.

Por exemplo: # type: (str, bool) -> str. Nesse caso, é esperado que os dados de entrada sejam ou do tipo string ou do tipo booleano, retornando apenas em formato de string.

Tipos de Dados Especiais

- Tipo Literal

```
1   from typing import Literal
2
3   def usuario(user: str) -> Literal['logado', 'deslogado']:
4       if user == 'logado':
5           print('Logado no sistema')
6       if user == 'deslogado':
7           print('Deslogado do sistema')
8       if user != 'logado' and user != 'deslogado':
9           print('Status inválido')
10
11  var1 = usuario('deslogado')
12
```

Dependendo do contexto e aplicação, pode ser necessário definir manualmente não somente o tipo de dado esperado como o tipo de dados/valores do campo de atributos a ser retornado.

Inicialmente da biblioteca typing importamos Literal, logo em seguida, ao definir a função usuario( ), especificamos no campo de parâmetros da mesma que o parâmetro user espera um dado do tipo string.

Pela notação do apontador ->, definimos que o tipo de dado a ser retornado será do tipo Literal, e não somente isso, mas que literalmente/obrigatoriamente este dado seja ou ‘logado’ ou ‘deslogado’, não permitindo nenhuma outra saída senão esta.

No corpo da função usuario( ) é criada toda uma estrutura condicional onde de acordo com a entrada é feita uma validação do tipo de dado de saída em Literal, reforçando que somente é permitido como dado retornado ‘logado’ ou ‘deslogado’, qualquer outra string diferente destas repassada como parâmetro para função usuario( ) irá exibir em tela ‘Status inválido’ para o usuário.

No corpo geral do código criamos uma variável de nome var1 que chama a função usuario( ) parametrizando a mesma com ‘deslogado’, dessa forma, será retornado ‘Deslogado do sistema’.

- Tipo Union

```
1   from typing import Union
2
3   def soma(num1: int, num2: int) -> Union[str, int]:
4
5     res_soma = num1 + num2
6     print(f'O resultado da soma é: {res_soma}')
7     return res_soma
8
9   calculo1 = soma(99, 12)
10
```

```
O resultado da soma é: 111
```

Outro tipo de dado, para situações bastante específicas, é o tipo Union, onde a grande diferença deste tipo para os demais é que em Union é permitido que os dados de saída sejam de tipos de dados normalmente incompatíveis.

Em outras palavras, até então era impossível realizar certas ações sobre dados como string e int, podendo gerar exceções. Por meio de Union dados destes tipos (apenas como exemplo) podem interagir entre si, mesmo com comportamentos estranhos (o que pode ser conferido debugando o código via MyPy).

Partindo para o exemplo em si, inicialmente importamos da biblioteca typing o módulo Union.

Na sequência definimos uma função chamada soma( ) que por sua vez recebe dois parâmetros/argumentos, sendo que para num1 é esperado um dado do tipo int, o mesmo se estende a num2.

Em seguida, é denotado que o dado de saída será do tipo Union, podendo ser tanto em formato str quanto em formato int.

```python
def soma(num1: Union[int, float], num2: Union[int, float]) -> Union[str, int]:
    res_soma = num1 + num2
    print(f'O resultado da soma é: {res_soma}')
    return res_soma

calculo1 = soma(99, 12)

```

```
O resultado da soma é: 111
```

Apenas lembrando que todas essas notações para tipagem de dados se aplicam para todo e qualquer objeto em Python.

Por exemplo, modificando nossa função soma( ) agora definimos que seu primeiro parâmetro num1 será do tipo Union, podendo ser um dado de entrada tanto em formato int quanto float, repetindo essa configuração para num2, por fim especificando que o dado/valor de saída será do tipo Union, podendo ser tanto em formato string quanto int.

- Tipo Final

```python
from typing import Final

NOME: Final = 'Fernando'

print(NOME)

```

Para alguns autores, o tipo de dado Final, também da biblioteca typing, visa trazer o comportamento de uma constante em Python.

Sabemos bem que em Python literalmente não existem constantes, embora seja possível criar uma máscara de identificação que aplicada a um objeto, fará com que o interpretador o identifique como tal.

Por meio do MyPy é possível verificar se algum iterador em algum momento tentou modificar essa “constante”, e se sim, irá

gerar um erro apontando que estrutura de código tentou modificar o tipo de dado padrão desse objeto.

Embora usualmente pela sintaxe uma variável/objeto qualquer declarado com nome todo em letras maiúsculas represente uma constante, de fato tal variável/objeto continuará dinamicamente tipado.

- Decorador final

Ainda se tratando de meios e métodos para tentar tornar uma variável/objeto uma constante, a biblioteca typing oferece um decorador dedicado a esse processo, pois decoradores atuam em uma camada de código ainda mais interna, fazendo com que o interpretador seja “forçado” e tratar os dados aqui decorados como constantes.

```
1  from typing import final
2
3  class Pessoa:
4
5    def acao1(self):
6      print('Trabalhando...')
7
```

Para ficar mais claro este conceito, inicialmente importamos da biblioteca typing o módulo final (não confundir com Final, de nome iniciado com letra maiúscula).

Em seguida criamos uma simples classe de nome Pessoa que possui o método de classe acao1( ), que por sua vez simplesmente exibe em tela via função print( ) a mensagem ‘Trabalhando...’.

```
1  @final
2  class Pessoa:
3    def acao1(self):
4      print('Trabalhando...')
5
```

Colocando o decorador @final na linha acima da declaração da classe Pessoa, estamos fazendo com que toda essa classe fique protegida como uma constante pelo interpretador.

Dessa forma, nenhuma classe que herdar essa estrutura conseguirá sobrescrever dados na mesma. Via MyPy é possível ver a exceção que acontece nesses casos, embora o código continue funcionando perfeitamente na camada de interação com o usuário, internamente haverá um logo dizendo que não se deveria iterar sobre a classe Pessoa.

```
1  class Pessoa:
2      @final
3      def acao1(self):
4          print('Trabalhando...')
5
```

Alterando a posição do decorador @final para dentro da classe Pessoa, o mesmo estando posicionado logo acima de um método de classe ou até mesmo de atributos de classe fará com que os mesmos fiquem também protegidos como uma constante.

Novamente, importante salientar que para estes tipos de dados especiais, caso haja alguma incompatibilidade de iteração sobre os mesmos, o interpretador irá contornar essas questões mantendo o código funcionando, porém, debugando o código é possível ver todas as pequenas exceções geradas.

- TypedDict

Outra possibilidade existente, no âmbito de tipagem de dados, é a de se criar um dicionário tipado orientado a objetos. Este dicionário por sua vez também é um tipo de dado especial, de comportamento parecido com a estrutura básica de dicionário do núcleo Python, porém de características de tipagem exclusivas.

```python
from typing import TypedDict

class Programa(TypedDict):
    versao: str
    revisao: float

projeto1 = Programa(versao = '1.0.237', revisao = 0.01)

print(projeto1)

```

Como de costume, inicialmente importamos as devidas bibliotecas, módulos e pacotes que serão utilizados ao longo de nosso código. Neste caso, da biblioteca typing importamos TypedDict.

Na sequência criamos uma classe de nome Programa( ) que instancia TypedDict como argumento. Dentro do corpo dessa classe criamos os objetos versao e revisao, que serão as chaves deste dicionário, e pela notação esperam receber dados dos tipos str e float, respectivamente.

Voltando ao escopo global do código, criamos uma variável de nome projeto1 que atribui para si a classe Programa, repassando os valores para as chaves anteriormente criadas nesse dicionário.

Dessa forma, como no enunciado deste tópico, teremos um dicionário totalmente tipado manualmente, sendo sua estrutura toda orientada a objetos.

- Tipo Protocol

Finalizando este capítulo, nos resta conhecer um último tipo de dado especial voltado para tipagem manual, o tipo de dado chamado Protocol.

Conceitualmente, este tipo de dado possui a característica de definir objetos/atributos de classe tipados, onde uma tipagem é definida manualmente e a partir daí todas classes filhar terão de

literalmente “seguir o protocolo”, respeitando a tipagem importa para que seus dados internamente não gerem exceções.

```
1   from typing import Protocol
2
3   class Pessoa(Protocol):
4     nome: str
5
6   def identificacao(primeiro_nome: Pessoa) -> str:
7
8     print(f'Usuário identificado como: {primeiro_nome.nome}')
9
10  class Pessoa1:
11    nome = 'Fernando'
12
13  pessoa1 = Pessoa1()
14  identificacao(pessoa1)
15
```

Da biblioteca typing importamos Protocol. Em seguida criamos a classe Pessoa( ) que instancia Protocol, onde logo na primeira linha do corpo da classe é criado o objeto nome que espera um dado do tipo str.

Aqui é definida a regra de que obrigatoriamente as classes herdeiras terão de ter um “nome” do tipo string para que se comportem como esperado pela classe molde.

No corpo geral do código criamos uma função de nome identificacao( ), que por sua vez tem um parâmetro primeiro_nome que espera um tipo de dado Pessoa, retornando uma string. Essa função por sua vez simplesmente exibe uma mensagem interagindo com o dado repassado para primeiro_nome, validado pelo objeto nome da classe.

Na sequência criamos uma classe herdeira de nome Pessoa1, e em seu bloco de código indentado instanciamos nome, atribuindo uma string, já que esse é um objeto obrigatório segundo o protocolo.

Por fim, criamos uma variável de nome pessoa1 que instancia a classe Pessoa1, a partir deste ponto, executando o método de classe identificacao( ) será retornado ‘Fernando’.

Listas

- Métodos inclusos nos built-ins

```
append()
# Adiciona um elemento ao final de uma lista

insert()
# Adiciona um elemento em uma posição específica do índice

clear()
# Remove todos os elementos de uma lista

copy()
# Associado a outra variável, copia todo o conteúdo de uma lista

count()
# Retorna o número de vezes que um elemento aparece em uma list
a

extend()
# Adiciona os elementos de um objeto (ou de outra lista) ao final da l
ista original

index()
# Retorna o número de índice de um determinado elemento da lista

pop()
# Remove um determinado elemento da lista por meio de seu númer
o de índice

remove()
# Remove um determinado elemento da lista por meio de seu nome/
dado/valor
```

```
reverse()
# Inverte a ordem dos elementos da lista

sort()
# Ordena os elementos da lista
```

Strings

- Métodos inclusos nos built-ins

```
capitalize()
# Converte o primeiro caractere para maiúsculo

casefold()
# Converte uma string para letras minúsculas

center()
# Retorna uma string centralizada

count()
# Retorna o número de vezes em que um elemento aparece na strin
g

encode()
# Retorna uma versão convertida da string (parametrizar com o tipo
de texto, ex: utf-8)

endswith()
# Dado um dado/valor a ser usado para identificador do fim do texto,
 retornará True caso o texto de fato termine com tal dado

expandtabs()
# Define o tamanho da tabulação do texto (margens)

find()
# Busca no texto um determinado elemento, retornando a posição d
e índice onde o mesmo foi localizado

index()
```

```
# Dada uma certa palavra, retorna o número de índice onde a mesm
a foi localizada por primeiro

isalnum()
# Retorna True se todos os caracteres forem de um tipo numérico

isalpha()
# Retorna True se todos os caracteres forem alfanuméricos, sem sí
mbolos especiais

isdecimal()
# Retorna True se todos os elementos forem do tipo numérico com c
asa decimal

isdigit()
# Retorna True se todos os elementos forem dígitos

islower()
# Retorna True se todos os caracteres da string estiverem em minús
culo

isnumeric()
# Retorna True se todos os elementos forem números, sem símbolo
s especiais

isspace()
# Retorna True se todos os elementos de uma string forem espaços
em branco

istitle()
# Retorna True se todas as palavras de uma frase forem iniciadas c
om letra maiúscula

isupper()
# Retorna True se todas as letras de uma string forem letras maiúsc
ulas
```

```
join()
# Junta uma nova string ao final da primeira usada como referência

ljust()
# Alinha à esquerda uma string

rjust()
# Alinha à direita uma string

lower()
# Converte todos elementos de uma string para letra minúscula

lstrip()
# Remove todos os espaços à esquerda de uma string

rstrip()
# Remove todos os espaços à direita de uma string

maketrans()
# Retorna uma tradução para a string a partir de outra variável

partition()
# Retorna uma tupla onde a string original é dividida em 3 partes

replace()
# Substitui um elemento de uma string por outro

rfind()
# Busca por uma determinada string mas retorna a última instancia
da mesma encontrada

rindex()
# Busca por uma determinada string mas retorna o último número d
e índice onde a mesma foi encontrada

startswith()
```

```
# Com base em uma palavrachave, retorna True caso essa palavra
seja a primeira da string

title()
# Converte o primeiro caractere de cada string para maiúsculo

translate()
# Retorna uma string traduzida com base em informação contida em
 uma variável

upper()
# Converte toda a string para letras maiúsculas
```

- Particularidades envolvendo o tamanho de uma string

```
1  nome = 'Alberto'
2
3  nome2 = 'Alberto\n'
4  nome3 = 'Alberto '
5
6  print(len(nome))
7  print(len(nome2))
8  print(len(nome3))
9
```

```
7
8
8
```

Alguns conflitos podem ser gerados em alguns contextos, às vezes por puro erro de interpretação de espaços pelo interpretador.

Em nosso exemplo, temos a variável nome que cujo atributo é a string ‘Alberto’, contendo 7 elementos/caracteres.

Já para a variável nome2 temos como atributo ‘Alberto\n’, onde o marcador \n apesar de ser constituído de dois caracteres, é lido e

interpretado como 1 elemento somente.

Por fim, para variável nome3 temos ‘Alberto ‘, onde o espaço após o nome em si é lido normalmente como 1 caractere. Em suma, certos métodos podem gerar exceções quando ler espaços em lugares imprevistos.

- Tratando os espaços desnecessários de uma string

```
1   nome = ' Fernando '
2
3   print(nome)
4
5   nome2 = nome.rstrip()
6   print(nome2)
7
8   nome3 = nome.lstrip()
9   print(nome3)
10
11  nome4 = nome.strip()
12  print(nome4)
13
```

```
 Fernando
 Fernando
Fernando
Fernando
```

Visando tratar os espaços de uma string no intuito de evitar exceções, podemos usar de algumas funções básicas do sistema.

Declarada uma variável de nome nome, cujo atributo é a string ‘ Fernando ’, ao exibir o conteúdo da variável via função print( ) é retornado o dado literal da variável, ou seja, seu texto incluído os espaços que houverem em sua composição.

Usando da função rstrip( ) aplicada à nossa variável nome, são removidos os espaços desnecessários à direita da string.

Da mesma forma, via lstrip( ) removemos os espaços desnecessários à esquerda da string, por fim via strip( ) são removidos os espaços em ambos os lados da string.

Dicionários

- Métodos inclusos nos built-ins

```
keys()
# Retorna uma lista contendo todas as chaves de um dicionário

values()
# Retorna uma lista contendo todos os valores de um dicionário

items()
# Retorna uma lista contendo tuplas com os pares de chaves e valor
es de um dicionário

get()
# Retorna o valor de uma chave específica

clear()
# Remove todos os elementos de um dicionário

copy()
# Associado a outra variável, retorna uma cópia do dicionário origina
l

fromkeys()
# Retorna um dicionário apenas com chaves e valores específicos

pop()
# Remove uma chave e seu respectivo valor

popitem()
# Remove o último par de chaves:valores inserido no dicionário

setdefault()
```

```
# Retorna o valor de uma chave específica, caso a mesma não exist
a, cria uma chave com valor padrão

update()
# Atualiza um dicionário inserindo no mesmo um novo par de chaves
:valores
```

- Dicionário ordenado

Se tratando de tipos de dados especiais, um dos tipos implementados no Python por meio da biblioteca collections é o chamado OrderedDict.

Basicamente este tipo de dado é um dicionário comum porém com um comportamento interno um pouco diferente do convencional, pois o mesmo possui um mapeamento e indexação de seus dados de modo que toda e qualquer alteração sobre o conteúdo do mesmo pode ser identificado com ferramentas de debug como MyPy.

Como em Python basicamente temos dados dinamicamente tipados de tal forma que não temos constantes absolutas, apenas máscaras que tentam proteger dados identificando variáveis/objetos e seus conteúdos como protegidos, simulando uma constante, o tipo OrderedDict veio trazer esse comportamento para dados em forma de dicionário.

```python
from collections import OrderedDict

dicionario = OrderedDict()
dicionario['a'] = 1
dicionario['b'] = 2
dicionario['c'] = 3
dicionario['d'] = 4
dicionario['e'] = 5
print(dicionario)

for k, v in dicionario.items():
    print(k, v)

```

```
OrderedDict([('a', 1), ('b', 2), ('c', 3), ('d', 4), ('e', 5)])
a 1
b 2
c 3
d 4
e 5
```

Por parte da criação e uso deste tipo de dado, basicamente de início é necessário importar o mesmo da biblioteca collections, em seguida, o atribuir a uma variável.

A partir daí, podemos inserir dados manualmente da mesma forma como fazíamos em um dicionário comum, criando um campo referente a chave, atribuindo um valor.

Em nosso exemplo, basicamente inserimos alguns itens neste dicionário, em seguida os exibindo em tela por meio da função print( ).

Novamente, o importante a ser entendido desse dado é a característica do mesmo tentar preservar seu conteúdo original, sendo útil para situações onde haverão dados em nosso programa dispostos em forma de chave e valor sendo os mesmos cruciais para o funcionamento do código de tal modo que seja necessário os proteger como constantes.

Os dados em si na prática poderão sim ser modificados, por exemplo, sendo atualizados como em um banco de dados, porém fazendo o debug de nosso código será possível ver todas as iterações e alterações sobre os dados deste objeto.

- DefaultDict

Outro tipo de dado bastante específico implementado pela biblioteca collections, usando como base e funcionalidade um dicionário, é o chamado DefalutDict.

Certamente você em seus estudos iniciais de dicionários já se deparou com situações onde ao tentar instanciar uma chave ou valor inexistente em um dicionário era gerado um erro, e não um simples erro, mas em certos casos um runtime error que encerrava o programa.

Visando contornar especificamente este tipo de problema, um DefaultDict pode ser usado de forma que qualquer coisa que iterar sobre o mesmo, mas em um conteúdo inexistente, apenas receberá um retorno padrão do sistema, não afetando em absolutamente nada no funcionamento de todo o contexto do código.

```
1   from collections import defaultdict
2
3   dicionario = defaultdict(int)
4   dicionario['Ano'] = 2021
5   dicionario['Mês'] = 5
6
7   print(dicionario.items())
8   print(dicionario['Dia'])
9
10
```

```
dict_items([('Ano', 2021), ('Mês', 5)])
0
```

Na prática, inicialmente importamos da biblioteca collections nosso defaultdict, em seguida atribuindo o mesmo a uma variável.

O mecanismo de adicionar dados manualmente se dá da mesma forma como no exemplo anterior e como em qualquer dicionário.

Por fim note que fazendo o uso de duas funções print( ), primeiro exibindo em tela todos os elementos de nosso dicionário, e na segunda função print( ) simulando de propósito um erro, tentando exibir uma chave/valor inexistente, simplesmente é retornado 0 e, como mencionado anteriormente, nenhum erro é gerado em função disso.

Bool

Para este tipo de dado em particular da linguagem Python, Booleano, vamos identificar algumas características pouco comentadas sobre estes tipos de dados tão úteis em aplicações reais.

Partindo do princípio de que todo e qualquer dado válido em Python tem um valor oculto booleano True, vamos entender algumas especifidades deste tipo de dado.

```
1   print(bool('Fernando'))
2
3   print(bool(''))
4
```

```
True
False
```

Toda string é True, exceto strings vazias, logo, para nossa primeira função print( ) é natural que o retorno seja True, já para nossa segunda string o valor será false, pois apesar das aspas já configurarem uma string, e ser perfeitamente normal termos strings vazias, para fins de lógica booleana este tipo de string não tem um dado/valor válido.

```
1   print(bool(1987))
2
3   print(bool(0))
4
```

```
True
False
```

Todo número é True, com exceção do número 0. E isto se dá porque quando programamos algo voltado para circuitos lógicos é uma prática normal ter um valor 1 ou qualquer outro valor para denotar uma ativação de nó, enquanto o número 0 representa a ausência de informação para aquele ponto, simulando interações como ligado e desligado em nós do circuito.

Exibindo em tela via função print( ) o valor booleano de 1987 é retornado True, já para 0 o retorno esperado é False.

```
1   nomes = ['Ana', 'Carolina', 'Jéssica']
2
3   print(bool(nomes))
4
5   lista = []
6
7   print(bool(lista))
8
```

```
True
False
```

Qualquer lista, tupla, conjunto ou dicionário será True, exceto quando vazios. Em outras palavras, um contêiner de dados só será válido pela lógica booleana a partir do momento que possuir ao menos um elemento em sua composição.

Por meio de nossa velha conhecida função print( ), exibindo o valor booleano referente a lista nomes o retorno será True, já ao repetir o mesmo processo para a lista lista, que é inicialmente uma lista vazia, o retorno é False.

```
1   id = None
2
3   print(bool(id))
4
```

```
False
```

None, palavra reservada ao sistema que define “nenhum dado/valor” para uma variável, sempre retornará False, uma vez que se uma variável não possui nenhum dado, não há como validá-lo.

```python
class MinhaClasse():
    def __len__(self):
        return 0

variavel1 = MinhaClasse()
print(bool(variavel1))

```

```
False
```

Para estrutura de dados que realizam alguma ação, em seguida retornando algo para o contexto geral do código, retornar o número 0 nesses casos fará com que o mesmo não seja lido de forma literal, mas em sua forma booleana, ou seja, return 0 é o mesmo que return False.

```python
def minha_funcao():
    return True

print(minha_funcao())

```

```
True
```

Por fim, o inverso também é perfeitamente normal, como por exemplo, return True. Na verdade, como dito anteriormente, retornar qualquer dado diferente de 0, diferente de False e diferente de uma estrutura contêiner vazia, será validado como True.

Sets

- Métodos inclusos nos built-ins

```
add()
# Adiciona um elemento ao set

pop()
# Remove um elemento específico do set

remove()
# Remove um elemento específico do set

discard()
# Remove um elemento específico do set

clear()
# Remove todos os elementos de um set

copy()
# Associado a outra variável, retorna uma cópia do set original

difference()
# Associado a outra variável, retorna um novo set contendo a diferen
ça entre dois sets

intersection()
# Associado a outra variável, retorna um novo set contendo a interse
ção de dois outros sets

union()
# Retorna um set contendo a união dos elementos de dois outros se
ts
```

```
update()
# Atualiza um set unindo seu conteúdo original com o de outro set

isdisjoint()
# Retorna True se dois sets tem uma interseção

issubset()
# Retorna True se outro set contém os elementos do set original

issuperset()
# Retorna True se o set original contém os elementos de outro set
```

Função print( )

- Quebrando em linhas o conteúdo de uma string parâmetro da função print( )

```
1   print('O resultado dos testes serão divulgados amanhã. Grato pela compreensão.')
2
```

```
O resultado dos testes serão divulgados amanhã. Grato pela compreensão.
```

```
1   print('O resultado dos testes serão divulgados amanhã.')
2   print('Grato pela compreensão.')
3
```

```
O resultado dos testes serão divulgados amanhã.
Grato pela compreensão.
```

```
1   print('O resultado dos testes serão divulgados amanhã.\nGrato pela compreensão.')
2
```

```
O resultado dos testes serão divulgados amanhã.
Grato pela compreensão.
```

Uma dúvida recorrente aos estudantes da linguagem Python é como se realiza a quebra em linhas do conteúdo de uma função print( ).

Explorando repositórios é bastante comum encontrar códigos onde o desenvolvedor quando precisa exibir em tela uma mensagem que se estende por linhas, acabar criando uma função print( ) para cada linha, o que não faz muito sentido, haja visto que temos um marcador específico para esse propósito.

Analise os blocos de código acima, no primeiro temos uma mensagem longa a ser exibida em tela, que dependendo do contexto pode ser quebrada em duas linhas.

No segundo bloco temos o método mais comum de ser realizado, usando de uma função print( ) para cada linha. É claro

que este modo funciona perfeitamente, porém para situações reais onde pode se ter uma mensagem um tanto quanto elaborada, se torna ineficiente, além de sujar o código.

Já no último bloco temos o que seria o mais adequado, uma vez que bastando posicionar o marcador “ \n “ em algum ponto da string/f’string, o texto será quebrado em linha naquele ponto.

- Exibindo em tela o nome e o conteúdo de uma variável

```
1   nome: str = 'Fernando Feltrin'
2
3   print(f"A variável é composta de: '{nome = }'")
4
```

Uma situação bastante boba, mas que aposto que fez você recorrer ao stackoverflow, e terminou em alguma gambiarra para realizar tal feito, é simplesmente exibir o nome literal de uma variável e seu respectivo atributo.

Pois bem, usando da notação especial, onde para nossa variável nome, definimos que obrigatoriamente seu tipo de dado será string, atribuindo para mesma um dado, nesse caso ‘Fernando Feltrin’, basta que em uma máscara de substituição instanciemos a mesma, da forma como está no exemplo.

Desse modo, será retornado: A variável é composta de: ‘nome = ‘Fernando Feltrin’’.

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.

- 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( ).

- 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.

- Parâmetro/argumento somente posicional

Explorando algumas funcionalidades um tanto desconhecidas do processo de definição de uma função em Python, vamos buscar entender que, como em Python uma função também é um objeto, existem formas de definir certas regras sobre como certas estruturas de código irão interagir com nossas funções.

Do básico estamos acostumados a usar da sintaxe padrão para criar uma função, assim como por padrão para os parâmetros da mesma temos o costume de trabalhar com parâmetros posicionais ou justapostos, assim como parâmetros nomeados.

Usuários com uma bagagem de conhecimento um pouco maior tem costume de implementar parâmetros baseados em *args e **kwargs, mas uma prática que é útil porém bastante desconhecida é a de usar de alguns marcadores para definir regras de interação sobre nossos parâmetros de função.

```
1  def msg(nome, /, idade):
2      return f'Olá {nome}, você tem {idade} anos.'
3
4  print(msg('Fernando', idade = 33))
5
6  print(msg(nome = 'Alberto', idade = 67))
7
```

Nada melhor que partir para prática para entender de fato o conceito de parâmetros regrados como somente posicionais/justapostos.

Inicialmente definimos nossa função msg( ) que receberá dados para dois parâmetros nome e idade, porém, note que entre os

mesmos foi colocado uma barra invertida. Este marcador em particular irá definir que todo parâmetro à sua esquerda obrigatoriamente será somente posicional.

Lembre-se dos parâmetros baseados em *args e **kwargs, para *args repassávamos todos parâmetros justapostos, e para **kwargs todos nomeados, sempre nessa ordem, nunca o contrário.

Aqui ocorre que pelo marcador “ / “ posicionado em algum ponto no campo de parâmetros de uma função, todos parâmetros situados à esquerda do mesmo serão obrigatoriamente posicionais, deixando para os demais parâmetros (à sua direita) que tenham qualquer comportamento.

Apenas como exemplo, veja que nossa função simplesmente retornar uma mensagem, fazendo uso de f’strings, incorporando nas máscaras de substituição os dados repassados para nome e para idade.

Em seguida criamos uma função print( ) que chama a função msg( ) parametrizando a mesma corretamente conforme a regra definida pelo marcador, onde ‘Fernando’ (argumento justaposto/posicional) será atribuído para nome, enquanto idade = 33 (argumento notavelmente nomeado) será atribuído para idade.

O retorno neste caso simplesmente seria: Olá Fernando, você tem 33 anos.

Na sequência, criamos uma outra função print( ), agora simulando de propósito um erro. Note que como parâmetro de nossa função print( ) instanciamos a função msg( ), agora parametrizando a mesma com dois parâmetros nomeados, o que certamente irá gerar um erro.

```
1  def msg(nome, idade, /):
2      return f'Olá {nome}, você tem {idade} anos.'
3
4  print(msg('Fernando', 33))
5
```

Simulando um outro contexto, onde por ventura seja preciso estipular que para nossa função será somente permitidos

parâmetros justapostos/posicionais, basta colocar o marcador “ / “ na última posição do campo de parâmetros da função.

- Parâmetro/argumento obrigatoriamente nomeado

Entendida a lógica da obrigatoriedade dos parâmetros de uma função serem somente justapostos/posicionais, sendo essa característica definida na sintaxe da definição da função, raciocine que em certas situações podemos precisar exatamente do oposto.

Em outras palavras, dependendo muito do contexto, podemos estipular que os parâmetros de uma função sejam obrigatoriamente nomeados.

```
1  def msg(*, nome, idade):
2      return f'Olá {nome}, você tem {idade} anos.'
3
4  print(msg(nome = 'Fernando', idade = 33))
5
```

Fazendo o uso do marcador “ * “ em uma certa posição do campo de parâmetros da função, definimos que todos parâmetros que estiverem à direita deste marcador obrigatoriamente terão de ser parâmetros nomeados.

Usando do mesmo exemplo anterior, agora a função print( ) que antes iria gerar erro, funcionará perfeitamente, pois chamamos nossa função msg( ) parametrizando a mesma com dois parâmetros nomeados.

- Regrando a obrigatoriedade de tipo de parâmetro da função

```python
def msg(nome, /, idade, *, altura):

    return f'Olá {nome}, você tem {idade} anos e mede {altura}m.'

print(msg('Fernando', 33, altura = 1.90))

```

Agora fazendo o uso combinado dos métodos de regra entendidos anteriormente, podemos definir o comportamento de recebimento dos parâmetros de nossa função simulando uma aplicação real.

Definida a função msg( ), que por sua vez receberá 3 parâmetros, nome, idade e altura, vamos às regras.

Pela notação, o primeiro parâmetro (nome) é obrigatoriamente posicional/justaposto, o segundo parâmetro (idade) é justaposto apenas pela posição que o mesmo se encontra, por fim o terceiro parâmetro (altura) é obrigatoriamente nomeado.

f’Strings

A partir da versão 3.6 da linguagem Python, tivemos a definitiva implementação das chamadas f’strings, para alguns autores chamada de estrutura de interpolação, onde usamos de máscaras de substituição “ { } “ para conseguir incorporar em uma string muitas funcionalidades além de seu padrão.

Em outras palavras, ao usarmos de uma f’string estamos usando de uma string “com super poderes”, pois dentro da mesma podemos instanciar de forma reduzida todo e qualquer objeto, inclusive iterando e realizando operações lógicas e/ou aritméticas com outros objetos.

- Notação de dicionário

```
1  pessoa1 = {
2      'Nome': 'Fernando',
3      'Linguagem': 'Python'
4  }
5
6  print(f"{pessoa1['Nome']} é professor de {pessoa1['Linguagem']}")
7
```

```
Fernando é professor de Python
```

Uma particularidade a ser levada em consideração em f’strings é que esta estrutura, por parte de sua sintaxe, é exatamente igual a sintaxe de um dicionário, no sentido de usar chaves { } para conter todo seu conteúdo dentro, onde o interpretador ao ler a chave de abertura já entende se tratar de um dicionário.

Já em uma f’string, o interpretador não espera que tenha propriamente dito um dicionário dentro da string, mas entende que naquele campo demarcado pela simbologia de um dicionário é uma máscara de substituição a ser preenchida com objetos.

No caso de realmente usar da máscara de substituição como iterador para um dicionário, basta trabalhar normalmente instanciando suas chaves/valores ou aplicando métodos comuns a todo e qualquer dicionário.

Como exemplo, temos um simples dicionário de dois pares de chaves/valores atribuído a variável pessoa1.

Por meio da função print( ), usando de f'strings, logo na primeira máscara de substituição, ao instanciar da variável pessoa1 seu campo 'Nome', que é uma chave de dicionário, como esperado será retornado seu valor. O mesmo ocorre para a segunda máscara, substituindo na saída pessoa1['Linguagem'] por Python.

- f'string chamando função do corpo do código

```
1  def soma(n1, n2):
2      return n1 + n2
3
4  n1 = 15
5  n2 = 77
6
7  print(f'O resultado da soma é: {soma(n1, n2)}')
8
```

```
O resultado da soma é: 92
```

Em situações onde realizamos alguma operação matemática simples, sem necessidade de guardar o dado/valor retornado em uma variável, podemos realizar a operação em si diretamente em uma máscara de substituição.

No exemplo temos uma função soma( ) que simplesmente recebe dois valores, retornando a soma dos mesmos, logo em seguida temos duas variáveis n1 e n2 com valores numéricos atribuídos para si.

Por fim em nossa função print( ), fazendo o uso de f'strings, temos uma mensagem a ser exibida para o usuário, onde dentro da

máscara de substituição é chamada a função soma( ), por sua vez parametrizada com os dados das variáveis n1 e n2, retornando uma mensagem personalizada com o retorno desta função.

- Operações matemáticas especiais via f’strings

```
1  import math
2
3  num = 2.2
4
5  print(f'{math.sin(num) = }')
6  print(f'{math.cos(num) = }')
7
```

Uma das possibilidades um tanto desconhecidas, e muito úteis quando aplicadas de forma correta, é usar de operações matemáticas diretamente na máscara de substituição. E quando digo operações matemáticas, não só interação entre dados de variáveis, mas uso de funções matemáticas propriamente ditas.

Inicialmente importamos a biblioteca math, na sequência é criada a variável num cujo atributo é o número do tipo float 2.2.

Na sequência diretamente em nossa função print( ), fazendo uso de f’strings, na máscara de substituição instanciamos a função math.sin( ) por sua vez parametrizada com o dado/valor de num, retornando o valor dessa função ainda dentro da máscara de substituição. Apenas como exemplo, o mesmo é feito usando da função math.cos( ) que calculo o cosseno do valor de num.

O retorno nesses casos será 0.8084964 como seno de 2.2, assim como -0.5885011 como cosseno de 2.2.

- Saída de múltiplas linhas via print( ) + f’string

```
1   nome1 = 'Fernando'
2   nasc1 = '08/11/1987'
3   sexo1 = 'M'
4
5   msg1 = (
6       f'Nome: {nome1}\n'
7       f'Nascido em: {nasc1}\n'
8       f'Sexo: {sexo1}'
9   )
10
11  print(msg1)
12
```

```
Nome: Fernando
Nascido em: 08/11/1987
Sexo: M
```

Em outros momentos já vimos que o marcador “ \n “ serve para identificar ao interpretador que exatamente naquele ponto haverá uma quebra de linha, porém poucos sabem que tal marcador é aplicável a vários contextos, inclusive dentro de f’strings.

Criadas três variáveis nome1, nasc1 e sexo1, com os respectivos atributos ‘Fernando’, ‘08/11/1987’ e ‘M’, em seguida é criada uma variável que fazendo o uso de notação de um contêiner simples de dados guarda três f’strings, onde cada uma delas exibe uma mensagem, incorporando o conteúdo de uma variável em seu corpo e ao fim da linha insere um marcador para quebra de linha.

Repare que aqui, entre cada f’string, não é necessário colocar nenhum separador, como estamos habituados a usar vírgula, porém uma boa prática é organizar as f’strings já em suas respectivas linhas, embora não haja problema nenhum em as declarar juntas.

```python
nome1 = 'Fernando'
nasc1 = '08/11/1987'
sexo1 = 'M'

msg1 = (
    f'Nome: {nome1}\n' f'Nascido em: {nasc1}\n' f'Sexo: {sexo1}'
)

print(msg1)

```

```
Nome: Fernando
Nascido em: 08/11/1987
Sexo: M
```

```python
print(f'Python usa {{}} para incorporar variáveis, objetos,\n'
f'dados, valores, métodos, etc... diretamente na string.')

```

```
Python usa {} para incorporar variáveis, objetos,
dados, valores, métodos, etc... diretamente na string.
```

- f’string como retorno do método de classe __repr__

```python
class Pessoa:
  def __init__(self, nome, profissao):
    self.nome = nome
    self.profissao = profissao
  def __repr__(self):
    return f'{self.nome} trabalha como {self.profissao}'

pessoa1 = Pessoa('Fernando', 'Eng. da Computação')

print(f'{pessoa1}')

```

```
Fernando trabalha como Eng. da Computação
```

Em estruturas de código orientadas a objetos, quando queremos retornar algo por meio de uma f’string podemos fazer uso dos métodos mágicos __str__( ) ou __repr__( ) que suportam este

tipo de dado, suas particularidades e comportamento em relação a outras estruturas de dados.

No exemplo, é criada uma classe de nome Pessoa, que em seu método construtor gera dois atributos de classe de nomes nome e profissao.

Na sequência é criado um método de classe que por sua vez é o método mágico __repr__( ), que basicamente retorna qualquer tipo de dado para qualquer escopo quando a classe for instanciada por uma variável/objeto externo.

Indentado para __repr__( ) temos um return seguido de uma f'string com suas respectivas máscaras de substituição para self.nome e self.profissao.

Já fora do corpo da classe, é criada uma variável de nome pessoa1 que instancia a classe Pessoa( ), repassando como argumentos para a mesma 'Fernando' e 'Eng. Da Computação' a serem atribuídos para seus atributos de classe.

Por fim, via função print( ) parametrizado com pessoa1, é possível ver que o retorno gerado foi referente ao método __repr__( ).

Contêineres de Dados

Em Python (e não somente nessa linguagem de programação) temos uma série de tipos de dados que suportam múltiplos dados/valores/elementos/objetos dentro de sua estrutura.

Falando de uma forma mais funcional, todo e qualquer tipo de variável que possui atribuído para si tipos de dados como listas, tuplas, dicionários, conjuntos, etc... podem ser chamados simplificadamente de contêineres.

Para alguns autores e colaboradores, por uma questão de conveniência é permitido o uso dessa nomenclatura uma vez que os métodos que serão aplicados sobre estes tipos de dados serão basicamente os mesmos, e devido a tipagem dinâmica dos dados em Python, o método que se aplica a um objeto tem características até mais importantes que o tipo de dado em si do objeto e seu comportamento.

- Iterando sobre contêineres

Explorando funcionalidades especiais sobre estes tipos de dados, um dos conceitos pouco conhecidos, porém não menos importante que os demais, é o uso de marcadores para separar elementos de um contêiner iterando diretamente sobre os mesmos, sem alterar a forma do dado original.

```
1   numeros = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
2
3   *primeiros, ultimo = numeros
4
5   print(f'Os primeiros números são: {primeiros}')
6   print(f'O último número é: {ultimo}')
7
```

```
Os primeiros números são: [1, 2, 3, 4, 5, 6, 7, 8, 9]
O último número é: 10
```

Partindo para prática, dada uma tupla com uma série de valores, atribuída para variável números, podemos desempacotar esses dados em uma seleção, diretamente na declaração das variáveis que receberão esses dados, não dependendo mais de estruturas de repetição ou funções específicas.

Basicamente o que faremos é desempacotar os elementos da variável números, inicialmente separando apenas o último elemento dessa tupla.

Para isso, criamos duas variáveis de nomes primeiros e ultimo, que recebem o conteúdo de numeros, porém, repare que na declaração da variável primeiros foi colocado um marcador/identificador “ * “ logo no início do nome da variável. Dessa forma, o interpretador irá reconhecer as duas variáveis que receberão os dados desempacotados, atribuindo para a primeira variável do primeiro ao penúltimo elemento da tupla, enquanto a variável ultimo recebe desse desempacotamento somente o último elemento da tupla.

Embora não exista de fato algo como *args na declaração de uma variável, o marcador “ * “ aqui também é usado, para sinalizar ao interpretador esta propriedade do desempacotamento a ser realizado.

Por meio das respectivas funções print( ) exibimos em tela os conteúdos das variáveis primeiros e ultimo, e como esperado, os dados foram desempacotados da tupla original e separados de acordo com a regra do marcador.

```python
numeros = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

primeiro, *ultimos = numeros

print(f'O primeiro número é: {primeiro}')
print(f'Os demais números são: {ultimos}')

```

```
O primeiro número é: 1
Os demais números são: [2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Da mesma forma, invertendo a atribuição do marcador da variável primeiro para a variável últimos, para a variável primeiro será separado, extraído e atribuído apenas o primeiro elemento da tupla, assim como para ultimos serão repassados todos os demais elementos.

```python
numeros = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

primeiro, *intermediarios, ultimo = numeros

print(f'O primeiro números é: {primeiro}')
print(f'Os números intermediários são: {intermediarios}')
print(f'O último número é: {ultimo}')

```

```
O primeiro números é: 1
Os números intermediários são: [2, 3, 4, 5, 6, 7, 8, 9]
O último número é: 10
```

Usando do mesmo recurso para distribuir dados desempacotados para 3 variáveis diferentes, basta colocar o identificador na posição desejada para ditar a regra de desempacotamento ao interpretador.

Nesse caso, o primeiro elemento será atribuído para a variável primeiro, o último elemento será atribuído para a variável ultimo, e automaticamente todos os demais elementos irão compor a variável intermediarios de acordo com a marcação.

```python
nomes = ['Ana Luiza', 'Carolina', 'Daiane', 'Franciele', 'Maria']

primeiro, *intermediarios, ultimo = nomes

print(f'A lista de chamada começa com: {primeiro}')
print(f'E termina com: {ultimo}')

```

```
A lista de chamada começa com: Ana Luiza
E termina com: Maria
```

Fechando nossa linha de raciocínio, podemos usar deste tipo de notação voltado ao desempacotamento de dados para separar todo e qualquer tipo de dado/valor de dentro de um contêiner.

Supondo que temos uma lista com alguns nomes em uma chamada, usando da mesma regra do exemplo anterior conseguimos sem a necessidade de funções adicionais separar e extrair o primeiro e o último elemento da lista nomes, em seguida via função print( ) criando uma f'string incorporando esses elementos por meio de suas variáveis.

- Migrando dados de contêineres

```python
tupla = (1, 2, 3)
conjunto = {4, 5, 6}

lista = [*tupla, *conjunto]

print(lista)

```

```
[1, 2, 3, 4, 5, 6]
```

Ainda sobre o marcador " * " podemos explorar uma última funcionalidade do mesmo, que se trada de quando houver a necessidade de unir os dados de dois contêineres. O processo comum iria requerer o uso de funções iterando sobre os elementos, desmembrando e depois reconstruindo estruturas de dados, algo

funcional porém não muito eficiente se comparado ao uso de marcador para esse propósito.

Em nosso exemplo temos uma tupla e um set com seus respectivos dados, em seguida criamos uma lista onde faremos a migração desses dados para dentro da mesma.

Para isso, em formato de lista, simplesmente instanciamos como elementos da mesma as variáveis tupla e conjunto, fazendo uso do marcador “ * “ em ambas, dessa forma, naquela posição de índice serão extraídos os dados dos contêineres anteriores.

```
1   dicionario1 = {'one': 1, 'two': 2}
2   dicionario2 = {'three': 3, 'four': 4}
3
4   dicionario3 = {**dicionario1, **dicionario2}
5
6   print(dicionario3)
7
```

```
{'one': 1, 'two': 2, 'three': 3, 'four': 4}
```

Apenas para evidenciar uma última particularidade, supondo que tenhamos que extrair dados de um dicionário, reconstruindo um novo dicionário com a migração de dois ou mais dicionários de origem, o procedimento é exatamente igual, porém fazendo uso do marcador “ ** “, haja visto que para este tipo de estrutura de dados estamos trabalhando com conjuntos de chaves e valores obrigatoriamente.

Orientação à Objetos

Se tratando de programação orientada à objetos, sabemos bem que em Python temos uma enorme flexibilidade para usar deste tipo de estrutura de dados, abrindo um enorme leque de possibilidades.

Claro que quando temos nossos primeiros contatos com este tipo de metodologia de programação, tão diferente da forma básica estruturada, é natural que tenhamos algumas dificuldades, porém com um pouco de prática dominamos esta forma e, fazendo o uso da mesma, aprimoramos de forma considerável nossos programas.

- setattr e getattr

Alguns dos métodos de se trabalhar com atributos de classe chamados setattr e getattr, vieram com o intuito de simplificar o processo de criar objetos dentro de uma classe e iterar sobre os mesmos.

Não menos importante que outros métodos, setattr e getattr nos permite adicionar itens em um dicionário de forma tão simples quanto inserir elementos em uma lista.

```python
class Pessoa():
    pass

pessoa1 = Pessoa()

primeiro_atributo = 'nome' # Primeira chave
segundo_atributo = 'Fernando' # Primeiro valor

setattr(pessoa1, primeiro_atributo, segundo_atributo)

primeiro = getattr(pessoa1, primeiro_atributo)

print(primeiro)

```

```
Fernando
```

Para não confundir este método com outras modalidades de geração de objetos dentro de uma classe, vamos partir da estrutura orientada a objetos mais básica possível. Importante salientar que este método que está sendo apresentado de forma alguma gera conflito com qualquer outro método de estrutura de dados orientada a objetos.

Pois bem, inicialmente criamos nossa classe Pessoa( ), indentado para a mesma apenas pass, palavra reservada ao sistema que instrui o interpretador a simplesmente não fazer nada e partir para próxima linha, já fora da classe.

Na sequência criamos uma variável de nome pessoa1 que instancia nossa classe Pessoa( ), para usar de sua estrutura.

Em seguida é criada uma estrutura de variáveis que se tornarão objetos de nossa classe, como se transferíssemos dados do escopo global do código para o escopo local da classe.

Então é criada a variável primeiro_atributo, que recebe como atributo a string ‘nome’. Da mesma forma é criada a variável segundo_atributo, por sua vez parametrizado como ‘Fernando’. Prestando atenção nessa estrutura é possível deduzir que primeiro_atributo terá o comportamento de uma chave de um

dicionário, assim como segundo_atributo terá o comportamento de um valor associado a chave recém criada.

Partindo para os métodos que são nosso foco neste capítulo, chamamos o método setattr( ) parametrizando o mesmo em justaposição com pessoa1, seguido de primeiro_atributo, finalizando com segundo_atributo.

Dessa forma, para estrutura de dado atribuída a nossa variável pessoa1, que é a classe Pessoa( ), dentro da mesma serão criados os objetos primeiro_atributo = ‘nome’ e segundo_atributo = ‘Fernando’, equivalendo chave e valor de um dicionário, porém sem a necessidade da notação de dicionário.

Para extrair um dado dessa estrutura criada anteriormente, é feito o uso do método getattr( ), usando como parâmetros justapostos a origem do dado, nesse caso pessoa1, e o dado em si de primeiro_atributo.

Lembre-se que o comportamento destes dados dentro da estrutura da classe se assemelha a um dicionário, sem a notação de um dicionário. Em outras palavras, tais dados não estarão declarados entre chaves { } como um dicionário, porém serão equivalentes como chave e valor, no sentido de que quando iterarmos sobre primeiro_atributo (chave), receberemos como retorno o dado de segundo_atributo (valor).

Por fim, usando de nossa função print( ) parametrizada com a variável primeiro, o retorno, como esperado, é ‘Fernando’.

```python
class Pessoa():
    pass

pessoa2 = Pessoa()

dicionario2 = {
        'nome': 'Fernando',
        'idade': '33',
        'nacionalidade': 'Brasileiro'
}

for chaves, valores in dicionario2.items():
    setattr(pessoa2, chaves, valores)

for chave in dicionario2.keys():
    print(getattr(pessoa2, chave))

```

```
Fernando
33
Brasileiro
```

Pois bem, e se quisermos usar de fato um dicionário, com toda sua notação e estrutura básica como estamos acostumados, é possível? Perfeitamente, única particularidade estrutural a se observar é que internamente esses dados se comportarão como um dicionário dentro de outro dicionário.

Usando como molde nossa classe Pessoa( ), é criada uma variável no corpo geral do código de nome pessoa2 que instancia a classe Pessoa( ).

Em seguida é criado um dicionário chamado dicionario2, com alguns itens compondo o mesmo.

Por fim, para inserir nosso dicionário2 dentro da classe, tendo o comportamento de dicionário dentro de dicionário, mas equiparando os dados verticalmente, usaremos de um laço for para inserir nos campos corretos os dados das respectivas chaves e valores.

Como ação executada a cada laço de repetição, aplicamos o método setattr( ) parametrizando o mesmo com pessoa2, chaves e valores.

Isto pode parecer um pouco confuso pois o que estamos acostumados a fazer é o processo oposto, desempacotando dados de um dicionário para duas ou mais variáveis, aqui estamos literalmente empacotando dados nos campos de chaves e valores mapeados pelo método setattr( ).

Para extrair um determinado dado de dentro dessa estrutura, novamente usamos do método getattr( ), iterando sobre a chave de nossa estrutura, obtendo como retorno seu valor.

Operador Walrus

Uma das últimas estruturas de dados implementadas no núcleo Python, consolidado a partir da versão 3.8, é o chamado operador Walrus.

Apesar do nome estranho, o operador Walrus nada mais é do que um operador que permite iterar, atribuir e retornar dados/valores de forma reduzida.

Por sintaxe básica, sempre que fizermos o uso deste tipo de operador, teremos como notação: variável := expressão, sendo a simbologia “ := “ o identificador do operador Walrus para nosso interpretador.

```
1  cidade = 'Curitiba'
2  print(cidade)
3
```

Na prática, podemos reduzir o código acima.

```
1  print(cidade := 'Curitiba')
2
```

Dessa forma!!!

Note que diretamente como parâmetro de nossa função print( ) temos o nome da variável em questão, nesse caso cidade, seu atributo, ‘Curitiba’ e entre os mesmos o operador Walrus.

Este operador tem uma característica em particular que pode parecer um pouco abstrata, mas entenda que pela sua notação, a variável cidade está sendo criada dentro da função print( ). Essa variável por sua vez é uma variável válida e iterável como qualquer outra variável criada no escopo global do código. Por fim, ainda dentro da função print( ), após criada a variável com seu respectivo atributo é executado de fato a função esperada pela função print( ) que é exibir em tela este conteúdo.

```
1  print(cidade := 'Curitiba')
2
3  print(cidade)
4
```

Adicionando uma nova função print( ) algumas linhas depois, parametrizando somente com a variável cidade, é retornado Curitiba, como esperado.

Isto pode parecer de fato um pouco abstrato pois não vemos a variável cidade declarada em sua posição usual, no escopo global do código, mas uma vez executada a linha 1 deste bloco de código, a variável cidade está alocada em memória tendo o comportamento de uma variável normal.

```
1  num1 = 2021
2
3  if num1 > 2000:
4      print(seculo := 'XXI')
5
6  print(num1)
7
8  print(seculo)
9
```

Partindo para outro exemplo, vamos buscar entender o mecanismo funcional do uso do operador Walrus.

Inicialmente é criada uma variável de nome num1, que recebe como atributo um simples número do tipo int.

Em seguida é criada uma estrutura condicional onde se o valor de num1 for maior que 2000, é exibida uma mensagem ao usuário. Porém, observando com atenção o campo de parâmetros, nele está sendo criada a variável seculo , tendo atribuído para si a string ‘XXI’ via operador Walrus, retornando desta linha ‘XXI’.

```
3  if num1 > 2000:
4      seculo = 'XXI'
5      print(f'{seculo}')
6
```

Funcionalmente falando seria exatamente o mesmo que o código acima, onde dentro da indentação da estrutura condicional

foi criada a variável seculo com seu respectivo atributo, em seguida servindo de parâmetro da função print( ) para exibir em tela seu conteúdo.

```
1   num1 = 2021
2
3   if num1 > 2000:
4       print(seculo := 'XXI')
5
6   print(num1)
7
8   print(seculo)
9
```

Retornando ao bloco reduzido via Walrus, finalizando nosso raciocínio, são criadas duas funções print( ), uma exibindo o conteúdo da variável num1, do escopo global do código, outra exibindo o conteúdo da variável seculo, variável suspensa em memória, equivalente a um “escopo local da função”, embora isto não exista na prática.

```
1   print(f'{(metade := 16 / 2)} é a metade de {metade * 2}')
2
```

Apenas demonstrando outra possibilidade interessante, por meio do operador Walrus conseguimos ao mesmo tempo criar uma variável que recebe o retorno se alguma expressão lógica ou aritmética, sem problema nenhum de incompatibilidade entre a expressão e a variável.

Nesse exemplo, em apenas uma linha de código, estamos criando uma variável de nome metade que recebe o resultado da divisão de 16 por 2, seguido de uma outra expressão onde se usa do valor associado a variável metade para agora multiplicar seu valor por 2.

Um ponto a se destacar aqui é que pela leitura léxica do interpretador, logo no início da f’string, em sua primeira máscara de substituição, foi criada a variável metade, na mesma linha, na segunda máscara de substituição já é perfeitamente possível fazer

uso da mesma, pois já está alocada em memória e com valor atribuído.

```
1  carrinho = []
2
3  print('Quando quiser, digite "sair" para encerrar o programa.')
4  while (objeto := input('Digite um objeto: ')) != 'sair':
5      carrinho.append(objeto)
6  print(carrinho)
7
```

Um último exemplo, para se demonstrar uma última importante característica, onde o ponto a ser entendido é que o operador Walrus pode iterar normalmente sobre praticamente qualquer estrutura de dados.

Repare no exemplo, um simples código com um carrinho de compras onde o usuário irá cadastrar itens. Na linha 4 do código temos uma estrutura de repetição while, onde na mesma é criada uma variável de nome objeto que, por meio do operador Walrus associado a nossa função input( ), recebe do usuário um certo objeto a ser inserido no carrinho de compras.

Por fim, ao final da declaração dessa estrutura de repetição while ainda é criada uma simples validação onde enquanto o que o usuário digitar for diferente de ‘sair’, os itens serão adicionados ao carrinho.

Apenas como um exercício mental, tente imaginar como seria toda essa estrutura teria de ser codificada com um bloco para cada funcionalidade, tenho mais linhas de código em sua estrutura final. Por meio do operador Walrus é possível, como descrito no início deste tópico, iterar, atribuir e retornar dados/valores em forma eficiente e reduzida.

Citações em Python

Quando iniciamos nossos estudos da linguagem Python, entendendo seus conceitos iniciais nos é apresentado seus tipos de dados com suas respectivas sintaxes.

Dentro dessa linha de raciocínio nos é dito que todo e qualquer texto que é atribuído a uma variável é um tipo de dado em particular, e que sua sintaxe padrão suporta qualquer coisa composta de caracteres alfanuméricos, desde que os mesmos estejam contidos entre aspas.

Falando especificamente da notação usada, é perfeitamente possível transformar qualquer dado em uma string por meio de aspas simples ‘ ‘ ou aspas duplas “ “, onde para certas situações usaremos ambas, contornando apóstrofos e citações.

```
1   msg1 = 'Olá, seja bem-vindo(a)!!!'
2
3   msg2 = "Marca d'água"
4
5   print(msg1)
6   print(msg2)
7
```

```
Olá, seja bem-vindo(a)!!!
Marca d'água
```

Apenas como exemplo, na primeira linha de código temos uma variável de nome msg1, cuja atribuição é a string ‘Olá, seja bem-vindo(a)!!!’. Como tudo está declarado da forma correta, toda essa frase é uma string, atributo da variável msg1, onde podemos iterar sobre essa frase como um todo ou sobre cada caractere da mesma.

Já na linha 3 temos uma variável de nome msg2, que atribuída para si tem a string “Marca d’água”. Note que na palavra d’água temos um apóstrofo, representado por uma aspa simples, logo, para que não seja gerado nenhum conflito de interpretação, a string como

um todo deverá ser escrita usando aspas duplas, para que esta aspa simples seja identificada apenas como apóstrofo.

Exibindo o conteúdo destas variáveis via função print( ) de fato temos ambas strings, inclusive na segunda mantendo a escrita normal, fazendo uso do apóstrofo na palavra d'água.

- Citações dentro de strings

```
1   msg3 = 'Ele disse, "Python é a melhor linguagem de programação!!!" '
2
3   msg4 = 'Ele disse, \'Python é a melhor linguagem de programação!!!\''
4
5   msg5 = "Ela disse, \"Não encontrei a marca d'água neste objeto...\" "
6
7   print(msg3)
8   print(msg4)
9   print(msg5)
10
```

```
Ele disse, "Python é a melhor linguagem de programação!!!"
Ele disse, 'Python é a melhor linguagem de programação!!!'
Ela disse, "Não encontrei a marca d'água neste objeto..."
```

Uma vez que podemos usar de ambos tipos de aspas como mostrado no exemplo anterior, intercalando corretamente as mesmas podemos normalmente criar citações dentro de uma string.

Observe que na linha 1 de nosso código temos a variável msg3 que por sua vez tem como atributo uma string. Essa string em particular tem uma citação dentro dela, evidenciando algo que uma pessoa supostamente disse, e essa citação em si será identificada pelo interpretador pois está contida em aspas duplas, enquanto toda a string está contida entre aspas simples. *A ordem inversa destas aspas também funcionaria perfeitamente.

Em seguida temos uma variável de nome msg4, e aqui temos uma particularidade. Estamos repetindo exatamente a mesma string da msg3, porém, note que em toda sua extensão foi usado somente de aspas simples, inclusive na citação.

Para que seja possível escrever a citação dessa forma, repare que o segmento da string onde se encontra a citação está contido não somente entre aspas simples, mas entre barras. Desse modo, as barras antes e depois da citação servirão como um marcador para que o interpretador entenda que ali é uma citação dentro de um texto, e que seu conteúdo deve ser exibido normalmente.

Já em nossa variável msg5 estamos unindo as 3 formas, apenas para conhecimento dessas possibilidades. Toda a string está contida entre aspas duplas, e a citação está contida entre barras e aspas duplas especificamente pois dentro de seu conteúdo existe uma palavra com apóstrofo.

Usando dessa notação conseguimos contornar este tipo de problema, um tanto quanto incomum, mas que pode ocorrer.

Docstrings

Basicamente uma docstring é um comentário propositalmente posicionado logo no início de um bloco de código como um módulo, uma classe ou uma função, de modo que seu texto seja identificado pelo interpretador como uma espécie de documentação resumida.

Se tratando de comentários em Python sabemos que internamente este tipo de dado é lido, porém ignorado pelo interpretador. Existem situações em particular onde mesmo é possível passar instruções ao interpretador por meio de comandos contidos em um comentário (por exemplo built-ins como ascii( ), onde passamos em forma de comentário, logo no início do código, a codificação de caracteres que será usada ao longo do código).

- Docstrings de bibliotecas/módulos de terceiros

```python
import pickle

print(pickle.__doc__)

```

```
Create portable serialized representations of Python objects.

See module copyreg for a mechanism for registering custom picklers.
See module pickletools source for extensive comments.

Classes:

    Pickler
    Unpickler

Functions:

    dump(object, file)
    dumps(object) -> string
    load(file) -> object
    loads(string) -> object

Misc variables:

    __version__
    format_version
    compatible_formats
```

Entendendo diretamente na prática, vamos visualizar uma docstring de um módulo já pronto.

Logo no início de nosso código importamos o módulo pickle, muito utilizado para manipulação de dados em contêiner que precisam estar em formato string.

Na sequência, por meio de nossa função print( ), parametrizada com pickle.__doc__ nos é retornado uma documentação resumida inline.

Repare que __doc__ por sua notação é um método interno do sistema, reservado especificamente para este tipo de dado, docstring.

- Criando uma docstring personalizada

Criar uma docstring personalizada para um módulo próprio é algo realmente simples, pois como dito anteriormente, se trata de um comentário posicional, contendo conteúdo relevante como uma documentação resumida.

```
1   # módulo soma.py
2
3   def soma(n1, n2):
4     ''' Esta é uma simples função de soma de dois números,
5     ao chamar a função, dois números devem ser obrigatoriamente
6     repassados para a função, a mesma irá retornar o valor
7     da soma destes números, em formato float.'''
8
9     return float(n1 + n2)
10
```

Supondo que estamos criando um módulo chamado soma.py, dentro do mesmo temos uma função soma( ) que recebe dois números n1 e n2, retornando em formato float o resultado da soma dos mesmos.

Note que logo na primeira linha do corpo da função soma( ) é colocada uma string, de sintaxe aspas triplas, contendo um pequeno bloco de texto com informações sobre a função em si.

```
1   from soma import soma
2
3   soma_numeros = soma(7, 14)
4
5   print(f'O resultado da soma é: {soma_numeros}')
6
7   print(f'Documentação do módulo soma: \n {soma.__doc__}')
8
```

```
O resultado da soma é: 21.0
Documentação do módulo soma:
  Esta é uma simples função de soma de dois números,
  ao chamar a função, dois números devem ser obrigatoriamente
  repassados para a função, a mesma irá retornar o valor
  da soma destes números, em formato float.
```

Em nosso arquivo principal, importamos do módulo soma a função soma criada anteriormente.

Na sequência criamos uma variável de nome soma_numeros que chama a função soma( ), parametrizando a mesma com os valores 7 e 14.

Logo em seguida por meio da função print( ) usando de f'strings e máscaras de substituição criamos uma mensagem a ser exibida ao usuário, incorporando o valor de soma_numeros na mensagem.

Também criamos uma segunda função print( ), esta agora com uma nova mensagem e dentro de sua máscara de substituição repare que é instanciado soma.__doc__, ou seja, da função soma, sua docstring.

Como esperado, ao executar esse arquivo são exibidos os retornos gerados pelas duas funções print( ), sendo a segunda delas nossa docstring personalizada.

Any e All

As funções any( ) e all( ) são funções para validação, muito úteis em casos onde temos contêineres de dados e queremos fazer uma rápida verificação sobre a validade dos mesmos.

- any( )

```
1   condicoes = [False, False, False, True, False]
2
3   condicoes2 = {'ligado': 1, 'desligado': 0}
4
5   print(any(condicoes))
6   print(any(condicoes2))
7
```

```
True
True
```

Diretamente na prática, declarada a variável condicoes que possui como atributo uma lista de elementos, destes bastando um deles ser True a função any( ) retornará True.

Any em tradução livre significa qualquer, logo, retornará True se ao menos um dos elementos de uma tupla/lista ou dicionário for válido.

Via função print( ), por sua vez parametrizada com any( ), verificando a integridade de condicoes2 que é um dicionário, será retornado True caso ao menos uma das chaves tenha um valor válido.

```
1   condicoes2 = {'ligado': 1, 'desligado': 0}
2
3   condicoes3 = {None: 1, 'desligado': 0}
4
5   condicoes4 = {None: 1, False: 0}
6
7   print(any(condicoes2))
8   print(any(condicoes3))
9   print(any(condicoes4))
10
```

```
True
True
False
```

Apenas como exemplo, para variável condicoes2 é retornado True pois é composta de um dicionário onde suas duas chaves possuem elementos válidos.

Para variável condicoes3 é retornado True pois ao menos uma das chaves possui em elemento válido.

Por fim, referente a variável condicoes4 é retornado False pois nenhuma chave desse dicionário possui elemento válido.

- all( )

```
1   condicoes = [True, True, True, True, True]
2   condicoes2 = {True: 1, True: 0}
3
4   print(all(condicoes))
5   print(all(condicoes2))
6
```

```
True
True
```

Diferentemente de any( ), a função all( ) somente retornará True caso todos os elementos de uma tupla, lista, conjunto ou dicionário

forem válidos.

```
1   condicoes3 = (1, 'Fernando', True, True)
2
3   condicoes4 = (0, 'Fernando', True, True)
4
5   condicoes5 = (1, 'Fernando', True, False)
6
7   condicoes6 = {True: 1, True: 0}
8
9   print(all(condicoes3))
10  print(all(condicoes4))
11  print(all(condicoes5))
12  print(all(condicoes6))
13
```

```
True
False
False
True
```

Repare nas variáveis deste exemplo, ao aplicar a função all( ) sobre a variável condicoes3 é retornado True, pois a mesma possui todos seus elementos válidos.

Já para variável condicoes4 é retornado False, pois 0 é um elemento interpretado como False, e neste tipo de validação, um dos elementos sendo False já invalida todo o contexto.

Da mesma forma para variável condicoes5 será retornado False pois existe um False declarado como elemento desta variável.

Por fim em condicoes6, quando verificado via all( ) é retornado True pois as duas chaves desse dicionário são elementos válidos. Nesse caso, o 0 atribuído como valor da segunda chave não é levado em consideração.

Asserções

Uma prática comum de todo desenvolvedor é, após implementar certas funcionalidades em seu programa, revisar blocos de código onde temos estruturas de dados que interagem com o usuário, criando validações na tentativa de contornar possíveis erros cometidos pelo usuário.

Como é de conhecimento comum, um programa robusto nunca estará totalmente isento de receber alguma instrução não esperada via usuário e acabar gerando erros, dependendo da gravidade dos mesmos, encerrando processos ou o programa como um todo.

Por parte de estruturas de validação, os meios e métodos convencionais tendem a sujar um pouco o código pois dependendo quantas e quais exceções estamos tentando prever, haverão linhas e códigos dedicados a este propósito.

Visando otimizar a eficiência de nosso código, certas situações podem receber estruturas de validação em forma de asserções.

Uma asserção, de forma resumida, será uma forma reduzida e personalizada de se gerar uma validação, de modo que podemos impor condições a serem validadas e o próprio erro a ser gerado caso a condição importa não for válida, tudo em uma linha de código.

```
1  def cadastro(nome, idade):
2      assert idade >= 18 and idade <= 65, 'Idade do funcionário deve estar entre
3      print(f'{nome} está em idade ativa, pois possui {idade} anos.')
4
5  idade = 55
6  nome = 'Fernando'
7
8  verificador = cadastro(nome, idade)
9
```

```
Fernando está em idade ativa, pois possui 55 anos.
```

Como exemplo, usaremos de uma simples função de nome cadastro( ) que recebe como parâmetros nome e idade.

Logo na primeira linha do bloco dessa função iniciamos nossa asserção com a palavra reservada assert, seguida da proposição lógica a ser validada, nesse caso, que a idade deve ser igual ou maior que 18 ou igual ou menor que 65, por fim, na mesma linha é colocada a mensagem de erro a ser apresentada caso a estrutura condicional anterior não seja válida.

Criadas duas variáveis idade e nome, com seus respectivos atributos, por fim é criada uma última variável de nome verificador que chama a função cadastro( ), parametrizando a mesma com os dados/valores de nome e idade, respectivamente.

O retorno, como esperado, será: Fernando está em idade ativa, pois possui 55 anos.

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-10-82ee38f83fed> in <module>()
      6 nome = 'Fernando'
      7
----> 8 verificador = cadastro(nome, idade)

<ipython-input-10-82ee38f83fed> in cadastro(nome, idade)
      1 def cadastro(nome, idade):
----> 2   assert idade >= 18 and idade <= 65, 'Idade do funcionário deve estar
entre 18 e 65 anos.'
      3   print(f'{nome} está em idade ativa, pois possui {idade} anos.')
      4
      5 idade = 88

AssertionError: Idade do funcionário deve estar entre 18 e 65 anos.
```

Simulando propositalmente um erro, alterando o valor da variável idade para 88, é gerado um código de erro do tipo AssertionError, mostrando em qual linha o interpretador encontrou um erro, assim como na parte final podemos ver a descrição personalizada do erro, que nada mais é do que a própria mensagem que declaramos junto a asserção.

Funções Embutidas / Built-ins

Explorando a documentação oficial da linguagem Python é possível certificar que, de acordo com o jargão “Python vem com pilhas inclusas”, de fato temos à disposição uma série de funções e ferramentas que implementam funcionalidades específicas em nosso código, muitas vezes desconhecidas pelo desenvolvedor.

A seguir, vamos ver em resumo algumas das funções embutidas no núcleo Python mais comumente usadas.

- abs( )

```
1  num_negativo = -276
2
3  print(num_negativo)
4
5  num_absoluto = abs(num_negativo)
6
7  print(num_absoluto)
8
```

```
-276
276
```

A função abs( ), basicamente tem o propósito de retornar um valor numérico em seu formato absoluto, ou seja, pela regra matemática, um número absoluto é sempre positivo, logo, mesmo parametrizando essa função com um valor negativo, o mesmo será convertido em positivo.

```
1  num = abs(-276)
2
3  print(num)
4
```

```
276
```

O número a ser usado pela função abs( ) pode perfeitamente vir do conteúdo de uma variável (como no exemplo anterior) ou ser diretamente repassado como parâmetro da função.

Em ambos casos, o valor retornado é 276.

- ascii( )

```
1  nome = 'Fernando'
2
3  nome_ascii = ascii(nome)
4
5  print(nome)
6  print(nome_ascii)
7
```

```
Fernando
'Fernando'
```

Certos códigos podem ter linguagem de codificação diferente, principalmente em outros idiomas que usam além dos caracteres alfanuméricos conhecidos e praticados nas linguagens ocidentais, caracteres e símbolos que por sua vez podem gerar erros quando transliterados.

Para certas aplicações de código compartilhado entre desenvolvedores de diferentes idiomas, é bastante comum inserir logo na primeira linha do código uma instrução ao interpretador em forma de comentário repassando a codificação a ser usada por todos desenvolvedores, normalmente # _*_ coding: utf-8 _*_ para que como dito anteriormente, se evite a geração de erros por parte de leitura de caracteres considerados inválidos.

Outra prática é a de fazer uso da função ascii( ), parametrizada com o dado/valor em questão, forçando o interpretador a reconhecer esse dado em padrão latino-americano, validando caracteres e/ou símbolos desconhecidos por um valor padrão.

```
1   nome = '网络Fernando'
2
3   nome_ascii = ascii(nome)
4
5   print(nome)
6   print(nome_ascii)
7
```

```
网络Fernando
'\u7f51\u7edcFernando'
```

Apenas exemplificando, observe que na variável nome temos um atributo em forma de string, porém composto por dois caracteres em forma de ideograma chinês.

Validando esta string via ascii( ) e retornando esse novo dado para variável nome_ascii, podemos via função print( ) exibir esse texto convertido.

No lugar de gerar uma exceção, no campo onde estavam situados anteriormente os ideogramas, agora temos um código para cada ideograma representando sua forma não literal, mas código da tabela de caracteres padrão do sistema operacional.

- bytes( )

```
1   tamanho = 8
2   tam_bytes = bytes(tamanho)
3
4   print(tam_bytes)
5   print(type(tam_bytes))
6
```

```
b'\x00\x00\x00\x00\x00\x00\x00\x00'
<class 'bytes'>
```

Para certas situações bastante específicas, pode ser necessário o uso de algum dado/valor em seu formato bytes, logo, via função bytes( ) parametrizada com o referido dado/valor, obteremos como retorno tal dado convertido.

Apenas realizando um pequeno adendo, por meio da função print( ), parametrizada com type( ), por sua vez parametrizada com o dado em questão, podemos inclusive ver de acordo com o retorno que bytes é um tipo de dado em Python.

```
1   nome = 'Fernando'
2   tam_bytes = bytes(nome, 'utf-8')
3
4   print(tam_bytes)
5   print(type(tam_bytes))
6
```

```
b'Fernando'
<class 'bytes'>
```

Uma particularidade que é interessante entender é que, uma vez que bytes é um tipo de dado em Python, este tipo de dado pode ser manipulado por boa parte dos métodos que se aplicam aos outros tipos de dados, desde que esteja na codificação correta de língua.

Por exemplo, em nossa variável tam_bytes, como atributo da mesma temos a função bytes( ) onde por justaposição como primeiro parâmetro é repassado o conteúdo da variável nome, porém convertido para a codificação utf-8.

Note que o retorno gerado é uma b'string, tipo de dado identificado pelo interpretador como uma string do tipo bytes.

```
1   pares = [2, 4, 6, 8, 10]
2   tbytes = bytes(pares)
3
4   print(pares)
5   print(type(pares))
6
7   print(tbytes)
8   print(type(tbytes))
9
```

```
[2, 4, 6, 8, 10]
<class 'list'>
b'\x02\x04\x06\x08\n'
<class 'bytes'>
```

Caso não seja necessário realizar a conversão de língua, o retorno padrão é na notação original de dados tipo bytes. Como exemplo em nossa linha 2 estamos convertendo para bytes o conteúdo da variável pares que por sua vez é uma lista com 5 elementos.

Por meio da função print( ) declarada na linha 7, parametrizada com o conteúdo de tbytes, podemos ver que, como esperado, o conteúdo da lista pares convertido para bytes é b'\x02\x04\x06\x08\n.

```
1  tamanho = 8
2  barr = bytearray(tamanho)
3
4  print(tamanho)
5  print(barr)
6
```

```
8
bytearray(b'\x00\x00\x00\x00\x00\x00\x00\x00')
```

Complementar a função bytes( ) temos a função bytearray( ) que basicamente recebe um dado/valor e retorna o mesmo convertido para array (internamente bytearray).

No exemplo, como atributo da variável tamanho temos o número 8, do tipo int, da mesma forma a variável barr recebe esse dado, pós conversão para bytearray. Em outras palavras, como atributo de barr temos uma array de 8 bytes.

- callable( )

```
1  def soma(n1, n2):
2      return n1 + n2
3
4  print(callable(soma))
5
```

```
True
```

A função callable( ) basicamente é um validador de outras funções, de forma que parametrizamos a mesma com uma função qualquer, de qualquer escopo do código, e caso essa função não contenha nenhum erro é retornado True.

- eval( )

```
1   variavel = 'print(99)'
2
3   print(variavel)
4
5   print(eval(variavel))
6
7
```

```
print(99)
99
None
```

Uma função pouco explorada, porém bastante interessante é a função eval( ), que basicamente avalia uma expressão fora de contexto, normalmente no formato de string, onde caso a expressão seja válida, a mesma será executada mesmo fora de contexto.

Repare na primeira linha do exemplo, temos uma variável de nome variavel que como atributo possui a string ‘print(99)’. Na linha 3 do código via função print( ) parametrizada com o conteúdo de variavel obtemos o retorno literal, o texto print(99). Já na linha 5, nossa função print( ) está parametrizada com a função eval( ) que por sua vez está parametrizada com o conteúdo da variável variavel.

O retorno nesse caso, 99, se dá pois a função eval( ) entendeu o conteúdo da string como uma função print( ) e a executou, retornando de acordo com seu parâmetro o número 99.

```
1   from math import *
2
3   opr = {'raiz_quadrada': sqrt, 'potenciacao': pow}
4
5   print(eval('dir()', opr))
6
7   print(eval('raiz_quadrada(81)', opr))
8
```

```
['__builtins__', 'potenciacao', 'raiz_quadrada']
9.0
```

Usando de um exemplo mais elaborado, logo na linha 3 de nosso código temos uma variável de nome por que atribuído para si tem um dicionário, com dois conjuntos de chaves e valores.

Na linha 5 temos uma função print( ), parametrizada com eval( ), por sua vez parametrizada com a string ‘dir( )’, e como segundo parâmetro justaposto o conteúdo da variável por. Em outras palavras, a função eval( ) entenderá a string, executando a função dir( ) parametrizando a mesma com por.

Já na linha 7, em nossa segunda função print( ), parametrizada com eval( ), parametrizamos a mesma com a string ‘raiz_quadrada( )’, que como chave do dicionário contido em opr, para seu valor terá a função sqrt anteriormente importada da biblioteca math, agora parametrizada com 81.

Como esperado, para primeira linha do retorno temos o retorno referente a dir(opr), da mesma forma, na segunda linha temos o retorno referente a sqrt(81).

- exec( )

```
1   programa = 'n1 = 5\nn2 = 10\nprint("Resultado da soma:", n1 + n2)'
2
3   exec(programa)
4
```

```
Resultado da soma: 15
```

Muito parecido com a função eval( ) temos a função exec( ), sendo a principal diferença entre ambas que exec( ) também possui capacidade de entender o conteúdo e o contexto de uma string, só que suportando duas ou mais expressões dentro da mesma string.

Direto ao ponto note que temos nossa variável programa, e como atributo da mesma a string ‘n1 = 5\nn2 = 10\nprint(“Resultado da soma:”, n1 + n2’.

```
1   programa = 'n1 = 5,
2              n2 = 10,
3              print("Resultado da soma:", n1 + n2)'
4
```

Desmembrando tal string seria o equivalente a este código, porém sabemos que essa sintaxe não é permitida.

Então, o que a função exec( ) fará nesse caso é desmembrar cada sentença entre “ \n “ que é um marcador para quebra de linha, entendendo e executando cada “linha” de código.

Logo, temos 3 expressões a serem lidas e interpretadas, primeira delas criando a variável n1 que tem como atributo 5, na sequência criando a variável n2 que tem o número 10 como atributo, por fim declarando uma função print( ) parametrizada com a string “Resultado da soma:”, seguido do resultado da soma entre n1 e n2.

O retorno, como esperado, é: Resultado da soma: 15

- format( )

```
1   num1 = 128
2   print(type(num1))
3
4   num1 = format(128, 'o') # octal
5   num2 = format(1987, 'x') # hexadecimal
6
7   print(num1)
8   print(type(num1))
9
10  print(num2)
11  print(type(num2))
12
```

```
<class 'int'>
200
<class 'str'>
7c3
<class 'str'>
```

A função format( ) age de duas formas, inicialmente ela altera o identificador de um tipo de dado internamente, de seu formato original para string, em seguida alterando seu tipo temporariamente para um novo tipo de acordo com um marcador, pelo período de uma iteração, onde após a mesma o dado retorna a seu tipo de dado original.

Como exemplo, criamos duas variáveis num1 e num2, que por sua vez fazendo uso da função format( ), parametrizam a mesma com um dado/valor, seguido do segundo parâmetro justaposto que é um identificador do tipo de dado a ser tipado.

Em seguida via função print( ) podemos realizar algumas verificações. Observe que referente a função print( ) da linha 8 temos o retorno 200, ou seja, 128 em sua forma octal, porém referente a função print( ) da linha 9, podemos ver que o tipo de dado de num1 após a iteração (que nesse caso foi a própria ação de ser exibida em tela sua função print( )) retornou ao seu estado original, tendo seu tipo redefinido como str.

Fazendo o mesmo com a variável num2 podemos reparar que a mesma teve seu conteúdo modificado para hexadecimal, em seguida retornando ao formato padrão imposto pela função format( ), nesse caso, str.

```
1   'b' - binário
2   'c' - unicode
3   'd' - decimal
4   'e' - número científico
5   'g' - geral
6   'o' - octal
7   'x' - hexadecimal
8   'n' - número genérico
9   '%' - percentual
10
```

Apenas como referência, segue a lista dos marcadores/identificadores com seus respectivos tipos de dados temporários.

- id( )

```
1   carros = ['Astra', 'Golf', 'Vectra']
2
3   ids = id(carros)
4
5   print(ids)
6
```

```
140596672410888
```

Todo e qualquer dado possui internamente um número identificador, em certas situações pode ser necessário descobrir o número desse objeto, e isto pode ser feito pela função id( ).

Criada a variável carros, sendo a mesma uma lista com 3 elementos, a mesma é instanciada pela função id( ) atribuída a uma variável qualquer, nesse caso ids.

Via função print( ) exibindo o conteúdo da variável ids é possível ver o número identificador do objeto carros.

- iter( )

```
1   cidades = iter(['Curitiba', 'Brasília', 'Florianópolis'])
2
3   cidade1 = next(cidades)
4   print(cidade1)
5
6   cidade2 = next(cidades)
7   print(cidade2)
8
9   cidade3 = next(cidades)
10  print(cidade3)
11
```

```
Curitiba
Brasília
Florianópolis
```

Quando temos contêineres compostos de múltiplos dados e necessitamos iterar sobre apenas um destes isoladamente podemos desempacotar todos e posteriormente rastrear o elemento em questão, ou simplesmente via função iter( ) desmembrar os elementos deste contêiner, iterando isoladamente um por um por meio da função next( ).

Para nosso exemplo, temos uma variável de nome cidades, que por sua vez possui uma lista com 3 nomes de cidades atribuída para si. Aplicando sobre essa lista o método iter( ), internamente é feito o desmembramento de cada elemento da lista tornando o mesmo um elemento independente com uma indexação própria da função iter( ).

A partir deste ponto, usando da função next( ) parametrizada com a variável de origem, nesse caso cidades, a cada ciclo de processamento será retornado apenas um elemento de cidades, quando não houverem mais elementos a função next( ) simplesmente não retornará mais nada.

Para ficar mais evidente o processo, a cada iteração sobre a variável cidades foi criada uma função print( ) exibindo o conteúdo da mesma naquela iteração, e como esperado, inicialmente foi

retornado apenas o primeiro elemento, posteriormente o segundo e assim por diante.

- open( )

```
1   arquivo1 = open('pareceres.txt', 'r')
2
3   print(arquivo1.read())
4
```

Das interações básicas do núcleo Python sobre arquivos em nosso sistema operacional, é possível diretamente via código abrir algum arquivo para realizar a manipulação do mesmo.

Esse processo é feito por meio da função open( ), parametrizada com o nome do arquivo, seguido do segundo parâmetro justaposto que define o método de leitura deste arquivo.

Apenas como exemplo, declarada a variável arquivo1, a mesma chama a função open( ) passando como primeiro parâmetro o nome do arquivo com sua extensão (em certos sistemas operacionais é necessário passar o caminho absoluto até o arquivo), seguido do parâmetro ‘r’ que é um marcador para definir que o arquivo está sendo aberto apenas para leitura.

Por fim, via função print( ), instanciando nossa variável arquivo1 e aplicando sobre a mesma o método read( ) é exibido em tela o conteúdo do arquivo pareceres.txt.

```
1   'r' - apenas leitura
2   'w' - leitura e escrita
3   't' - abre um arquivo em formato apenas texto
4   'b' - abre um arquivo em seu modo binário
5   'x' - cria no diretório um arquivo
6
```

Apenas como referência, segue uma simples relação dos marcadores com suas respectivas funcionalidades.

- slice( )

```
1   nomes = ('Ana', 'Carlos', 'Daniela', 'Elis', 'Fernando', 'Mariana', 'Paulo')
2   primeiros3 = slice(3)
3
4   print(nomes[primeiros3])
5
```

```
('Ana', 'Carlos', 'Daniela')
```

Por meio da função slice( ) é possível fatiar e extrair informação dos elementos de uma tupla, a contar do número de índice 1 até o número especificado.

Nesse caso, importante salientar uma particularidade dessa função, onde a mesma precisa de uma variável temporária que servirá como identificador dos elementos selecionados para a devida leitura dos mesmos.

Em outras palavras, temos nossa variável nomes cujo atributo é uma tupla com 7 nomes de pessoas, na sequência temos uma variável de nome primeiros3 que chama a função slice( ) parametrizando a mesma com o número 3, selecionando e extraindo as informações dos elementos 1, 2 e 3 da tupla.

Para exibir em tela o conteúdo da seleção, por meio da função print( ) parametrizamos a mesma com a variável nomes, usando da notação de número de índice a variável primeiros3. Dessa forma serão retornados os dados em questão.

Funções Matemáticas Específicas

Vamos explorar algumas funções implementadas em revisões recentes das bibliotecas math e statistics, diretamente na prática.

- Distância entre espaço 2D

```
1  import math
2
3  pontoa = [45, 9]
4  pontob = [23, 5]
5
6  print(math.dist(pontoa, pontob))
7
```

Da biblioteca math, temos a função dist( ) que por sua vez tem o propósito de mensurar a distância entre dois pontos.

Em nosso exemplo, declaradas duas variáveis pontoa e pontob, cada um com uma lista de duas coordenadas atribuída para si, podemos calcular a distância entre esses dois pontos. Observe que cada lista temos dois valores, os quais podemos abstrair como eixos X e Y de um plano cartesiano.

Diretamente em nossa função print( ) parametrizada com a função math.dist( ), por sua vez parametrizada com os dados de pontoa e pontob, quando executada retornará a distância entre esses dois pontos.

Nesse caso, dados os valores referenciais, o retorno será: 22.36.

- Distância entre espaço 3D

```
1  import math
2
3  pontoa = (12, 15, 9)
4  pontob = (66, 9, 19)
5
6  print(math.dist(pontoa, pontob))
7
```

De forma muito parecida com o exemplo anterior, usando da função math.dist( ) podemos calcular a distância entre dois pontos.

Agora, se tratando de pontos tridimensionais, podendo serem abstraídos como eixos X, Y e Z, o tipo de dado associado aos pontos deve respeitar a notação de um contêiner.

Executando este bloco de código o retorno será: 55.24.

- Cálculo de hipotenusa

```
1  import math
2
3  objeto3d = (12, 15, 9)
4
5  print(math.hypot(*objeto3d))
6
```

Uma função muito pedida no passado era a de cálculo de uma hipotenusa, no sentido de ter uma função dedicada, sem ter de recorrer a nenhuma gambiarra para realizar tal feito.

Implementada nas últimas revisões da biblioteca math, a função hypot( ) pode receber referências abstraindo base e altura ou coordenadas calculando assim os valores pela norma Euclideana.

No exemplo, a variável objeto3d tem em forma de um contêiner/tupla 3 valores numéricos.

Via função print( ) chamamos a função math.hypot( ) parametrizando a mesma com *objeto3d. Repare que a notação

usando o marcador “ * “ nesse caso serve para desempacotar cada valor do contêiner e aplicar as devidas funções matemáticas individualmente.

Nesse caso, dados os valores de objeto3d, o retorno é 21.21.

- Média de valores (números reais)

```
1  import statistics
2
3  num_reais = [3.5, 4.4, 9, 12.1, 7.9]
4
5  print(statistics.fmean(num_reais))
6
```

Se tratando da biblioteca statistics, em suas últimas revisões foram implementadas diversas funções estatísticas específicas, vamos entender na prática duas delas mais comumente usadas.

Em nosso exemplo temos uma variável de nome num_reais, que por sua vez tem como atributo uma lista com uma série de números, tanto do tipo int quanto do tipo float.

Em nossa função print( ), parametrizada com statistics.fmean( ), parametrizamos a mesma com os valores contidos em num_reais, sendo assim retornada a média destes valores, nesse caso, 7.38

- Média geométrica

```
1  import statistics
2
3  num_reais = [3.5, 4.4, 9, 12.1, 7.9]
4
5  print(statistics.geometric_mean(num_reais))
6
```

Outra função bastante útil é a de cálculo de média geométrica, agora pronta, bastando fornecer os valores. Lembrando que diferentemente do cálculo da média convencional, uma média geométrica é aquela que multiplica todos os elementos realizando em seguida a média deles em relação a raiz destes elementos.

Em outras palavras, a média geométrica de três elementos é o resultado da multiplicação dos mesmos, calculando em seguida a raiz cúbica (referente a 3 elementos) e retornando esse valor.

No exemplo, usando dos mesmos valores do exemplo anterior, o resultado da média geométrica é 6.67

- Valor mais frequente

```
1   import statistics
2
3   num_reais = [7, 3, 5, 9, 7, 2, 2, 4, 1, 8, 7, 0, 7]
4
5   print(statistics.multimode(num_reais))
6
```

Por meio da função statistics.multimode( ) podemos fornecer um conjunto de elementos para o mesmo, que irá ler todos elementos retornando o mais frequente deles.

Nesse caso, o elemento mais recorrente nos elementos da lista atrobuída a num_reais foi o número 7.

```
1   import statistics
2
3   exame = 'Histerossalpingografia Exploratória'
4
5   print(statistics.multimode(exame))
6
```

A função multimode( ) também suporta strings, onde internamente a mesma desmembrará elemento por elemento (caractere por caractere), retornando o mais frequente dos mesmos.

Nesse caso, associada a variável exame, uma string ‘Histerossalpingografia Exploratória’, que apenas por curiosidade, é um exame de raios-x contrastado para estudo do útero, trompas e ovários, quando passada como parâmetro para função statistics.multimode( ) o retorno gerado é ‘a’, que neste nome de exame aparece 5 vezes.

Metadados

A partir do momento em que estamos começando a estudar sobre modularizações, bibliotecas, módulos e pacotes de terceiros, etc... nos focamos completamente em importar tais recursos e partir para implementação de suas ferramentas, passando desapercebido alguns pontos importantes no que diz respeito a buscar conhecer os chamados metadados, tópicos de informação sobre a biblioteca/módulo em si, seus recursos, ferramentas, informações provindas pelos próprios criadores, etc...

Porém, existem alguns destes pontos que são importantes se ter conhecimento diretamente da fonte e não de dados especulativos de usuários.

```
1  #pip install importlib
2  from importlib import metadata
3
```

Inicialmente para fazer consultas em metadados é necessário instalar a biblioteca importlib caso a mesma não esteja instalada no ambiente virtualizado onde se encontra seu núcleo Python. Logo após, realizamos a importação do módulo metadata da biblioteca importlib.

```
1  from importlib import metadata
2
3  versao_numpy = metadata.version('numpy')
4
5  print(versao_numpy)
6
```

Criada uma variável, nesse caso de nome versao_numpy, que chama do módulo metadata a função version( )por sua vez parametrizada com o nome de outra biblioteca em formato de string, é possível ver a versão exata da biblioteca instalada em seu sistema.

```
metadados_numpy = metadata.metadata('numpy')

print(list(metadados_numpy))

```

De modo parecido, agora para uma variável de nome metadados_numpy atribuímos a função metadata.metadata( ) também parametrizada com ‘numpy’.

O retorno nesse caso será uma lista com todos metadados disponíveis para a atual versão da biblioteca Numpy.

['Metadata-Version', 'Name', 'Version', 'Summary', 'Home-page', 'Author', 'Maintainer', 'Maintainer-email', 'License', 'Download-URL', 'Project-URL', 'Project-URL', 'Project-URL', 'Platform', 'Platform', 'Platform', 'Platform', 'Platform', 'Classifier', 'Classifier', 'Classifier', 'Classifier', 'Classifier', 'Classifier', 'Classifier', 'Classifier', 'Classifier', 'Classifier', 'Classifier', 'Classifier', 'Classifier', 'Classifier', 'Classifier', 'Classifier', 'Classifier', 'Classifier', 'Requires-Python']

```
print(metadados_numpy)

print(metadados_numpy['Maintainer-email'])

```

Escolhendo um metadado em particular, e exibindo seu conteúdo por meio da função print( ), por exemplo, o metadado ‘Maintainer-email’ obteremos como retorno numpy-discussion@python.org.

```
print(len(metadata.files('numpy')))

```

Podemos perfeitamente aplicar os métodos mais comumente usados sobre os arquivos de uma biblioteca, por exemplo via função len( ) podemos verificar o numero de arquivos que compõe o pacote. Nesse caso retornado o valor de 1064.

```
print(metadata.files('numpy'))

```

Reaproveitando a linha de código anterior, apenas removendo da mesma a função len( ), nos é retornado todos arquivos dispostos

no pacote como um todo.

[PackagePath('../../Scripts/f2py.exe'), PackagePath('numpy-1.20.1.dist-info/INSTALLER'), PackagePath('numpy-1.20.1.dist-info/LICENSE.txt'), PackagePath('numpy-1.20.1.dist-info/LICENSES_bundled.txt'), PackagePath('numpy-1.20.1.dist-info/METADATA'), PackagePath('numpy-1.20.1.dist-info/RECORD'), PackagePath('numpy-1.20.1.dist-info/REQUESTED'), PackagePath('numpy-1.20.1.dist-info/WHEEL'), PackagePath('numpy-1.20.1.dist-info/entry_points.txt'), PackagePath('numpy-1.20.1.dist-info/top_level.txt'), PackagePath('numpy/.libs/libopenblas.JPIJNSWNNAN3CE6LLI5FWSPHUT2VXMTH.gfortran-win_amd64.dll'), PackagePath('numpy/LICENSE.txt'), PackagePath('numpy/__config__.py'), PackagePath('numpy/__init__.cython-30.pxd'), PackagePath('numpy/__init__.pxd'), PackagePath('numpy/__init__.py'), PackagePath('numpy/__init__.pyi'), PackagePath('numpy/__pycache__/__config__.cpython-38.pyc'), PackagePath('numpy/__pycache__/__init__.cpython-38.pyc'), PackagePath('numpy/__pycache__/_distributor_init.cpython-38.pyc'), PackagePath('numpy/__pycache__/_globals.cpython-38.pyc'), PackagePath('numpy/__pycache__/_pytesttester.cpython-38.pyc'), PackagePath('numpy/__pycache__/conftest.cpython-38.pyc'), PackagePath('numpy/__pycache__/ctypeslib.cpython-38.pyc'), PackagePath('numpy/__pycache__/dual.cpython-38.pyc'), PackagePath('numpy/__pycache__/matlib.cpython-38.pyc'), PackagePath('numpy/__pycache__/setup.cpython-38.pyc'), PackagePath('numpy/__pycache__/version.cpython-38.pyc'), PackagePath('numpy/_distributor_init.py'), PackagePath('numpy/_globals.py'), PackagePath('numpy/_pytesttester.py'), PackagePath('numpy/char.pyi'),

...

```
print(metadata.requires('numpy'))

```

Por fim, mas não menos importante, por meio da função metadata.requires( ) é possível visualizar todas as dependências de uma biblioteca. A primeira vez que esta linha de código for executada, retornará alguns nomes de alguns módulos e pacotes com suas referidas versões.

Ao tentar executar novamente esta mesma linha é gerado um retorno None, significando que todas dependências para o pleno funcionamento da biblioteca Numpy estão devidamente instaladas.

Erros

Uma das partes ao mesmo tempo mais frequentes, e menos exploradas, são os códigos de erro gerados quando alguma exceção é cometida por algum bloco de código, sendo um erro tão grave a ponto do interpretador não conseguir o contornar, acabando por encerrar a execução do programa em função do erro.

Basicamente existem uma série de erros que podem ser cometidos pelo desenvolvedor, que principalmente enquanto estudante costuma não os dar muita atenção, simplesmente vendo que um erro foi gerado e trabalhando para o corrigir, ou em contextos mais avançados, criar as devidas validações para que o erro seja contornado.

O fato é que, ao menos em Python, erros são mensagens de retorno muito bem organizadas que nos fornecem ao menos dois dados cruciais, primeiro, em que linha o erro foi encontrado, segundo, que tipo de erro foi cometido. Se tratando dos tipos de erros mais comuns, é interessante entendermos resumidamente o que foi considerado erro e como contornar o mesmo.

- SintaxError

```
1   num1 47
2

 File "<ipython-input-4-6994c1d04998>", line 4
   num1 47
         ^
SyntaxError: invalid syntax
```

Erro de sintaxe é o erro gerado ao tentar passar uma instrução usando de um comando ou “gramática” não reconhecida pelo interpretador.

No exemplo, toda variável deve usar do operador de atribuição “ = “ para associar algum dado/valor para si própria. A forma correta seria: num1 = 47

```
1   num1 = 47
2
3   print(num1 -> 10)
4
```

```
  File "<ipython-input-7-9967fb4735da>", line 3
    print(num1 -> 10)
                 ^
SyntaxError: invalid syntax
```

O uso de símbolos não reconhecidos pelo interpretador ou usados em local errado também geram erros de sintaxe. A forma correta de se comparar se o valor da variável num1 é maior que 10 seria: print(num1 > 10)

```
1   num1 = 47
2
3   if num1 > 10
4   print('Maior que 10')
5
```

```
  File "<ipython-input-8-4e139dc49e2f>", line 3
    if num1 > 10
                ^
SyntaxError: invalid syntax
```

Expressões que possuem blocos de códigos filhas dessa instrução devem possuir o marcador “ : “ para que seja identificado o final da linha de instrução. Todo código pertencente a essa instrução deve estar indentado (tabulado) para ser reconhecido pelo interpretador corretamente.

No exemplo, toda estrutura condicional deve iniciar com if, seguido da proposição imposta para ser verificada e validade, finalizando com “ : “.

A forma correta seria:
if num1 > 10:
    Print(‘Maior que 10’)

```
1   lista = [1, 3, 5, 7, 9, 2, 4, 6, 8]
2
3   for num lista:
4       print(num)
5
```

```
  File "<ipython-input-10-e0209a9d7ad9>", line 3
    for num lista:
                 ^
SyntaxError: invalid syntax
```

Todo laço de repetição precisa de uma variável temporária definida, nesse caso num, assim como na sequência deve ser usado de um operador que indique onde iterar a cada ciclo de repetição, nesse caso dentro do conteúdo da variável lista.

A forma correta seria:
for num in lista:
    print(num)

- NameError

```
1   print(var_num47)
2
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-5-8d920ee18c18> in <module>()
----> 1 print(var_num47)

NameError: name 'var_num47' is not defined
```

Todo objeto em Python deve ter um nome para que o mesmo possa ser instanciável. Nesse caso, a função print( ) precisa conseguir ler o conteúdo da variável var_num47 para conseguir exibir em tela seu conteúdo. Como a variável em si não está declarada no código, será gerado um erro.

- TypeError

```
1   num1 = 47 # int
2   num2 = '48' # string
3
4   print(num1 + num2)
5
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-6-12fef66bddf1> in <module>()
      2 num2 = '48' # string
      3
----> 4 print(num1 + num2)

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Um erro de tipo é aquele erro que será gerado quando tentarmos realizar alguma operação entre tipos de dados não equivalentes. Nesse caso, note que o erro gerado é em função de que na função print( ) foi tentado realizar a soma entre um número e um texto.

```
1   class Pessoa():
2       def msg():
3           print('Olá!')
4
5   visitante = Pessoa()
6   visitante.msg()
7
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-35-c653be488131> in <module>()
      8
      9 visitante = Pessoa()
---> 10 visitante.msg()

TypeError: msg() takes 0 positional arguments but 1 was given
```

Em uma estrutura de dados orientada a objetos, é muito comum um erro ser gerado quando um parâmetro/argumento ou atributo de classe não é um objeto válido.

No exemplo, como self não foi declarado, o próprio método de classe foi considerado como argumento, gerando um erro.

```
1   abs('Olá!')
2

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-36-ba2c7d01f3d4> in <module>()
      2 # em uma variável de tipo de dado não suportado.
      3
----> 4 abs('Olá!')

TypeError: bad operand type for abs(): 'str'
```

Usar de uma função que itera apenas sobre dados numéricos em um texto, certamente irá gerar um erro.

- IndentationError

```
1   if 10 > 5:
2   print('10 é maior que 5')
3

  File "<ipython-input-26-a7c9e3c6a4ad>", line 5
    print('10 é maior que 5')
        ^
IndentationError: expected an indented block
```

Pela leitura léxica realizada pelo interpretador, todo código é lido de cima para baixo e cada linha é lida da esquerda para direita, considerando inclusive as tabulações e espaços.

Em nosso exemplo existe uma estrutura condicional porém o interpretador não conseguirá identificar qual seria o bloco de código filho dessa instrução, gerando um erro.

A forma correta seria:
if 10 > 5:
    print(’10 é maior que 5’)

- IndexError

```
1   lista1 = ['Ana', 'Bia', 'Eli', 'Iuri']
2   print(lista1[4])
3

---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-7-272bd8c70fde> in <module>()
      1 lista1 = ['Ana', 'Bia', 'Eli', 'Iuri']
----> 2 print(lista1[4])

IndexError: list index out of range
```

Todo contêiner em Python, ou em outras palavras, todo tipo de dado que guarda como atributo em sua estrutura múltiplos dados/valores, possui seus elementos internamente indexados. Ao tentar buscar um elemento que não existe no índice, é gerado um erro.

- KeyError

```
1   nomes = {'Paulo': 'pai',
2            'Alice': 'mãe',
3            'Guilherme': 'filho'}
4
5   print(nomes['Miguel'])
6

---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-30-e828da1ee6f6> in <module>()
      5          'Guilherme': 'filho'}
      6
----> 7 print(nomes['Miguel'])

KeyError: 'Miguel'
```

Dicionários são estruturas de dados que possuem a particularidade de guardar seus dados em conjuntos de chaves e

valores, obrigatoriamente tendo os dois dados associados entre si. Ao tentar buscar um item não existente no dicionário, será gerado um erro.

- ZeroDivisionError

```
1   num1 = 99
2   num2 = 0
3
4   print(num1 / num2)
5

---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-12-c70eed83d334> in <module>()
      5 num2 = 0
      6
----> 7 print(num1 / num2)

ZeroDivisionError: division by zero
```

Pela regra matemática, é impossível realizar uma divisão por zero, logo, ao tentar realizar uma operação dessas no código, será gerado um erro.

- FileNotFoundError

```
1   base = open('planilha.csv', 'r')
2

---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
<ipython-input-8-01b0ddfdf333> in <module>()
----> 1 base = open('planilha.csv', 'r')

FileNotFoundError: [Errno 2] No such file or directory: 'planilha.csv'
```

Ao tentar abrir um arquivo inexistente (ou com caminho declarado errado), é gerado um erro.

- RunTimeError

Será todo erro crítico, ou em outras palavras, será todo erro grave ao ponto do interpretador não conseguir contorna-lo, resultando no encerramento da execução do programa.

- OSError

```
1   import os
2
3   print(os.ttyname(1))
4
```

```
---------------------------------------------------------------------------
OSError                                   Traceback (most recent call last)
<ipython-input-15-49b724290632> in <module>()
      4 import os
      5
----> 6 print(os.ttyname(1))

OSError: [Errno 25] Inappropriate ioctl for device
```

Um OSError será todo e qualquer erro gerado ao tentar sem sucesso interagir com alguma funcionalidade do sistema operacional.

No exemplo, um erro é gerado pois em um ambiente virtualizado comum é esperado que se tenha apenas um terminal ativo dedicado.

Pip

A linguagem Python é de fato uma linguagem muito robusta, composta por um núcleo onde temos uma enorme quantidade de bibliotecas, módulos e pacotes que nada mais são do que ferramentas que implementam funcionalidades em nossos códigos.

Python possui uma comunidade gigante e muito ativa, o que resulta que muitas das funcionalidades que hoje temos nativas na linguagem surgiram de propostas dos próprios usuários que inicialmente criaram tais ferramentas e após uma série de testes e aperfeiçoamento, as mesmas foram incorporadas no núcleo da linguagem.

Da mesma forma, com uma rápida pesquisa é possível ver uma quantidade incontável de bibliotecas, módulos e pacotes que foram desenvolvidas por terceiros e que constantemente estão sendo melhoradas.

Bibliotecas estas que implementam ainda mais funcionalidades para nossa tão querida linguagem de programação. Apenas como simples exemplo, temos à disposição bibliotecas específicas para visualização de dados e plotagem de gráficos como a MatPlotLib ou a SeaBorn, funções matemáticas aplicadas a objetos escalares, vetores e matrizes pelas bibliotecas Numpy e Pandas, métricas e funções de matemática científica pela SciKitLearn, aprendizado de máquina via TensorFlow e Keras, visão computacional pela OpenCV (biblioteca esta nativa de outras linguagens, totalmente adaptada para Python), etc... Para praticamente todo tipo de aplicação temos um recurso ou nativo da linguagem Python ou implementável por meio de uma biblioteca externa.

O processo de instalar uma nova biblioteca em nossa linguagem Python é bastante simples, normalmente feito via Pip, um gerenciador de pacotes que realiza o download e instalação de todas dependências no ambiente virtualizado que estivermos trabalhando.

```
1   https://pypi.org/project/pip/
2
```

A documentação oficial do instalador e gerenciador de pacotes Pip pode ser acessada pela página da mantenedora do projeto através do link acima.

```
1   pip install camelcase
2
```

```
Collecting camelcase
  Downloading https://files.pythonhosted.org/packages/24/54/6bc20bf371
Building wheels for collected packages: camelcase
  Building wheel for camelcase (setup.py) ... done
  Created wheel for camelcase: filename=camelcase-0.2-cp36-none-any.wh
  Stored in directory: /root/.cache/pip/wheels/b1/fe/08/84d2143069bc44
Successfully built camelcase
Installing collected packages: camelcase
Successfully installed camelcase-0.2
```

Como principal funcionalidade do Pip, ou ao menos a mais comumente utilizada, temos o próprio instalador de pacotes da mesma, acessível por qualquer terminal.

Via terminal digitando pip install, seguido do nome do pacote da biblioteca, é instalada a referida biblioteca no ambiente virtualizado padrão do sistema operacional.

No exemplo, por meio do comando pip install camelcase é instalada a biblioteca CamelCase, que oferece uma série de ferramentas para manipulação de textos.

```
1   pip uninstall camelcase
2
```

Da mesma forma, é possível desinstalar uma biblioteca que por algum motivo esteja gerando algum conflito ou problema. O processo é feito por meio do comando pip uninstall seguido do nome da biblioteca.

```
1   import camelcase
2
```

Quando estamos começando a estudar sobre bibliotecas, módulos e pacotes, além é claro dos mecanismos de modularização de nossos códigos, é natural começarmos a importar ferramentas de bibliotecas para usar de recursos específicos das mesmas.

Logo nas primeiras linhas de nosso código, por convenção, realizamos as devidas importações das bibliotecas, módulos e pacotes que utilizaremos ao longo de nossos códigos.

Bastando escrever import camelcase, por exemplo, todo conteúdo dessa biblioteca será carregado e estará disponível para uso irrestrito.

```
1   from camelcase import CamelCase
2
```

À medida que os desenvolvedores vão implementando novos recursos e funcionalidades a uma biblioteca, tornando a mais robusta, é natural que pro uma questão de organização cada categoria de funcionalidades da mesma seja modularizada.

Além da questão de organização, a modularização desses pacotes internos a uma biblioteca nos permite fazer uma coisa interessante. Uma vez que temos um pacote dentro de uma biblioteca, podemos especificamente importar somente o conteúdo deste pacote no lugar de toda a biblioteca.

Raciocine que uma biblioteca robusta pode ter uma grande quantidade de arquivos internos, carrega-los sem necessidade acarreta em uma perda de performance, já quando importamos apenas o necessário de uma biblioteca, esse impacto na performance é mínimo.

No exemplo, por meio do comando from camelcase import CamelCase, estamos importando apenas o módulo CamelCase da biblioteca camelcase.

```
1   from camelcase import CamelCase as cc
2
```

Uma prática comum adotada por boa parte dos desenvolvedores é referenciar o conteúdo importado por meio de

uma abreviação, desse modo quando instanciarmos este objeto em uma determinada linha de código, podemos o fazer pelo seu “apelido”.

Por exemplo, from camelcase import CamelCase as cc significa da biblioteca camelcase importe o módulo CamelCase e o referencie como cc.

```
1   nome = 'fernando'
2
3   conversor = cc()
4
5   print(conversor.hump(nome))
6
```

```
Fernando
```

A partir desse ponto sempre que formos fazer o uso do módulo CamelCase, no código o chamaremos simplesmente por cc( ).

Gerenciamento de Memória

Um dos pontos mais importantes em programação é o chamado gerenciamento de memória, muitas vezes negligenciado por parte do desenvolvedor, ou apenas não levado em conta pelo mesmo, uma vez que este parâmetro é de fato relevante de acordo com o tipo de aplicação sendo desenvolvida.

O fato é que todo e qualquer programa, dependendo de sua complexidade, terá uma determinada performance dependendo do hardware onde o qual está sendo executado, mas antes mesmo disso, deve ser considerado em sua estrutura se o mesmo está otimizado o suficiente.

Se você joga algum jogo certamente deve ter notado que conforme a complexidade em melhorias gráficas e física simulada foram avançando ao longo dos anos, os jogos se tornaram mais “pesados”, exigindo hardwares especializados para tal tipo de processamento, problema também muito comum para profissionais de computação gráfica, engenharia, ciência de dados, etc... Áreas onde as ferramentas de trabalho são softwares bastante complexos que exigem de hardware dedicado.

Em nossos programas, por mais básicos ou até complexos que sejam, sempre devemos levar em consideração o que pode ser feito e como pode ser feito para que as funcionalidades de nosso programa tenham o mínimo de impacto na performance do computador.

Assim como em um jogo, é uma boa prática de programação, após o básico de nosso programa estar pronto e funcional, realizar uma série de polimentos sobre o mesmo visando otimizar o código, o que refletirá diretamente na performance de execução do mesmo.

Dada esta introdução, eis que chegamos no ponto de interesse desse tópico, onde entenderemos de forma descomplicada como é feito o gerenciamento de memória para nossos programas.

Sabemos que uma variável/objeto é um objeto alocado em memória que serve para guardar um certo dado/valor. Em Python tudo é objeto, até mesmo funções, embora esse conceito pareça muito abstrato, internamente são objetos.

Também é comum abstrairmos o conceito de variável para algo tangível, apenas como exemplo, como um móvel com várias gavetas, porém com um número limitado de gavetas. Sempre que necessário guardamos um objeto dentro de uma gaveta, quando vamos fazer uso do mesmo abrimos esta gaveta, o pegamos, usamos e devolvemos para a gaveta. Quando o uso deste objeto não se fizer mais necessário, o removemos da gaveta, reutilizando do mesmo espaço da mesma para guardar outro novo objeto.

Uma memória de computador funciona de certo modo funciona dessa maneira, logo, por pura lógica, quanto mais objetos carregados em memória, realizando operações entre si, mais memória é dedicada e consumida em tempo real por nosso programa.

Uma das camadas de processamento, onde nosso programa interage com recursos do sistema operacional, é feito pelo chamado gerenciador de memória. Em sua estrutura lógica ele mapeia e gerencia todo e qualquer objeto carregado em memória com duas instancias, uma stack layer, que serve como mapeamento e indexação do objeto e suas interações, e uma heap layer que é de fato o espaço lógico ocupado pelo objeto.

Linguagens modernas como Python, pensando nesse quesito como um problema computacional, tendem a ter uma engenhosa solução para gerenciar a memória do computador e fazer com que nossos programas sejam executados de forma eficiente.

```
1   # Meu Programa
2   variavel1 = 1987
3
```

Partindo para prática, em nosso programa temos uma variável/objeto de nome variavel1, cujo atributo é o número 1987 do tipo int.

Nossa variavel1 estará situada na stack layer, como uma referência mapeada e indexada para um objeto abstrato que estará carregado na memória, alocando espaço para guardar um dado/valor a ser utilizado.

1987, atributo da variável/objeto variavel1, estará em um espaço de memória alocado na memória heap, sendo assim um objeto que pode ser instanciado e manipulado pelo programa.

Cada vez que um objeto tem seu atributo modificado por uma função, é criado um novo stack com o mesmo objeto, porém atributo diferente.

Em outras palavras, um dado/valor da memória heap pode ser compartilhado por vários objetos da memória stack, mas o contrário não é possível, pois cada objeto que instancia e interage sobre um dado/valor alocado em memória é uma stack única.Ou de forma ainda mais resumida, um objeto situado na memória heap pode ter várias indexações, uma para cada objeto da memória stack que interage com o mesmo.

```
1   num1 = 1987
2
3   num2 = 237
4   num3 = num2
5
6   print(num2)
7   print(type(num2))
8   print(id(num2))
9
10  print(num3)
11  print(type(num3))
12  print(id(num3))
13
```

```
237
<class 'int'>
10922048
237
<class 'int'>
10922048
```

Em nosso exemplo, a variável num1 é um objeto alocando um espaço em memória para guardar seu dado/valor. Da mesma forma num2 é outro objeto totalmente independente, com seu respectivo espaço alocado em memória para seu dado/valor atribuído.

Já para a variável num3, que tem seu dado/valor igual ao dado/valor atribuído para num2, terá um comportamento diferente, pois aqui temos dois objetos usando de apenas um dado alocado em memória.

Para entender de fato como isso internamente ocorre, por meio da função print( ), vamos mostrar o conteúdo das variáveis num2 e num3, assim como os tipos de dados das mesmas e seus identificadores internos de memória por meio da função id( ).

Repare que para num2 temos 237, class ‘int’ e 10922048, enquanto para num3 temos 237, class ‘int’ e o mesmo número identificador 10922048.

A camada onde estamos trabalhando via código, possui todos esses objetos separados e mapeados como tal, internamente, os

objetos que tinham o mesmo dado/valor atribuído, por uma questão de performance, guardavam esse dado/valor em um espaço alocado de memória, disponível para os dois objetos que interagem com o mesmo.

```
1   num1 = 1987
2   num2 = 237
3   num3 = num2
4
5   print(id(num1))
6   print(id(num2))
7   print(id(num3))
8
```

```
139861147165840
10922048
10922048
```

Exibindo o identificador dos três objetos do exemplo anterior, podemos ver que num1 possui um identificador, enquanto num2 e num3 possuem outro, o mesmo, pois apontam para o mesmo objeto alocado em memória.

```
1   n1 = 1987
2   n2 = 1987
3
4   if id(n1) == id(n2):
5       print(f'{n1} e {n2} São o mesmo dado alocado em memória.')
6   else:
7       print(f'{n1} e {n2} possuem dados alocados em memória diferentes.')
8
```

```
1987 e 1987 possuem dados alocados em memória diferentes.
```

```python
n1 = 2000
n2 = 1987

if id(n1) == id(n2):
    print(f'{n1} e {n2} São o mesmo dado alocado em memória.')
else:
    print(f'{n1} e {n2} possuem dados alocados em memória diferentes.')

```

```
2000 e 1987 possuem dados alocados em memória diferentes.
```

```python
n1 = 1987
n2 = n1

if id(n1) == id(n2):
    print(f'{n1} e {n2} São o mesmo dado alocado em memória.')
else:
    print(f'{n1} e {n2} possuem dados alocados em memória diferentes.')

```

```
1987 e 1987 São o mesmo dado alocado em memória.
```

Apenas mostrando uma particularidade que pode gerar confusão, quando estamos falando de que duas ou mais variáveis podem ter o mesmo objeto alocado em memória, estamos falando que estas variáveis precisam ser identificadas pelo interpretador como objetos em stack que apontam para o mesmo objeto heap.

Repare nos exemplos acima. No primeiro bloco temos duas variáveis n1 e n2 que apesar de possuir como atributo o número 1987, possuem seus referentes objetos alocados em espaços de memória diferentes, pois o interpretador mapeia essas variáveis de modo que podemos interagir com cada uma delas de forma independente, inclusive alterando seu atributo sem quebrar sua indexação em memória.

No segundo bloco temos as duas variáveis, com dois atributos diferentes, logo, dois espaços alocados em memória como esperado.

Por fim, no terceiro bloco, n1 tem seu atributo 1987 e n2 tem como seu atributo n1. Em outras palavras, o valor de n2 será o

mesmo valor de n1, logo, se n1 tiver seu valor alterado, consequentemente irá alterar junto o valor de n2. Como essas duas variáveis estão atreladas, possuindo o mesmo dado/valor, estas sim serão dois objetos slack que instanciam apenas um objeto heap em memória.

```
1   num1 = 20
2
3   def soma(num1):
4       num2 = num1 + 10
5       return num2
6
7   calculo = soma(num1 = 10)
8
9   print(f'Identificador do objeto num1: {id(num1)}')
10  print(f'Identificador do objeto num2: {id(num2)}')
11  print(f'Identificador do objeto soma: {id(calculo)}')
12
```

```
Identificador do objeto num1: 10915104
Identificador do objeto num2: 10922048
Identificador do objeto soma: 10915104
```

Partindo para um exemplo um pouco mais complexo, vamos analisar o código acima.

Inicialmente é declarada uma variável de nome num1 que possui como atributo o número 20.

Na sequência temos uma função de nome soma( ) que recebe como parâmetro num1 e seu conteúdo. Como mencionado anteriormente, uma função também é um objeto em Python, e o comportamento aqui será o seguinte:

Da maneira como soma( ) foi declarada, está atrelada a num1, e caso o retorno dessa soma seja igual ao dado/valor de num1 e igual especificamente ao número 20, ambos objetos (num1 e soma) estarão instanciando o mesmo dado/valor, objeto alocado em memória. Caso o retorno de soma for diferente do dado/valor de num1, ou diferente de 20 (valor inicialmente alocado em memória

por num1), será criado um novo objeto slack com outro objeto heap em memória.

Dentro do corpo da função soma( ) é criada uma nova variável de nome num2 que recebe o dado/valor de num1 e soma o mesmo com 10, resultando 20, que é retornado pela função.

Nesse caso, note que o id de num1 e de soma são o mesmo, pois os dados/valores foram todos validados como iguais, logo, apontarão para o mesmo objeto alocado em memória.

```
1   num1 = 20
2
3   def soma(num1):
4       num2 = num1 + 15
5       return num2
6
7   calculo = soma(num1 = 10)
8
9   print(f'Identificador do objeto num1: {id(num1)}')
10  print(f'Identificador do objeto num2: {id(num2)}')
11  print(f'Identificador do objeto soma: {id(calculo)}')
12
```

```
Identificador do objeto num1: 10915104
Identificador do objeto num2: 10922048
Identificador do objeto soma: 10915264
```

Agora, dentro do corpo da função soma( ), alterando o atributo de num2, que nesse caso recebe o dado/valor de num1 (10) somando com 15, resultando 25, o que ocorre é que agora num1 tem um valor, num2 tem outro e o retorno de soma tem outro, todos diferentes entre si.

Embora os objetos num1 e soma estejam atrelados, os mesmos possuem dados/valores diferentes, consequentemente apontarão para objetos diferentes alocados em memória.

```python
class Pessoa:
    def __init__(self, idade):
        self.idade = idade

    def att_idade(self):
        return self.idade

pessoa1 = Pessoa(18)

pessoa2 = pessoa1

```

Em uma estrutura orientada a objetos, é o convencional uma classe ter atributos de classe e métodos de classe, porém haverá uma identificação e indexação interna gerada pelo interpretador, onde quando a classe deixar de servir como molde para um objeto do escopo global do código, internamente os atributos de classe iniciais serão classificados como dead object, permanecendo em memória por alguns ciclos e, caso o interpretador identifique que nenhum objeto voltou a instanciar esse objeto sinalizado como dead object, o mesmo será removido da memória.

Esse mecanismo, chamado garbage collector (em tradução livre coletor de lixo), irá remover/desalocar o objeto da memória, zerando todas as referências ao mesmo, liberando espaço para que um novo objeto vindo de qualquer outra estrutura de dado possa vir a alocar e ocupar esse espaço em memória heap.

```python
class Pessoa:
    def __init__(self, idade):
        self.idade = idade

    def att_idade(self):
        return self.idade

pessoa1 = Pessoa(18)

pessoa2 = pessoa1

```

Importante salientar que aqui o laço que faz com que variáveis/objetos estejam entrelaçados é ainda mais forte. Ainda mais complexo se comparado com o exemplo anterior da ligação entre num1, num2 e soma( ), em uma estrutura orientada a objetos como a do exemplo acima, a classe, o atributo de classe, o dado/valor atribuído ao objeto atributo de classe e a variável que instancia a classe devem possuir o mesmo valor para que seu identificador slack aponte para o mesmo objeto alocado em memória heap.

```
1  print(id(pessoa1))
2  print(id(pessoa2))
3
4
```

```
139861137928264
139861137928264
```

Além disso, qualquer nova variável/objeto do escopo global do código que estender as propriedades de outra, como no exemplo pessoa2 que estende todas as características de pessoa1, enquanto todas suas referências possuírem os mesmos valores/atributos de pessoa1, terão o mesmo número identificador pois apontarão para o mesmo objeto alocado em memória.

Nesse exemplo, pessoa1 e pessoa2 são idênticos, logo, possuem o mesmo identificador e apontam para o mesmo objeto alocado em memória.

```
1   class Pessoa:
2       def __init__(self, idade):
3           self.idade = idade
4
5       def att_idade(self):
6           return self.idade
7
8   pessoa1 = Pessoa(18)
9
10  pessoa2 = pessoa1
11
12  pessoa3 = Pessoa(33)
13
```

Agora incluída a variável pessoa3 que instanciou a classe Pessoa, repassando como atributo para o objeto idade o número 33, temos 3 variáveis apontando para dois objetos alocados em memória. Sendo que destes objetos, pessoa1 e pessoa2 possuem o mesmo identificador.

```
1   pessoa1 = Pessoa(18)
2   pessoa2 = pessoa1
3   pessoa3 = Pessoa(33)
4
5   print(id(pessoa1))
6   print(id(pessoa2))
7   print(id(pessoa3))
8
9   import sys
10
11  print(sys.getrefcount(pessoa1))
12  print(sys.getrefcount(pessoa2))
13  print(sys.getrefcount(pessoa3))
14
```

```
139861137902672
139861137902672
139861130677664
3
3
2
```

Para finalizar nossa linha de raciocínio, ainda usando como referência o código do exemplo anterior, vamos buscar intender na prática como se dá a indexação interna que serve como referência para o garbage collector do núcleo Python.

Basicamente existem de fato duas indexações internas, uma para todo e qualquer objeto de nosso código, outra para as instancias de quantos objetos que estão usando do mesmo objeto alocado em memória. Quando o interpretador identifica que um objeto em memória deixou de ser instanciado por um objeto no código, fica monitorando o mesmo esperando se o mesmo deixará de ser utilizado por completo, sendo assim encaminhado para o coletor de lixo.

Do exemplo acima, temos apenas três instâncias, sendo duas apontando para o mesmo objeto como mencionado anteriormente.

- Verificando o tempo de execução de um código

```python
from timeit import timeit as t
from collections import namedtuple

class Pessoa:
    def __init__(self, nome):
        self.nome = nome

    def att_nome(self):
        return self.nome

Pessoa = namedtuple('Pessoa', 'nome email')

pessoa1 = Pessoa('Pessoa', 'fernando2rad@gmail.com')

print(t(pessoa1.nome, globals=globals()))

```

```
0.02824479899936705
```

Outra funcionalidade interessante associada com esta parte de gerenciamento de memória é justamente o recurso de verificar o tempo de execução de nossos códigos.

Inicialmente importamos da biblioteca timeit o módulo timeit, o referenciando como t. Exatamente da mesma forma da biblioteca collections importamos namedtuple, apenas para criarmos uma estrutura de dados que irá gerar um tempo considerável de execução.

Em seguida nos mesmos moldes do exemplo anterior criamos uma classe de nome Pessoa, com método construtor/inicializador e com um método de classe chamado att_nome( ) que serve para retornar um nome ao objeto homônimo.

De volta ao escopo global do código instanciamos nossa classe pessoa, onde fazendo o uso de namedtuple( ) criamos os campos ‘Pessoa’ e ‘nome email’ como referência para os dados que serão inseridos.

Também criamos uma variável de nome pessoa1 que esta sim, instancia a classe Pessoa( ), repassando para a mesma algumas informações.

Voltando ao foco de verificar o tempo de execução do código, por meio de nossa função print( ) chamamos a função t( ) parametrizando a mesma com pessoa1.nome, seguido de globals = globals( ), parâmetro que define o formato do valor retornado.

Agora que temos não só a estrutura do código, mas uma ação sendo realizada, é possível medir o tempo de execução de nosso código, sendo o retorno gerado pela função timeit( ).

Nesse caso, o valor 0.02... pode ser interpretado como 2000 milissegundos.

Threads

Quando aprendemos a programar, a maneira convencional como aprendemos gradualmente os conceitos nos faz subentender que nossos programas serão algoritmos aplicados, scripts contendo instruções a serem lidas, interpretadas e executadas sequencialmente, como em um fluxo de fila onde uma instrução é executada após a outra.

Claro que devemos sempre considerar estruturas de código que dependem umas das outras em conjunto e simultaneidade, porém muitas vezes haverão estruturas que podem serem processadas em paralelo.

Literalmente a décadas que os processadores de nossos computadores ganharam capacidade de processar fluxos de instruções em paralelo, sejam em conjunto ou independentes, de modo a quebrar grandes blocos de instrução em pequenos que em paralelo podem ser mais rapidamente resolvidos.

Partindo dessa premissa, também existem diversas estruturas de dados, a nível da camada de desenvolvimento mesmo e não de arquitetura de hardware, que permitem criar códigos otimizados para serem executados suportando paralelismo.

A nível de código, cada instrução é uma thread.

- Execução sequencial Vs paralela

```
1   def mensagem1():
2     contador = 0
3     while contador <= 3:
4       print('Estou trabalhando...')
5       contador += 1
6
7   def mensagem2():
8     contador2 = 0
9     while contador2 <= 3:
10      print('Estou estudando...')
11      contador2 += 1
12
13  mensagem1()
14  mensagem2()
15
```

```
Estou trabalhando...
Estou trabalhando...
Estou trabalhando...
Estou trabalhando...
Estou estudando...
Estou estudando...
Estou estudando...
Estou estudando...
```

Partindo para o código, em um exemplo básico apenas para entendimento da lógica dos pontos abordados anteriormente, temos duas funções mensagem1( ) e mensagem2( ) que como o nome sugere, exibem em tela uma mensagem.

Apenas para que nosso exemplo fique mais claro, é criada uma estrutura de repetição em ambas funções que faz com que a mensagem em si seja repetida um número específico de vezes.

Por fim chamamos as funções, acompanhando via terminal seu retorno.

Repare que pela leitura léxica usual, o interpretador logo na primeira linha leu a função mensagem1( ) e a executou, sendo assim gerado 4 mensagens “Estou trabalhando” como retorno. Em seguida ao chegar na linha 7 do código, é executada a função

mensagem2( ), idêntica a mensagem1( ), com exceção da mensagem exibida, que neste caso é “ Estou estudando”.

O ponto é, de acordo com o que nos é mostrado de retorno, primeiro foi executada toda a função mensagem1( ) e somente após a conclusão da mesma foi executada a função mensagem2( ), sequencialmente, uma após a outra.

```
1   import threading
2
3   def mensagem3():
4     contador = 0
5     while contador <= 3:
6       print('Estou trabalhando...')
7       contador += 1
8
9   def mensagem4():
10    contador2 = 0
11    while contador2 <= 3:
12      print('Estou estudando...')
13      contador2 += 1
14
15  threading.Thread(target=mensagem3).start()
16  mensagem4()
17
```

```
Estou trabalhando...
Estou estudando...
Estou trabalhando...
Estou estudando...
Estou trabalhando...
Estou estudando...
Estou trabalhando...
Estou estudando...
```

Partindo para a execução em paralelismo, inicialmente importamos a biblioteca threading.

Na sequência criamos duas funções iguais as do exemplo anterior, apenas de números diferentes (mensagem3( ) e mensagem4( )).

Repare na linha 15 do código, da biblioteca threading chamamos o método Thread( ), onde definimos como parâmetro para target o nome da função a qual queremos paralelizar, seguido do método start( ) que simplesmente aciona o gatilho para que esta thread seja criada.

Executando esse código agora obtemos como retorno “Estou trabalhando...” e “Estou estudando...” de forma intercalada, pois ambas funções estão sendo executadas ao mesmo tempo, logo, nos é fornecido seus retornos de forma paralela.

```
1   from threading import Thread
2
3   class MsgInterna(Thread):
4     def __init__(self, num):
5       Thread.__init__(self)
6       self.num = num
7
8     def run(self):
9       print('Você está logado no sistema.')
10
11  var1 = MsgInterna(1)
12  var1.start()
13
```

```
Você está logado no sistema.
```

Supondo que tenhamos uma estrutura orientada a objetos, as mesmas regras da programação estruturada são validas também nesse contexto no que diz respeito ao paralelismo.

No exemplo, da biblioteca threading importamos apenas a função Thread.

Em seguida criamos uma classe de nome MsgInterna( ) que recebe como atributo de classe Thread, dessa forma, podemos definir para o interpretador quais estruturas podem ser executadas em paralelo.

Repare que no corpo da classe é criado o método construtor __init__( ) que recebe um número para num, apenas como gatilho

para ativação de algum método de classe interno.

Na sequência instanciamos Thread.__init__( ), assim definindo que toda estrutura de dados dessa classe pode ser executada em paralelo.

Por fim é criado um método de classe chamado run( ) que simplesmente exibe em tela uma mensagem. O importante a se observar aqui é que este método de classe, indentado como método filho da classe MsgInterna( ), será executado em paralelo, assim como todos métodos subsequentes seriam caso existissem definidos no corpo dessa classe.

- Single-thread vs multi-thread

Entendida a lógica de um paralelismo vamos ver na prática sua justificativa de obter melhor performance. De modo geral, como dito no início desse capítulo, quebrar um problema computacional em problemas computacionais tende a resultar em melhor performance.

Dentro da mesma lógica funcional, quebrar um fluxo de processamento de dados grande em partes menores e as redistribuir internamente para núcleos de processamento separados também tende a resultar em melhor performance de execução.

```python
import time

contador = 100000000

def contagem_regressiva(num):
    while num > 0:
        num -= 1

inicio = time.time()
contagem_regressiva(contador)
fim = time.time()

print(f'Tempo de execução: {fim - inicio}')

```

```
Tempo de execução: 4.646906852722168
```

Apenas como exemplo, criada uma estrutura de código básica que realiza uma contagem regressiva de um determinado número, nesse caso, um quintilhão, até zero.

Em seguida criamos uma estrutura de mede um tempo antes da função ser inicializada, retornando também um tempo quando a função foi concluída, para medir em segundos o tempo de execução desse programa.

Note que no método single thread, a contagem regressiva de 100000000 a 0 foi concluída em 4.64 segundos.

```python
import time
from threading import Thread

contador = 100000000

def contagem_regressiva(num):
    while num > 0:
        num -= 1

tempo = Thread(target = contagem_regressiva,
               args = (contador//2,))

inicio = time.time()
tempo.start()
tempo.join()
fim = time.time()

print(f'Tempo de execução: {fim - inicio}')

```

```
Tempo de execução: 2.784353733062744
```

Implementando multi-thread podemos realizar as devidas comparações.

Para esse exemplo, novamente da biblioteca threading importamos Thread.

Em seguida é criada uma variável de nome tempo, que chama a função Thread( ) a parametrizando com contagem_regressiva como o parâmetro de targer, assim como para args repassamos contador//2, que resultará no processamento dessa estrutura em duas vias paralelas.

Também é criada a estrutura para inicializar a thread em tempo.start( ), retornando os dados processados em paralelo novamente juntos via tempo.join( ).

O resultado como pode ser observado, é o tempo de execução de nossa contagem regressiva de 100000000 a 0 em 2.33 segundos, uma performance muito superior se comparado ao método comum single thread.

- Multi-processing

Concluindo nossa linha de raciocínio, podemos realizar o mesmo feito do exemplo anterior pelo método chamado multi-processing, onde cada processo ganhará seu próprio interpretador Python e espaço dedicado em memória, simulando computação paralela.

Em outras palavras, podemos dividir um certo problema computacional em partes onde cada núcleo do processador com um bloco específico de memória ram irá processar cada parte.

```
1   from multiprocessing import Pool
2   import time
3
4   contador = 100000000
5
6   def contagem_regressiva(num):
7       while num > 0:
8           num -= 1
9
10  if __name__ == '__main__':
11      pool = Pool(processes = 4)
12      inicio = time.time()
13      tempo = pool.apply_async(contagem_regressiva, [contador//2])
14      pool.close()
15      pool.join()
16      fim = time.time()
17      print(f'Tempo de execução: {fim - inicio}')
18
```

```
Tempo de execução: 1.4101715087890625
```

Dessa vez, da biblioteca multiprocessing importamos Pool, assim como importamos a biblioteca time para que possamos mensurar o tempo de execução doc código como feito nos exemplos anteriores.

Criada a mesma estrutura de contador e contagem regressiva dos exemplos anteriores, é criada uma estrutura condicional que valida que as operações ocorram a partir do arquivo principal do código e não de estruturas modularizadas.

Na sequência é criada uma variável de nome pool que chama a função Pool( ), parametrizando a mesma com processes = 4, que define o número núcleos do processador independentes.

Em seguida, em tempo aplicamos o método pool.apply_async( ) parametrizando com nossa função contagem_regressiva( ) e [contador//2]. Dessa forma, nosso código será executado em 4 núcleos de processamento, cada núcleo executando 2 threads em paralelo.

O resultado, nesse caso, é á contagem regressiva de 100000000 a 0 em 2.41 segundos.

Em resumo, a mesma função calculada em single thread demorou 4.64 segundos, em multi-thread 2.33 segundos e em multi-processing 1.41 segundos.

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.

```
Administrador: Prompt de Comando
(c) 2019 Microsoft Corporation. Todos os direitos reservados.

C:\Windows\system32>pip install numpy
Requirement already satisfied: numpy in c:\program files\python38\
lib\site-packages (1.18.1)

C:\Windows\system32>_
```

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 ordenadamente dentro de um intervalo numérico. 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.

```
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
```

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.

```
[[3 1 1 4]
 [3 2 1 2]
 [3 1 3 0]]
(3, 4)
```

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.

```
Cada elemento possui 8 bytes
Toda a matriz possui 480 bytes
```

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.

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]
[0 1 2 3 4]
[ 3  4  5  6  7  8  9 10 11 12 13 14]
[4 5 6 7]
[ 0  2  4  6  8 10 12 14]
[ 3  4  5  6  7  8  9 10 11 12 13 14]
```

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.

```
[0 1 2 3 4 5 6 7]
[[0 1 2 3]
 [4 5 6 7]]
```

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 gerados 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( ).

```
[[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]
```

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( ).

```
[ 3  6  9 12 15 18 21 24 27 30]
```

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.

```
[ 2  4  6  8 10]
```

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_GcNWQIMltp47OFyqVhUD2R?usp=sharing

- dataset
- minisom-master
- restricted-boltzmann-machines-master
- Advertising Dataset.py
- Advertising.csv
- Autos Dataset.py
- autos.csv
- Bichos Dataset.py
- Bolsa Dataset.py
- Boltzmann Machine.py
- Boltzmann.py
- Breast Cancer Dataset Carregando Rede Pronta.py
- Breast Cancer Dataset.py
- Breast Cancer Dataset Salvando Modelo.py
- Breast Cancer Dataset Teste Uma Amostra.py
- Breast Cancer Dataset Tuning.py
- classificador_binario.json
- classificador_binario_pesos.h5
- credit-data.csv
- Digits Dataset.py
- entradas-breast.csv
- Fashion Dataset TensorFlow.py
- games.csv
- Iris Dataset Densa Validação Cruzada.py
- Iris Dataset Densa.py
- Iris Dataset Simples.py
- iris.csv
- MNIST Validação Cruzada.py
- MNIST.py
- Perceptron 1 Camada Otimizado.py
- Perceptron 1 Camada Tabela AND.py
- Perceptron Multicamada XOR.py
- PETR4.SA.csv
- petr4-teste.csv
- petr4-treinamento.csv
- rbm.py
- saidas-breast.csv
- VGDB.py
- Vinhos.py
- wines.csv

Ou por meio de meu repositório no GitHub:

https://github.com/fernandofeltrin/Redes-Neurais-Artificiais

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 $\quad Z = \text{entradas} \times \text{pesos}$

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:

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]
saida = stepFunction(s)
```

```
[8]
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.

```
[1]    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]
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])
```

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]
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]
def Soma(e,p):
    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]  1 def stepFunction(soma):
     2   if (soma >= 1):
     3     return 1
     4   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]  1 def calculoSaida(reg):
     2   s = reg.dot(pesos)
     3   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

```
import numpy as np

```

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.

```
entradas = np.array([[0,0],
                     [0,1],
                     [1,0],
                     [1,1]])
saidas = np.array([[0],[1],[1],[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.

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.

```
pesos0 = np.array([[-0.424, -0.740, -0.961],
                   [0.358, -0.577, -0.469]])
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.

```
ntreinos = 100
taxaAprendizado = 0.3
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.

```
def sigmoid(soma):
    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).

```
20 for i in range(ntreinos):
21     camadaEntrada = entradas
22     somaSinapse0 = np.dot(camadaEntrada, pesos0)
23     camadaOculta = sigmoid(somaSinapse0)
24
25     somaSinapse1 = np.dot(camadaOculta, pesos1)
26     camadaSaida = sigmoid(somaSinapse1)
27
28     erroCamadaSaida = saidas - camadaSaida
29     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.

```
35 derivadaSaida = erroCamadaSaida * derivadaSaida
36 derivadaSaida = sigmoideDerivada(camadaSaida)
```

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.

```
pesos1Transposta = pesos1.T
deltaSaidaXpesos = deltaSaida.dot(pesos1Transposta)
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.

```
Console IPython
Console 1/A
Margem de Erro: 0.49767112522643897
Margem de Erro: 0.4976484214145201
Margem de Erro: 0.4976254998572073
Margem de Erro: 0.4976023592551371
Margem de Erro: 0.4975789982549841
Margem de Erro: 0.49755541545037174
Margem de Erro: 0.4975316093827602
Margem de Erro: 0.49750757854230654
Margem de Erro: 0.49748332136870266
Margem de Erro: 0.4974588362519857
Margem de Erro: 0.4974341215333255
Margem de Erro: 0.4974091755057867
Margem de Erro: 0.4973839964150665
Margem de Erro: 0.49735858246020803
Margem de Erro: 0.4973329317942916
Margem de Erro: 0.4973070425251006

In [30]:
```

Executando 100 vezes a margem de erro cai para 0.497.

```
Console IPython
Console 1/A
Margem de Erro: 0.36869231728556695
Margem de Erro: 0.3685977889255375
Margem de Erro: 0.3685034121821234
Margem de Erro: 0.36840918664731676
Margem de Erro: 0.3683151119144694
Margem de Erro: 0.36822118757829
Margem de Erro: 0.3681274132348392
Margem de Erro: 0.3680337884815269
Margem de Erro: 0.3679403129171078
Margem de Erro: 0.36784698614167816
Margem de Erro: 0.3677538077566716
Margem de Erro: 0.36766077736485536
Margem de Erro: 0.3675678945703269
Margem de Erro: 0.36747515897850924
Margem de Erro: 0.36738257019614784
Margem de Erro: 0.3672901278313063

In [31]:
```

Executando 1000 vezes, a margem de erro cai para 0.367.

```
Console IPython
Console 1/A
Margem de Erro: 0.13312615898616278
Margem de Erro: 0.1331261170573894
Margem de Erro: 0.13312607512928198
Margem de Erro: 0.1331260332018406
Margem de Erro: 0.13312599127506525
Margem de Erro: 0.13312594934895583
Margem de Erro: 0.1331259074235124
Margem de Erro: 0.13312586549873495
Margem de Erro: 0.1331258235746234
Margem de Erro: 0.13312578165117778
Margem de Erro: 0.133125739728398
Margem de Erro: 0.13312569780628414
Margem de Erro: 0.1331256558848362
Margem de Erro: 0.13312561396405417
Margem de Erro: 0.1331255720439378
Margem de Erro: 0.1331255301244873

In [32]:
```

Executando 100000 vezes, a margem de erro cai para 0.133.

```
Console IPython
Console 1/A
Margem de Erro: 0.006978827431434885
Margem de Erro: 0.006978823127971268
Margem de Erro: 0.006978818824515499
Margem de Erro: 0.006978814521067677
Margem de Erro: 0.006978810217627711
Margem de Erro: 0.00697880591419575
Margem de Erro: 0.006978801610771656
Margem de Erro: 0.006978797307355465
Margem de Erro: 0.00697879300394726
Margem de Erro: 0.006978788700546881
Margem de Erro: 0.006978784397154563
Margem de Erro: 0.0069787800937701015
Margem de Erro: 0.00697877590393495
Margem de Erro: 0.006978771487024861
Margem de Erro: 0.006978767183664164
Margem de Erro: 0.006978762880311359

In [33]:
```

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:

```
1 import numpy as np
2
3 entradas = np.array([[0,0],
4                      [0,1],
5                      [1,0],
6                      [1,1]])
7 saidas = np.array([[0],[1],[1],[0]])
8
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]])
12
13 ntreinos = 1000000
14 taxaAprendizado = 0.3
15 momentum = 1
16
17 def sigmoid(soma):
18     return 1 / (1 + np.exp(-soma))
19
20 def sigmoideDerivada(sig):
21     return sig * (1-sig)
22 sigDerivada = sigmoid(0.5)
23 sigDerivada1 = sigmoideDerivada(sigDerivada)
```

```
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)
50
51     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

```python
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.

stopwords - List (204 elements)

Index	Type	Size	Value
0	str	1	de
1	str	1	a
2	str	1	o
3	str	1	que
4	str	1	e
5	str	1	é
6	str	1	do
7	str	1	da
8	str	1	em
9	str	1	um
10	str	1	para

Save and Close Close

Explorando a variável stopwords podemos entender que aqui são filtradas e guardadas 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.

frases_reduzidas - List (34 elements)

Index	Type	Size	Value
0	tuple	2	(['feliz'], 'emoção positiva')
1	tuple	2	(['pesso', 'feliz'], 'emoção positiva')
2	tuple	2	(['tud', 'bem', 'comig'], 'emoção positiva')
3	tuple	2	(['alegr', 'lem'], 'emoção positiva')
4	tuple	2	(['comig', 'tud', 'ok'], 'emoção positiva')
5	tuple	2	(['bom', 'ser', 'am'], 'emoção positiva')
6	tuple	2	(['empolg', 'começ'], 'emoção positiva')
7	tuple	2	(['elogi', 'trabalh'], 'emoção positiva')
8	tuple	2	(['venc', 'part'], 'emoção positiva')
9	tuple	2	(['ganh', 'pres'], 'emoção positiva')
10	tuple	2	(['receb', 'promoç', 'carg'], 'emoção positiva')

Save and Close Close

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.

Index	Type	Size	Value
0	str	1	feliz
1	str	1	pesso
2	str	1	feliz
3	str	1	tud
4	str	1	bem
5	str	1	comig
6	str	1	alegr
7	str	1	lem
8	str	1	comig

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 do 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 exemplos subsequentes, usaremos de outras bibliotecas que nos permitirão uma margem maior de customização dos meios e métodos de processamento de nossos dados.

```
Console 1/A

In [ ]: runfile('C:/Users/Fernando/Desktop/Data Science/Mineração de Emoções/
mineracao.py'

Digite como você está se sentindo: Eu me sinto bem
emoção positiva

In [ ]:
```

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:

```
import nltk
#from base import *
#nltk.download('popular')

stopwords = nltk.corpus.stopwords.words('portuguese')

def removestopwords(texto):
    frases = []
    for (palavras, emocao) in texto:
        removesw = [p for p in palavras.split() if p not in stopwords]
        frases.append((removesw, emocao))
    return frases

def reduzpalavras(texto):
    steemer = nltk.stem.RSLPStemmer() # faz o radical das palavras
    frases_redux = []
    for (palavras, emocao) in texto:
        reduzidas = [str(steemer.stem(p)) for p in palavras.split() if p no
        frases_redux.append((reduzidas, emocao))
    return frases_redux

frases_reduzidas = reduzpalavras(base)

```

```
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
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
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
54  classificador = nltk.NaiveBayesClassifier.train(baseprocessada)
55
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
```

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.

Iris Versicolor **Iris Setosa** **Iris Virginica**

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:

Iris Dataset Simples.py

```
1 from sklearn.datasets import load_iris
2
```

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.

base - Dicionário (6 elementos)

Chave	Tipo	Tamanho	Valor
DESCR	str	1	.. _iris_dataset:
data	float64	(150, 4)	[[5.1 3.5 1.4 0.2] [4.9 3. 1.4 0.2]
feature_names	list	4	['sepal length (cm)', 'sepal width (cm)', ' length (cm)', 'petal ...
filename	str	1	C:\Users\Fernando\Anaconda3\lib\site-packag \sklearn\datasets\data\ir ...
target	int32	(150,)	[0 0 0 ... 2 2 2]
target_names	str320	(3,)	ndarray object of numpy module

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.

```
print(base.data)
print(base.target)
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 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 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.

```
entradas = base.data
saidas = base.target
rotulos = base.target_names
```

Em seguida criamos três variáveis, 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.

```
from sklearn.neighbors import KNeighborsClassifier

```

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.

```
especie = knn.predict([[5.9,3,5.1,1.8]])[0]
print(especie)
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.

valores_k - Dicionário (24 elementos)

Chave	Tipo	Tamanho	Valor
1	float64	1	0.9473684210526315
2	float64	1	0.9473684210526315
3	float64	1	0.9473684210526315
4	float64	1	0.9473684210526315
5	float64	1	0.9736842105263158
6	float64	1	0.9736842105263158
7	float64	1	1.0
8	float64	1	1.0
9	float64	1	1.0
10	float64	1	1.0
11	float64	1	1.0

Salvar e Fechar Fechar

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.

```
from sklearn.linear_model import LogisticRegression

```

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:

```
1 from sklearn.datasets import load_iris
2 from sklearn.neighbors import KNeighborsClassifier
3 from sklearn.model_selection import train_test_split
4 from sklearn import metrics
5 import matplotlib.pyplot as plt
6 from sklearn.linear_model import LogisticRegression
7
8 base = load_iris()
9 entradas = base.data
10 saidas = base.target
11 rotulos = base.target_names
12
13 knn = KNeighborsClassifier(n_neighbors = 1)
14 knn.fit(entradas, saidas)
15 knn.predict([[5.1,3.1,1.4,0.2]])
16
17 especie = knn.predict([[5.9,3,5.1,1.8]])[0]
18 print(especie)
19 rotulos[especie]
20
21 etreino, eteste, streino, steste = train_test_split(entradas,
22                                                    saidas,
23                                                    test_size = 0.25)
24 knn.fit(etreino, streino)
25 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.

Explorador de variáveis

Nome	Tipo	Tamanho	Valor
base	DataFrame	(150, 5)	Column names: sepal length, sepal width, petal length petal width, cl ...

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.

base - DataFrame

Index	sepal length	sepal width	petal length	petal width	class
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5	3.6	1.4	0.2	Iris-setosa
5	5.4	3.9	1.7	0.4	Iris-setosa
6	4.6	3.4	1.4	0.3	Iris-setosa

Formato | edimensiona | Cor de fu | Min/max c | alvar e Fech | Fechar

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.

```
5 entradas = base.iloc[:, 0:4].values
6 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.

```
8 from sklearn.preprocessing import LabelEncoder
9
10 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	Neurônio	Neurônio

	1	2	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.

```
classificador = Sequential()
classificador.add(Dense(units = 4,
                        activation = 'relu',
                        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.

```
classificador.add(Dense(units = 4,
                        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.

```
classificador.add(Dense(units = 3,
                        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.

```
classificador.compile(optimizer = 'adam',
                      loss = 'categorical_crossentropy',
                      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.

```
classificador.fit(etreino,
                  streino,
                  batch_size = 10,
                  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.

```
avalPerformance = classificador.evaluate(eteste, steste)

previsoes = classificador.predict(eteste)
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.

avalPerformance - Lista (2 elementos)

Índice	Tipo	Tamanho	Valor
0	float64	1	0.03765642162608473
1	float64	1	0.9736842105263158

Salvar e Fechar Fechar

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.

	0	1	2
0	3.61044e-05	0.999927	3.6697e-05
1	5.57247e-18	7.9111e-05	0.999921
2	2.45434e-06	0.999437	0.00056071
3	5.47915e-08	0.992885	0.00711488
4	2.62146e-08	0.93504	0.0649605
5	0.000404379	0.999546	4.93405e-05

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.

	0	1	2
0	False	True	False
1	False	False	True
2	False	True	False
3	False	True	False
4	False	True	False
5	False	True	False

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.

```
resultadosValCruzada = cross_val_score(estimator = classificadorValCruzada,
                                       X = entradas,
                                       y = saidas,
                                       cv = 10,
                                       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:

```python
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.

entradas - DataFrame

Index	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mea	ompactness_mea
0	17.99	10.38	122.8	1001	0.1184	0.2776
1	20.57	17.77	132.9	1326	0.08474	0.07864
2	19.69	21.25	130	1203	0.1096	0.1599
3	11.42	20.38	77.58	386.1	0.1425	0.2839
4	20.29	14.34	135.1	1297	0.1003	0.1328
5	12.45	15.7	82.57	477.1	0.1278	0.17
6	18.25	19.98	119.6	1040	0.09463	109
7	13.71	20.83	90.2	577.9	0.1189	0.1645
8	13	21.82	87.5	519.8	0.1273	0.1932
9	12.46	24.04	83.97	475.9	0.1186	0.2396
10	16.02	23.24	102.7	797.8	0.08206	0.06669
11	15.78	17.89	103.6	781	0.0971	0.1292
12	19.17	24.8	132.4	1123	0.0974	0.2458
13	15.85	23.95	103.7	782.7	0.08401	0.1002

Formato | Redimensionar | Cor de fundo | Min/max de coluna | Salvar e Fechar | Fechar

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.

```
from sklearn.model_selection import train_test_split

etreino, eteste, streino, steste = train_test_split(entradas,
                                                    saidas,
                                                    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.

```
classificador = Sequential()
classificador.add(Dense(units = 16,
                        activation = 'relu',
                        kernel_initializer = 'random_uniform',
                        input_dim = 30))
classificador.add(Dense(units = 1,
                        activation = 'sigmoid'))
classificador.compile(optimizer = 'adam',
                      loss = 'binary_crossentropy',
                      metrics = ['binary_accuracy'])
classificador.fit(etreino,
                  streino,
                  batch_size = 10,
                  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.

```
previsor = classificador.predict(eteste)

```

Criamos nosso previsor, que realiza as devidas previsões sobre eteste por meio da função .predict( ).

```
from sklearn.metrics import confusion_matrix, accuracy_score

margem_acertos = accuracy_score(steste, previsor)
matriz_confusao = confusion_matrix(steste, previsor)
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.

```
classificador = KerasClassifier(build_fn = valCruzada,
                                epochs = 100,
                                batch_size = 10)
resultadoValCruzada = cross_val_score(estimator = classificador,
                                      X = entradas,
                                      y = saidas,
                                      cv = 10,
                                      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.

```
mediaValCruzada = resultadoValCruzada.mean()
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, neurc
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( ).

```
parametros = {'batch_size':[10,30],
              'epochs':[50,100],
              'optimizer':['adam','sgd'],
              'loss':['binary_crossentropy','hinge'],
              'kernel_initializer':['random_uniform','normal'],
              'activation':['relu','tanh'],
              '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.

```
tunagem = GridSearchCV(estimator = classificadorTunado,
                       param_grid = parametros,
                       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.

```
tunagem = tunagem.fit(entradas,saidas)
```

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.

```
melhores_parametros = tunagem.best_params_
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.

melhores_parametros - Dicionário (7 elementos)

Chave	Tipo	Tamanho	Valor
activation	str	1	relu
batch_size	int	1	10
epochs	int	1	100
kernel_initializer	str	1	random_uniform
loss	str	1	binary_crossentropy
neurons	int	1	8
optimizer	str	1	adam

Salvar e Fechar Fechar

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:

```
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
7
8 entradas = pd.read_csv('entradas-breast.csv')
9 saidas = pd.read_csv('saidas-breast.csv')
10
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
```

```
classificadorTunado = KerasClassifier(build_fn = tuningClassificador)
parametros = {'batch_size':[10,30],
              'epochs':[50,100],
              'optimizer':['adam','sgd'],
              'loss':['binary_crossentropy','hinge'],
              'kernel_initializer':['random_uniform','normal'],
              'activation':['relu','tanh'],
              'neurons':[10,8]}
tunagem = GridSearchCV(estimator = classificadorTunado,
                       param_grid = parametros,
                       scoring = 'accuracy')
tunagem = tunagem.fit(entradas,saidas)
melhores_parametros = tunagem.best_params_
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

```
import pandas as pd
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, Dropout
```

A partir de um novo arquivo iniciamos como sempre com as importações necessárias.

```
entradas = pd.read_csv('entradas-breast.csv')
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).

```
1 import pandas as pd
2 import numpy as np
3 from keras.models import Sequential
4 from keras.layers import Dense, Dropout
5
6 entradas = pd.read_csv('entradas-breast.csv')
7 saidas = pd.read_csv('saidas-breast.csv')
8
9 classificadorB = Sequential()
10 classificadorB.add(Dense(units = 8,
11                         activation = 'relu',
12                         kernel_initializer = 'normal',
13                         input_dim = 30))
14 classificadorB.add(Dropout(0.2))
15 classificadorB.add(Dense(units = 8,
16                         activation = 'relu',
17                         kernel_initializer = 'normal',))
18 classificadorB.add(Dropout(0.2))
19 classificadorB.add(Dense(units = 1,
20                         activation = 'sigmoid'))
21 classificadorB.compile(optimizer = 'adam',
22                       loss = 'binary_crossentropy',
23                       metrics = ['binary_accuracy'])
24 classificadorB.fit(entradas,
25                    saidas,
26                    batch_size = 10,
27                    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.

```
29 classificador_json = classificadorB.to_json()
30
31 with open('classificador_binario.json', 'w') as json_file:
32     json_file.write(classificador_json)
33
34 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.

Nome	Tipo	Tamanho
saidas-breast.csv	Microsoft Excel ...	2 KB
Breast Cancer Dataset.py	Arquivo PY	5 KB
classificador_binario.json	Arquivo JSON	2 KB
classificador_binario_pesos.h5	Arquivo H5	18 KB
entradas-breast.csv	Microsoft Excel ...	115 KB

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.

Editor de texto - estrutura_rede


```
{"class_name": "Sequential", "config": {"name": "sequential_2", "layers":
[{"class_name": "Dense", "config": {"name": "dense_4", "trainable": true,
"batch_input_shape": [null, 30], "dtype": "float32", "units": 8,
"activation": "relu", "use_bias": true, "kernel_initializer":
{"class_name": "RandomNormal", "config": {"mean": 0.0, "stddev": 0.05,
"seed": null}}, "bias_initializer": {"class_name": "Zeros", "config": {}},
"kernel_regularizer": null, "bias_regularizer": null,
"activity_regularizer": null, "kernel_constraint": null, "bias_constraint":
null}}, {"class_name": "Dropout", "config": {"name": "dropout_3",
"trainable": true, "rate": 0.2, "noise_shape": null, "seed": null}},
{"class_name": "Dense", "config": {"name": "dense_5", "trainable": true,
"units": 8, "activation": "relu", "use_bias": true, "kernel_initializer":
{"class_name": "RandomNormal", "config": {"mean": 0.0, "stddev": 0.05,
"seed": null}}, "bias_initializer": {"class_name": "Zeros", "config": {}},
"kernel_regularizer": null, "bias_regularizer": null,
"activity_regularizer": null, "kernel_constraint": null, "bias_constraint":
null}}, {"class_name": "Dropout", "config": {"name": "dropout_4",
"trainable": true, "rate": 0.2, "noise_shape": null, "seed": null}},
{"class_name": "Dense", "config": {"name": "dense_6", "trainable": true,
"units": 1, "activation": "sigmoid", "use_bias": true,
"kernel_initializer": {"class_name": "VarianceScaling", "config": {"scale":
1.0, "mode": "fan_avg", "distribution": "uniform", "seed": null}},
"bias_initializer": {"class_name": "Zeros", "config": {}},
"kernel_regularizer": null, "bias_regularizer": null,
"activity_regularizer": null, "kernel_constraint": null, "bias_constraint":
null}}]}, "keras_version": "2.2.4", "backend": "tensorflow"}
```


Salvar e Fechar Fechar

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.

```
dateCrawled,name,seller,offerType,price,abtest,vehicleType,yearOfRegistration,gearbox,powerPS,model,kilometer,monthOfRegistration,fuelType,brand,notRepairedDamage,dateCreated,nrOfPictures,post
2016-03-24 11:52:17,Golf_3_1.6,privat,Angebot,480,test,,1993,manuell,0,golf,150000,0,benzin,volkswagen,,2016-03-24 00:00:00,0,70435,2016-04-07 03:16:57
2016-03-24 10:58:45,A5_Sportback_2.7_Tdi,privat,Angebot,18300,test,coupe,2011,manuell,190,,125000,5,diesel,audi,ja,2016-03-24 00:00:00,0,66954,2016-04-07 01:46:50
2016-03-14 12:52:21,Jeep_Grand_Cherokee_"Overland",privat,Angebot,9800,test,suv,2004,automatik,163,grand,125000,8,diesel,jeep,,2016-03-14 00:00:00,0,90480,2016-04-05 12:47:46
2016-03-17 16:54:04,GOLF_4_1_4__3TÜRER,privat,Angebot,1500,test,kleinwagen,2001,manuell,75,golf,150000,6,benzin,volkswagen,nein,2016-03-17 00:00:00,0,91074,2016-03-17 17:40:17
2016-03-31 17:25:20,Skoda_Fabia_1.4_TDI_PD_Classic,privat,Angebot,3600,test,kleinwagen,2008,manuell,69,fabia,90000,7,diesel,skoda,nein,2016-03-31 00:00:00,0,60437,2016-04-06 10:17:21
2016-04-04 17:36:23,BMW_316i___e36_Limousine___Bastlerfahrzeug__Export,privat,Angebot,650,test,limousine,1995,manuell,102,3er,150000,10,benzin,bmw,ja,2016-04-04 00:00:00,0,33775,2016-04-06 19:17:0
2016-04-01 20:48:51,Peugeot_206_CC_110_Platinum,privat,Angebot,2200,test,cabrio,2004,manuell,109,2_reihe,150000,8,benzin,peugeot,nein,2016-04-01 00:00:00,0,67112,2016-04-05 18:18:39
2016-03-21 18:54:38,VW_Derby_Bj_80__Scheunenfund,privat,Angebot,0,test,limousine,1980,manuell,50,andere,40000,7,benzin,volkswagen,nein,2016-03-21 00:00:00,0,19348,2016-03-25 16:47:58
2016-04-04 23:42:13,Ford_C___Max_Titanium_1_0_L_EcoBoost,privat,Angebot,14500,control,bus,2014,manuell,125,c_max,30000,8,benzin,ford,,2016-04-04 00:00:00,0,94505,2016-04-04 23:42:13
2016-03-17 10:53:50,VW_Golf_4_5_tuerig_zu_verkaufen_mit_Anhaengerkupplung,privat,Angebot,999,test,kleinwagen,1998,manuell,101,golf,150000,0,,volkswagen,,2016-03-17 00:00:00,0,27472,2016-03-31 17:1
2016-03-26 19:54:18,Mazda_3_1.6_Sport,privat,Angebot,2000,control,limousine,2004,manuell,105,3_reihe,150000,12,benzin,mazda,nein,2016-03-26 00:00:00,0,96224,2016-04-06 10:45:34
2016-04-07 10:06:22,Volkswagen_Passat_Variant_2.0_TDI_Comfortline,privat,Angebot,2799,control,kombi,2005,manuell,140,passat,150000,12,diesel,volkswagen,ja,2016-04-07 00:00:00,0,57290,2016-04-07 10:2
2016-03-15 22:49:09,VW_Passat_Facelift_35i__"7Sitzer",privat,Angebot,999,control,kombi,1995,manuell,115,passat,150000,11,benzin,volkswagen,,2016-03-15 00:00:00,0,37269,2016-04-01 13:16:16
2016-03-21 21:37:40,VW_PASSAT_1.9_TDI_131_PS_LEDER,privat,Angebot,2500,control,kombi,2004,manuell,131,passat,150000,2,,volkswagen,nein,2016-03-21 00:00:00,0,90762,2016-03-23 02:50:54
2016-03-21 12:57:01,Nissan_Navara_2.5DPF_SE4x4_Klima_Sitzheizg_Bluetooth.Doppelkabine,privat,Angebot,17999,control,suv,2011,manuell,190,navara,70000,3,diesel,nissan,nein,2016-03-21 00:00:00,0,04177,
2016-03-11 21:39:15,KA_Lufthansa_Edition_450€_VB,privat,Angebot,450,test,kleinwagen,1910,,0,ka,5000,0,benzin,ford,,2016-03-11 00:00:00,0,24148,2016-03-19 08:46:47
2016-04-01 12:46:46,Polo_6n_1_4,privat,Angebot,300,test,,2016,,60,polo,150000,0,benzin,volkswagen,,2016-04-01 00:00:00,0,38871,2016-04-01 12:46:46
2016-03-20 10:25:19,Renault_Twingo_1.2_16V_Aut.,privat,Angebot,1750,control,kleinwagen,2004,automatik,75,twingo,150000,2,benzin,renault,nein,2016-03-20 00:00:00,0,65599,2016-04-06 13:16:07
2016-03-23 15:48:05,Ford_C_MAX_2.0_TDCI_DPF_Titanium,privat,Angebot,7550,test,bus,2007,manuell,136,c_max,150000,6,diesel,ford,nein,2016-03-23 00:00:00,0,88361,2016-04-05 18:45:11
2016-04-01 22:55:47,Mercedes_Benz_A_160_Classic_Klima,privat,Angebot,1850,test,bus,2004,manuell,102,a_klasse,150000,1,benzin,mercedes_benz,nein,2016-04-01 00:00:00,0,49565,2016-04-05 22:46:05
2016-04-01 19:56:48,Volkswagen_Scirocco_1.4_TSI_Sport,privat,Angebot,10400,control,coupe,2009,manuell,160,scirocco,100000,4,benzin,volkswagen,nein,2016-04-01 00:00:00,0,75365,2016-04-05 16:45:49
2016-03-27 11:38:00,BMW_530i_TÜV_7/17_Scheckheftgepflegt_sehr_guter_Zustand,privat,Angebot,3699,test,limousine,2002,automatik,231,5er,150000,7,benzin,bmw,nein,2016-03-27 00:00:00,0,68309,2016-04-
```

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.

base - DataFrame

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

Formato | Redimensionar | Cor de fundo | Min/max de coluna | Salvar e Fechar | Fechar

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.

```
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)
```

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.

```
base['name'].value_counts()

```

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.

```
base = base.drop('name', axis=1)
base = base.drop('seller', axis=1)
base = base.drop('offerType', axis=1)
```

Nesta etapa, eliminamos três colunas em função de suas inconsistências em seus dados.

```
varTeste1 = base.loc[base.price <= 10]

```

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.

```
base = base[base.price > 10]

```

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.

```
varTeste2 = base.loc[base.price > 350000]
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.

```
base.loc[pd.isnull(base['vehicleType'])]
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.

```
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()
```

Da mesma forma repetimos o processo para todas colunas onde houverem dados faltando (NaN), descobrindo os dados/valores médios para preencher tais espaços vazios.

```
valores_medios = {'vehicleType': 'limousine',
                  'gearbox': 'manuell',
                  'model': 'golf',
                  'fuelType': 'benzin',
                  '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.

	0	1	2	3
0	0	1	0	0
1	0	1	0	0
2	0	1	0	0
3	0	1	0	0
4	0	1	0	0
5	0	1	0	0
6	0	1	0	0
7	1	0	0	1
8	0	1	0	0
9	1	0	0	0
10	1	0	0	0
11	1	0	0	0

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.

```
from keras.models import Sequential
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.

```
regressor.fit(entradas,
              saidas,
              batch_size = 300,
              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.

```
previsoes = regressor.predict(entradas)
saidas.mean()
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.

```
media = resValCruzada.mean()
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:

```python
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]
```

```python
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])
```

```
55 onehotencoder = OneHotEncoder(categorical_features = [0,1,3,5,8,9,10])
56 entradas = onehotencoder.fit_transform(entradas).toarray()
57
58 regressor = Sequential()
59 regressor.add(Dense(units = 158,
60                     activation = 'relu',
61                     input_dim = 316))
62 regressor.add(Dense(units = 158,
63                     activation = 'relu'))
64 regressor.add(Dense(units = 1,
65                     activation = 'linear'))
66 regressor.compile(loss = 'mean_absolute_error',
67                  optimizer = 'adam',
68                  metrics = ['mean_absolute_error'])
69 regressor.fit(entradas,
70               saidas,
71               batch_size = 300,
72               epochs = 100)
73
74 previsoes = regressor.predict(entradas)
75 saidas.mean()
76 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 base = pd.read_csv('games.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.

base - DataFrame

Index	Name	latforr	Year_of_Release	Genre	Publisher
0	Wii Sports	Wii	2006	Sports	Nintendo
1	Super Mario Bros.	NES	1985	Platform	Nintendo
2	Mario Kart Wii	Wii	2008	Racing	Nintendo
3	Wii Sports Resort	Wii	2009	Sports	Nintendo
4	Pokemon Red/ Pokemon Blue	GB	1996	Role-Playing	Nintendo
5	Tetris	GB	1989	Puzzle	Nintendo
6	New Super Mario Bros.	DS	2006	Platform	Nintendo
7	Wii Play	Wii	2006	Misc	Nintendo
8	New Super Mario Bros. …	Wii	2009	Platform	Nintendo
9	Duck Hunt	NES	1984	Shooter	Nintendo
10	Nintendogs	DS	2005	Simulation	Nintendo
11	Mario Kart DS	DS	2005	Racing	Nintendo
12	Pokemon Gold/ Pokemon Silv…	GB	1999	Role-Playing	Nintendo
13	Wii Fit	Wii	2007	Sports	Nintendo

Formato | Redimensionar | Cor de func | Min/max de | alvar e Fecha | Fechar

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.

```
7 base = base.drop('Other_Sales', axis = 1)
8 base = base.drop('Global_Sales', axis = 1)
9 base = base.drop('Developer', axis = 1)
10 base = base.dropna(axis = 0)
11 base = base.loc[base['NA_Sales'] > 1]
12 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.

```
14 base['Name'].value_counts()
15 backupName = base.Name
16 base = base.drop('Name', axis = 1)
```

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.

```
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
```

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.

```
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
```

Como sempre, realizamos as devidas importações das ferramentas que inicialmente não são pré-carregadas por nossa

IDE.

```
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])
```

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.

```
onehotencoder = OneHotEncoder(categorical_features = [0,2,3,8])
entradas = onehotencoder.fit_transform(entradas).toarray()
```

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( ).

```
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)
```

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
previsao_NA, previsao_EU, previsao_JP = regressor.predict(entradas)
```

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.

backupName - Series

Index	Name
0	Wii Sports
2	Mario Kart Wii
3	Wii Sports Resort
6	New Super Mario Bros.
7	Wii Play
8	New Super Mario Bros. …
11	Mario Kart DS
13	Wii Fit
14	Kinect Adventures!
15	Wii Fit Plus
16	Grand Theft Auto V
19	Brain Age: Train Your B…
23	Grand Theft Auto V
24	Grand Theft Auto: Vice C…

Formato | Redimensior

previsao_NA - Matriz N

	0
0	41.1149
1	21.3701
2	15.6081
3	16.6921
4	10.7007
5	13.0384
6	15.5568
7	12.1215
8	8.7159
9	11.2487
10	4.89999
11	6.74438
12	4.9

Formato | Redimensi

vendas_NA - Matriz NumPy

	0
0	41.36
1	15.68
2	15.61
3	11.28
4	13.96
5	14.44
6	9.71
7	8.92
8	15
9	9.01
10	7.02
11	4.74
12	9.66

Formato | Redimensionar | Cor de fun

Salva

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:

```python
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()
```

```python
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.

An Introduction to Statistical Learning

with Applications in R

Gareth James, Daniela Witten, Trevor Hastie and Robert Tibshirani

Home

About this Book

R Code for Labs

Data Sets and Figures

ISLR Package

Get the Book

Author Bios

Errata

Figures	Data Sets
All Figures In A Single Zip File	Advertising.csv

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.

sales
TV
0
5
10
15
20
25
0
50
100
150
200
250
300

radio
0
10
20
30
40
50

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.

```
from sklearn.model_selection import train_test_split

etreino, eteste, streino, steste = train_test_split(entradas,
                                                    saidas,
                                                    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.

```
27
28 print(list(zip(['TV', 'radio', 'newspaper'], reglinear.coef_)))
```

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.

```
print(reglinear.predict([[230.1,37.8,69.2]]))
```

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.

```
previsor = reglinear.predict(eteste)
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.

```
from sklearn import metrics
```

Podemos ainda realizar alguns testes para confirmar a integridade de nossos dados. Aqui usaremos da ferramenta metrics da biblioteca sklearn para tal fim.

```
mae = metrics.mean_absolute_error(steste,previsor)
```

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:

```python
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.

Digits Dataset.py

```
1 from sklearn import datasets
2
```

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.

```
from sklearn import svm
classificador = svm.SVC()
classificador.fit(etreino,streino)
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.

```
from sklearn import metrics
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.

```
import numpy as np
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.

```
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
```

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.

```
regressor = LogisticRegression()
regressor.fit(entradas,saidas)
previsor_regl = regressor.predict([rgb2gray(imagem)])
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:

```
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).

```
9 (etreino,streino),(eteste,steste) = mnist.load_data()
10
```

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).

```
11 plt.imshow(etreino[0], cmap = 'gray')
12 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.

```
classificador.compile(loss = 'categorical_crossentropy',
                      optimizer = 'adam',
                      metrics = ['accuracy'])
classificador.fit(etreino,
                  streino,
                  batch_size = 128,
                  epochs = 10,
                  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.

```
1 from keras.datasets import mnist
2 from keras.models import Sequential
3 from keras.layers import Dense, Flatten, Conv2D, MaxPooling2D
4 from keras.utils import np_utils
5 import numpy as np
6 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.

```
8 seed = 5
9 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.

```
11 (etreino,streino), (eteste,steste) = mnist.load_data()
12 entradas = etreino.reshape(etreino.shape[0], 28, 28, 1)
13 entradas = entradas.astype('float32')
14 entradas /= 255
15 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.

```
17 kfold = StratifiedKFold(n_splits = 5, shuffle = True, random_state = seed)
18 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.

```
20 a = np.zeros(5)
21 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.

```
23 for evalcruzada,svalcruzada in kfold.split(entradas,
24                                            np.zeros(shape=(saidas.shape[0],1)
25     classificador = Sequential()
26     classificador.add(Conv2D(32, (3,3), input_shape=(28,28,1), activation='rel
27     classificador.add(MaxPooling2D(pool_size = (2,2)))
28     classificador.add(Flatten())
29     classificador.add(Dense(units = 128, activation = 'relu'))
30     classificador.add(Dense(units = 10, activation = 'softmax'))
31     classificador.compile(loss = 'categorical_crossentropy', optimizer='adam',
32                           metrics = ['accuracy'])
33     classificador.fit(entradas[evalcruzada], saidas[evalcruzada],
34                       batch_size = 128, epochs = 5)
35     precisao = classificador.evaluate(entradas[svalcruzada], saidas[svalcruzac
36     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):

```
1 from keras.datasets import mnist
2 from keras.models import Sequential
3 from keras.layers import Dense, Flatten, Conv2D, MaxPooling2D
4 from keras.utils import np_utils
5 import numpy as np
6 from sklearn.model_selection import StratifiedKFold
7
8 seed = 5
9 np.random.seed(seed)
10
11 (etreino,streino), (eteste,steste) = mnist.load_data()
12 entradas = etreino.reshape(etreino.shape[0], 28, 28, 1)
13 entradas = entradas.astype('float32')
14 entradas /= 255
15 saidas = np_utils.to_categorical(streino, 10)
16
17 kfold = StratifiedKFold(n_splits = 5, shuffle = True, random_state = seed)
18 resultados = []
19
20 a = np.zeros(5)
21 b = np.zeros(shape = (saidas.shape[0], 1))
22
23 for evalcruzada,svalcruzada in kfold.split(entradas,
24                                            np.zeros(shape=(saidas.shape[0],1)
```

```
25     classificador = Sequential()
26     classificador.add(Conv2D(32, (3,3), input_shape=(28,28,1), activation='rel
27     classificador.add(MaxPooling2D(pool_size = (2,2)))
28     classificador.add(Flatten())
29     classificador.add(Dense(units = 128, activation = 'relu'))
30     classificador.add(Dense(units = 10, activation = 'softmax'))
31     classificador.compile(loss = 'categorical_crossentropy', optimizer='adam',
32                           metrics = ['accuracy'])
33     classificador.fit(entradas[evalcruzada], saidas[evalcruzada],
34                       batch_size = 128, epochs = 5)
35     precisao = classificador.evaluate(entradas[svalcruzada], saidas[svalcruzac
36     resultados.append(precisao[1])
37
38 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.

Nome	Data de modificaç...	Tipo	Tamanho
dataset	19/08/2019 16:05	Pasta de arquivos	
Bichos Dataset.py	19/08/2019 15:24	Arquivo PY	3 KB

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.

```
6 gerador_treino = ImageDataGenerator(rescale = 1.0/255,
7                                     rotation_range = 7,
8                                     horizontal_flip = True,
9                                     shear_range = 0.2,
10                                    height_shift_range = 0.07,
11                                    zoom_range = 0.2)
12 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.

```
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)
```

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:

```python
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')
```

```python
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.

```
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()
```

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.

```
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)
])
```

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.

```
classificador.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
classificador.fit(etreino,
                  streino,
                  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:

```
1 from __future__ import absolute_import, division, print_function, unicode_lite
2 import tensorflow as tf
3 from tensorflow import keras
4 import numpy as np
5 import matplotlib.pyplot as plt
6
7 base = keras.datasets.fashion_mnist
8
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
16
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()
```

```
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 ])
33
34 classificador.compile(optimizer='adam',
35               loss='sparse_categorical_crossentropy',
36               metrics=['accuracy'])
37 classificador.fit(etreino,
38                   streino,
39                   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

```
import numpy as np
import pandas as pd
from keras.models import Sequential
from keras.layers import Dense, Dropout
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.

```
15 previsores = []
16 preco_real = []
17
18 for i in range(90,1242):
19     previsores.append(base_treino_normalizada[i-90:i,0])
20     preco_real.append(base_treino_normalizada[i,0])
21
22 previsores, preco_real = np.array(previsores), np.array(preco_real)
23 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 previsto 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.

```
previsores_teste.mean()
preco_real_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:

```python
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
```

```python
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()
```

Classificação por Recorrência de Dados – Imdb Dataset

Seguindo com nossos exemplos, vamos buscar entender na prática uma rede neural recorrente, tipo este que via TensorFlow se mostrou um eficiente modelo de classificação quando temos que minerar informações a partir de textos usando dos mesmos como base.

review	sentiment
49582 unique values	2 unique values
One of the other reviewers has mentioned that after watching just 1 Oz episode you'll be hooked. The...	positive
A wonderful little production. The filming technique is very unassuming- very old-time-B...	positive
I thought this was a wonderful way to spend time on a too hot summer weekend, sitting in the air con...	positive

Fazendo uso de uma das bases de dados fornecidas como exemplo para a biblioteca Keras, dessa vez usaremos da imdb dataset, que por sua vez é uma base de dados gerada a partir do site Imdb onde cerca de 50000 reviews de filmes gerados pelos usuários do site passam a ser usados como parâmetro para avaliar um filme como bom ou ruim.

A base de dados em si que usaremos, integrada aos datasets de exemplo da biblioteca keras, já vem por padrão com todas palavras de cada review convertidas para um formato numérico e internamente indexado, dessa forma nosso foco será apenas identificar os padrões que podem ser associados a uma determinada emoção.

Sempre que estamos minerando dados a partir de textos o intuito é treinar uma rede neural artificial para que aprenda as associações de palavras que descrevem um contexto, nesse caso, quais padrões de palavras recorrentemente aparecem quando um filme é avaliado positivamente assim como quando ocorre o oposto, e nesse processo a rede neural dará menos peso a palavras que se repetem em ambos contextos ou que são ferramentas de linguagem, e usará as palavras características para bom ou para ruim como critério de aprendizado.

```
1  import tensorflow as tf
2  from tensorflow.keras.datasets import imdb
3
```

Como de costume, inicialmente importamos as bibliotecas, módulos e pacotes que usaremos em nosso código.

Para esse exemplo em particular importamos apenas a biblioteca tensorflow e do módulo datasets da biblioteca keras importamos imdb.

```
1  (X_treino, y_treino), (X_teste, y_teste) = imdb.load_data(num_words=20000)
2
```

Em seguida criamos nossas variáveis X_treino, y_treino, X_teste, y_teste que por sua vez guardarão os dados oriundos da base de dados imdb. Novamente, o processo de desempacotar tais

dados e os dividir para as respectivas variáveis é feito por meio da função load_data( ), porém, nesse caso não faremos o uso de toda a base de dados, mas de uma parte da mesma, bastando assim especificar o número de palavras a ser importada como valor do parâmetro num_words.

```
1  print(X_treino.shape)
2

(25000,)
```

Uma vez que tenhamos os dados distribuídos para suas respectivas variáveis podemos fazer algumas verificações de praxe para explorar e entender algumas características dos mesmos.

Por meio de nossa função print( ) parametrizada com X_treino.shape podemos ver o tamanho de nossa base de dados. Uma pequena disparidade entre os dados definidos para importação e a quantidade apresentada pode ser percebida, isso se dá por uma questão de compatibilidade, pois a quantidade de dados importados deverá ser compatível com os possíveis formatos de cruzamento de dados internamente em suas matrizes.

```
1  print(X_treino)
2

[list([1, 14, 22, 16, 43, 530, 973, 1622, 1385, 65, 458, 4468, 66, 3941,
 list([1, 194, 1153, 194, 8255, 78, 228, 5, 6, 1463, 4369, 5012, 134, 26,
 list([1, 14, 47, 8, 30, 31, 7, 4, 249, 108, 7, 4, 5974, 54, 61, 369, 13,
 ...
 list([1, 11, 6, 230, 245, 6401, 9, 6, 1225, 446, 2, 45, 2174, 84, 8322,
 list([1, 1446, 7079, 69, 72, 3305, 13, 610, 930, 8, 12, 582, 23, 5, 16,
 list([1, 17, 6, 194, 337, 7, 4, 204, 22, 45, 254, 8, 106, 14, 123, 4, 12
```

Parametrizando nossa função print( ) apenas com X_treino nos é retornado uma matriz onde cada elemento é uma lista de características, onde cada valor corresponde a uma palavra mapeada para a base de dados.

Apenas como exemplo, raciocine que a palavra “carro” tenha sido indexada como 87, desse modo, em todas as listas onde houvesse referência ao número 87 ali estaria o valor matemático

equivalente a palavra carro. Toda base de dados está nesse formato, haja visto que o processamento dos dados se aplica sobre dados em formato numérico.

```
1  print(X_treino[44])
2

[1, 3420, 9, 34, 230, 61, 514, 7, 32, 7, 5712, 495, 2, 9, 31, 2304, 7, 35
```

Isso fica mais claro ao selecionarmos apenas um elemento da base de dados. Inicialmente este elemento era uma string composta de um número x de palavras que formavam a resenha de um determinado filme, após a conversão, cada string foi convertida para int, possuindo uma indexação interna para que seja possível aplicar sobre este tipo de dado uma série de ações.

```
1  print(y_treino)
2

[1 0 0 ... 0 1 0]
```

Realizando a mesma verificação para y_treino é possível ver que aqui temos os dados de saída para cada review da matriz anterior, resultando 1 ou 0 de acordo com a característica do review (avaliando positivamente ou negativamente).

```
1  print(len(X_treino[44]))
2

122
```

Verificando o tamanho deste objeto específico por meio da função len( ) repassada como parâmetro de nossa função print( ), obtemos um retorno 122, em outras palavras, uma review composta de 122 palavras.

```
1  X_treino = tf.keras.preprocessing.sequence.pad_sequences(X_treino,
2                                                            maxlen = 100)
3  X_teste = tf.keras.preprocessing.sequence.pad_sequences(X_teste,
4                                                           maxlen = 100)
5
```

Aqui temos um pequeno problema pois cada review é única, inclusive em sua quantidade de palavras que a descreve. Para evitar conflitos, podemos fazer um simples tratamento de dados convertendo todas reviews para um tamanho uniforme.

Isso é feito de forma bastante simples, usando da função pad_sequences( ) parametrizada com os dados de origem e com o tamanho fixo que tais dados deverão ter após conversão. O mesmo é feito para base de teste.

Apenas um pequeno adendo, este valor não respeita nenhum padrão, pois, inclusive devem haver reviews dessa base com menos de 100 palavras, logo, em uma aplicação real podemos testar valores diferentes e acompanhar o comportamento da base de dados ao ser processada.

```
1   modelo = tf.keras.Sequential()
2
3   modelo.add(tf.keras.layers.Embedding(input_dim = 20000,
4                                        output_dim = 128,
5                                        input_shape = (X_treino.shape[1], )))
6   modelo.add(tf.keras.layers.LSTM(units = 128,
7                                   activation = 'tanh'))
8   modelo.add(tf.keras.layers.Dense(units = 1,
9                                    activation= 'sigmoid'))
10  modelo.compile(optimizer = 'rmsprop',
11                 loss = 'binary_crossentropy',
12                 metrics = ['accuracy'])
13
```

Uma vez encerrada a fase de tratamento dos dados, podemos construir nosso modelo de rede neural.

Inicialmente criamos, como de costume, nossa variável modelo, que instancia e inicializa Sequential( ).

Na sequência por meio da função add( ) inserimos o que seria nossa camada de entrada de nossa rede neural. Como parâmetro da função add( ) repassamos tf.keras.layers.Embedding( ), modelo de camada este para uso exclusivo do tipo de base de dados que estamos usando para esse exemplo.

Como parâmetros de Embedding( ) passamos para input_dim o número de dados a serem lidos, da mesma forma para output_dim definimos o número de dados a serem retornados, por fim, definindo via input_shape o formato desses dados de entrada. Para esse exemplo, usando de referência X_treino.shape[1], que em outras palavras significa de nossa base de dados X_treino apenas a coluna referente as saídas (0 ou 1) e não a lista de todos reviews.

Em seguida adicionamos uma camada LSTM, sigla para long short time memory, em tradução livre memória de longo ou curto prazo. Vamos buscar entender o que é esse conceito, pois é de suma importância para esse modelo.

Em redes neurais recorrentes temos um problema computacional que se dá porque em modelos como esse, normalmente usados com dados alimentados baseados em uma série temporal ou histórico, obteremos dados tanto atualizados como defasados. Para contornar isso, usaremos de algumas métricas para que se dê mais valor aos dados recentes do que aos dados desatualizados.

Para essa camada da rede, definimos um número de 128 neurônios, sendo esses ativados por uma função chamada tangente hiperbólica, que justamente à medida que a rede reconhece um padrão correto dá mais valor para esse dado do que o geral, tendendo também a descartar totalmente dados que se mostrem irrelevantes para o processo de aprendizado de máquina.

Dando continuidade, adicionamos nossa camada de saída, que nesse caso é uma camada gerada via Dense( ) com apenas um neurônio de saída, usando como função de ativação a função matemática sigmoid, que por sua vez, considera os dados de maior peso para determinar se com base nos mesmos a saída derá 1 ou 0.

Tenha em mente que aqui não estamos trabalhando com uma simples probabilidade de 0 ou 1, muito pelo contrário, para esse tipo de classificação não existe meio termo, sendo que o resultado, baseado nos dados mais relevantes levantados para retornar como dado de saída 0 ou 1.

Para fase de compilação também teremos algumas pequenas particularidades para esse modelo, sendo a primeira dela que como optimier temos ‘rmsprop’, parâmetro que costuma apresentar melhores resultados para esse modelo de rede neural e para mineração de informações.

Como função de erro usamos ‘binary_crossentropy’ pois como dito anteriormente, não estamos trabalhando com valores de probabilidade, mas valores absolutos para 0 ou 1.

Finalizando, como metrics usamos de [‘accuracy’], reforçando que a métrica usada para tal classificação deverá ser sempre a maior margem de acerto possível.

```
1   print(modelo.summary())
2
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, 100, 128)          2560000
_________________________________________________________________
lstm (LSTM)                  (None, 128)               131584
_________________________________________________________________
dense (Dense)                (None, 1)                 129
=================================================================
Total params: 2,691,713
Trainable params: 2,691,713
Non-trainable params: 0
```

Terminada a criação do modelo podemos ver em seu sumário suas características. Note que para este modelo em particular, em função do grande volume de dados de base, as possíveis interconexões superam dois milhões e meio.

```
1   modelo.fit(X_treino,
2              y_treino,
3              epochs = 50,
4              batch_size = 128,
5              use_multiprocessing = True)
6
```

Encerrando esse exemplo, por meio da função fit( ) aplicada a nosso modelo, parametrizada com os dados de treino, teste, assim como o número de vezes a serem processados pela rede neural.

Aqui temos um parâmetro não explorado manualmente pelos exemplos anteriores chamado batch_size. Este parâmetro em particular está diretamente ligado ao mecanismo de aprendizado de máquina, pois define de quantas em quantas amostras processadas a rede neural deve ter seus pesos (valores de interconexão entre neurônios e camadas) atualizados.

O parâmetro batch_size é crucial para realização de diferentes testes visando sempre obter um modelo de alta performance.

```
Train on 25000 samples
Epoch 1/50
25000/25000 [==============================] - 9s 350us/sample - loss: 0.4576 - accuracy: 0.7849
Epoch 2/50
25000/25000 [==============================] - 5s 218us/sample - loss: 0.2892 - accuracy: 0.8842
Epoch 3/50
25000/25000 [==============================] - 5s 213us/sample - loss: 0.2277 - accuracy: 0.9121
Epoch 4/50
25000/25000 [==============================] - 5s 216us/sample - loss: 0.1847 - accuracy: 0.9310
Epoch 5/50
25000/25000 [==============================] - 6s 222us/sample - loss: 0.1505 - accuracy: 0.9460
Epoch 6/50
```

Ao executar o bloco de código anterior é possível acompanhar via terminal a rede neural realizando seu processamento. Como sempre, é possível notar que a mesma começa com uma baixa margem de acertos, evoluindo rapidamente.

```
25000/25000 [==============================] - 5s 214us/sample - loss: 0.0025 - accuracy: 0.9996
Epoch 45/50
25000/25000 [==============================] - 5s 213us/sample - loss: 0.0019 - accuracy: 0.9994
Epoch 46/50
25000/25000 [==============================] - 5s 217us/sample - loss: 0.0011 - accuracy: 0.9998
Epoch 47/50
25000/25000 [==============================] - 5s 216us/sample - loss: 0.0013 - accuracy: 0.9997
Epoch 48/50
25000/25000 [==============================] - 6s 226us/sample - loss: 0.0017 - accuracy: 0.9995
Epoch 49/50
25000/25000 [==============================] - 6s 226us/sample - loss: 8.7114e-04 - accuracy: 0.
Epoch 50/50
25000/25000 [==============================] - 5s 218us/sample - loss: 0.0017 - accuracy: 0.9995
<tensorflow.python.keras.callbacks.History at 0x7fa76de44780>
```

Minutos depois do início de processamento podemos ver que época após época a rede neural aumentou sua margem de acertos, estabilizando em 99%.

Código Completo:

```python
import tensorflow as tf
from tensorflow.keras.datasets import imdb

(X_treino, y_treino), (X_teste, y_teste) = imdb.load_data(num_words=20000)

print(X_treino.shape)
print(X_treino)
print(X_treino[44])

print(y_treino)

print(len(X_treino[44]))

X_treino = tf.keras.preprocessing.sequence.pad_sequences(X_treino,
                                                          maxlen = 100)
X_teste = tf.keras.preprocessing.sequence.pad_sequences(X_teste,
                                                         maxlen = 100)

print(len(X_treino[44]))


```

```python
modelo = tf.keras.Sequential()

modelo.add(tf.keras.layers.Embedding(input_dim = 20000,
                                     output_dim = 128,
                                     input_shape = (X_treino.shape[1], )))
modelo.add(tf.keras.layers.LSTM(units = 128,
                                activation = 'tanh'))
modelo.add(tf.keras.layers.Dense(units = 1,
                                 activation= 'sigmoid'))
modelo.compile(optimizer = 'rmsprop',
               loss = 'binary_crossentropy',
               metrics = ['accuracy'])

print(modelo.summary())

modelo.fit(X_treino,
           y_treino,
           epochs = 50,
           batch_size = 128,
           use_multiprocessing = True)

```

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

```
import pandas as pd
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.

```
base = pd.read_csv('wines.csv')
entradas = base.iloc[:, 1:14].values
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.

```
from sklearn.preprocessing import MinMaxScaler

```

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).

```
normalizador = MinMaxScaler(feature_range = (0,1))
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. $5 * \sqrt{n}$, onde n é o número de entradas.

Nesse nosso exemplo, temos 178 entradas, dessa forma, $5 * \sqrt{178}$ 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.

```
from pylab import pcolor, colorbar

pcolor(som.distance_map().T)
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:

```python
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.

```
rbm.train(base,
          max_epochs = 3000)
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.

```
usuario = np.array([[1,1,0,1,0,0]])

```

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.

```
rbm.run_visible(usuario)
camada_oculta = np.array([[0,1]])
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.

```
for i in range(len(usuario[0])):
    print(usuario[0,i])
    if usuario[0,i] == 0 and recomendacao[0,i] == 1:
        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:

```python
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']
```

```python
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**.

```
Administrador: Anaconda Prompt (Anaconda3) - conda  activate root

(base) C:\Windows\system32>conda activate AITRADE

(AITRADE) C:\Windows\system32>_
```

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**.

```
Administrador: Anaconda Prompt (Anaconda3) - conda  activate root

(AITRADE) C:\Windows\system32>pip install tensorflow==1.15_
```

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.

```
Administrador: Anaconda Prompt (Anaconda3) - conda  activate root
  Using cached astor-0.8.1-py2.py3-none-any.whl (27 kB)
Processing c:\users\fernando\appdata\local\pip\cache\wheels\3f\e3\ec\8
termcolor-1.1.0-py3-none-any.whl
Collecting protobuf>=3.6.1
  Using cached protobuf-3.11.3-cp37-cp37m-win_amd64.whl (1.0 MB)
Collecting keras-applications>=1.0.8
  Using cached Keras_Applications-1.0.8-py3-none-any.whl (50 kB)
Processing c:\users\fernando\appdata\local\pip\cache\wheels\21\7f\02\4
gast-0.2.2-py3-none-any.whl
Collecting werkzeug>=0.11.15
  Using cached Werkzeug-1.0.1-py2.py3-none-any.whl (298 kB)
Requirement already satisfied: setuptools>=41.0.0 in c:\programdata\an
rboard<1.16.0,>=1.15.0->tensorflow==1.15) (46.1.3.post20200330)
Collecting markdown>=2.6.8
  Using cached Markdown-3.2.1-py2.py3-none-any.whl (88 kB)
Collecting h5py
  Using cached h5py-2.10.0-cp37-cp37m-win_amd64.whl (2.5 MB)
Installing collected packages: numpy, opt-einsum, werkzeug, six, grpci
-preprocessing, tensorflow-estimator, google-pasta, astor, termcolor,
Successfully installed absl-py-0.9.0 astor-0.8.1 gast-0.2.2 google-pas
ons-1.0.8 keras-preprocessing-1.1.0 markdown-3.2.1 numpy-1.18.2 opt-ei
1.15.0 tensorflow-1.15.0 tensorflow-estimator-1.15.1 termcolor-1.1.0 w

(AITRADE) C:\Windows\system32>
```

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.

```
Administrador: Anaconda Prompt (Anaconda3) - conda  activate root

(AITRADE) C:\Windows\system32>pip install pandas-datareader
```

O processo é exatamente o mesmo da instalação anterior, apenas alterando o comando para **pip install pandas-datareader**.

```
Administrador: Anaconda Prompt (Anaconda3) - conda  activate root

(AITRADE) C:\Windows\system32>pip install matplotlib tqdm
```

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.

```
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

```

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.

```
from tqdm import tqdm_notebook, tqdm
from pandas.util.testing import *
from collections import deque

```

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.

Data	Abrir	Alto	Baixo	Fechamento*	Fechamento ajustado**	Volume
20 de abr. de 2020	78.988,64	80.105,99	76.942,89	78.950,14	78.950,14	-
17 de abr. de 2020	77.817,00	79.846,00	77.754,00	78.990,00	78.990,00	8.317.800
16 de abr. de 2020	78.837,00	80.167,00	77.452,00	77.812,00	77.812,00	9.028.400

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.

```
print(dataset.head())
print(str(dataset.index[0]).split()[0])
print(dataset.index[-1])
print(dataset['Close'])
```

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.

```
85  def state_creator(data, timestep, window_size):
86      starting_id = timestep - window_size + 1
87
88      if starting_id >= 0:
89          windowed_data = data[starting_id:timestep + 1]
90      else:
91          windowed_data = - starting_id * [data[0]] + list(data[0:timestep + 1])
92
93      state = []
94      for i in range(window_size - 1):
95          state.append(sigmoid(windowed_data[i + 1] - windowed_data[i]))
96
97      return np.array([state]), windowed_data
98
```

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.

```
99    stock_name = "AAPL"
100   data = dataset_loader(stock_name)
101   s = state_creator(data, 0, 5)
102   window_size = 10
103   episodes = 1000
104   batch_size = 32
105   data_samples = len(data) - 1
106
```

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.

```
AI Trader vendeu:  R$ 153.13 Lucro de: R$ 2.58
 49%|          | 614/1257 [01:24<01:31,  7.06it/s]AI Trader comprou:  R$ 154.22
AI Trader vendeu:  R$ 153.27 Lucro de: - R$ 0.94
 49%|          | 620/1257 [01:25<01:27,  7.25it/s]AI Trader comprou:  R$ 155.38
AI Trader comprou:  R$ 155.30
 49%|          | 622/1257 [01:25<01:28,  7.18it/s]AI Trader comprou:  R$ 155.83
AI Trader vendeu:  R$ 155.89 Lucro de: R$ 0.50
 50%|          | 624/1257 [01:25<01:28,  7.14it/s]AI Trader comprou:  R$ 156.55
AI Trader vendeu:  R$ 156.00 Lucro de: R$ 0.69
 50%|          | 626/1257 [01:25<01:29,  7.08it/s]AI Trader comprou:  R$ 156.99
AI Trader vendeu:  R$ 159.88 Lucro de: R$ 4.04
 50%|          | 629/1257 [01:26<01:29,  7.05it/s]AI Trader vendeu:  R$ 159.75
de: R$ 3.20
 50%|          | 631/1257 [01:26<01:28,  7.08it/s]AI Trader comprou:  R$ 156.25
AI Trader comprou:  R$ 156.16
 50%|          | 633/1257 [01:26<01:28,  7.01it/s]AI Trader comprou:  R$ 157.10
AI Trader vendeu:  R$ 156.41 Lucro de: - R$ 0.58
```

Resultados melhorando a medida que o Agente evolui.

```
 99%|          | 1239/1257 [02:58<00:02,  7.06it/s]AI Trader vendeu:  R$ 246.88
de: - R$ 28.54
AI Trader vendeu:  R$ 245.52 Lucro de: - R$ 32.44
 99%|          | 1248/1257 [02:59<00:01,  7.08it/s]AI Trader comprou:  R$ 262.4
AI Trader vendeu:  R$ 259.42 Lucro de: - R$ 3.04
 99%|          | 1250/1257 [02:59<00:00,  7.10it/s]AI Trader comprou:  R$ 266.0
AI Trader comprou:  R$ 267.98
100%|          | 1252/1257 [03:00<00:00,  7.03it/s]AI Trader comprou:  R$ 273.2
AI Trader comprou:  R$ 287.04
100%|          | 1254/1257 [03:00<00:00,  7.07it/s]AI Trader vendeu:  R$ 284.42
de: R$ 18.35
100%|          | 1256/1257 [03:00<00:00,  7.16it/s]AI Trader vendeu:  R$ 282.79
de: R$ 14.80
AI Trader vendeu:  R$ 276.92 Lucro de: R$ 3.67
########################
Lucro Total Estimado: 9850.96
########################
100%|          | 1257/1257 [03:00<00:00,  6.95it/s]
```

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.

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.

```
    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)

```

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( ).

```
    def normalize(self, inputs):
        obs_mean = self.mean
        obs_std = np.sqrt(self.var)
        return (inputs - obs_mean) / obs_std

```

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.

```
class Policy():
    def __init__(self, input_size, output_size):
        self.theta = np.zeros((output_size, input_size))

```

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.

```
    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)

```

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.

```
    def sample_deltas(self):
        return [np.random.randn(*self.theta.shape) for _ in range(hp.nb_directions)]

```

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.

```
    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_r) * step

```

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.

```
        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]) for k in order]
        policy.update(rollouts, sigma_r)
        reward_evaluation = explore(env, normalizer, policy)
        print('Step:', step, 'Reward:', reward_evaluation)

```

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.

```python
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')

```

Em seguida é realizada uma pequena codificação para criação de um diretório onde serão salvos algumas amostras de nosso modelo em operação.

```python
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)

```

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.

openaigym.video.0.30883.video000000.mp4
openaigym.video.0.30883.video000001.mp4
openaigym.video.0.30883.video000008.mp4
openaigym.video.0.30883.video000027.mp4
openaigym.video.0.30883.video000064.mp4
openaigym.video.0.30883.video000125.mp4
openaigym.video.0.30883.video000216.mp4
openaigym.video.0.30883.video000343.mp4
openaigym.video.0.30883.video000512.mp4
openaigym.video.0.30883.video000729.mp4
openaigym.video.0.30883.video001000.mp4
openaigym.video.0.30883.video002000.mp4
openaigym.video.0.30883.video003000.mp4
openaigym.video.0.30883.video004000.mp4
openaigym.video.0.30883.video005000.mp4
openaigym.video.0.30883.video006000.mp4
openaigym.video.0.30883.video007000.mp4
openaigym.video.0.30883.video008000.mp4
openaigym.video.0.30883.video009000.mp4
openaigym.video.0.30883.video010000.mp4
openaigym.video.0.30883.video011000.mp4
openaigym.video.0.30883.video012000.mp4
openaigym.video.0.30883.video013000.mp4
openaigym.video.0.30883.video014000.mp4
openaigym.video.0.30883.video015000.mp4
openaigym.video.0.30883.video016000.mp4
openaigym.video.0.30883.video017000.mp4
openaigym.video.0.30883.video018000.mp4
openaigym.video.0.30883.video019000.mp4
openaigym.video.0.30883.video020000.mp4
openaigym.video.0.30883.video021000.mp4
openaigym.video.0.30883.video022000.mp4
openaigym.video.0.30883.video023000.mp4
openaigym.video.0.30883.video024000.mp4
openaigym.video.0.30883.video025000.mp4
openaigym.video.0.30883.video026000.mp4
openaigym.video.0.30883.video027000.mp4

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.

```
Administrador: Anaconda Prompt (anaconda3)

(base) C:\Windows\system32>conda create --name AIBreakout
```

Sendo assim, via prompt Anaconda por meio do Código acima é criado o ambiente virtual isolado AIBreakout.

```
Administrador: Anaconda Prompt (anaconda3)

(base) C:\Windows\system32>conda activate AIBreakout

(AIBreakout) C:\Windows\system32>
```

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.

```
Administrador: Anaconda Prompt (anaconda3)

(base) C:\Windows\system32>conda activate AIBreakout

(AIBreakout) C:\Windows\system32>pip install numpy
```

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!!!

```
import os
import random
import gym
import tensorflow as tf
import numpy as np
import imageio
from skimage.transform import resize

```

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.

```
ENV_NAME = 'BreakoutDeterministic-v3'

```

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.

```
80  class ExplorationExploitationScheduler(object):
81      def __init__(self, DQN, n_actions, eps_initial=1, eps_final=0.1, eps_final_frame=0.01,
82                   eps_evaluation=0.0, eps_annealing_frames=1000000,
83                   replay_memory_start_size=50000, max_frames=25000000):
84          self.n_actions = n_actions
85          self.eps_initial = eps_initial
86          self.eps_final = eps_final
87          self.eps_final_frame = eps_final_frame
88          self.eps_evaluation = eps_evaluation
89          self.eps_annealing_frames = eps_annealing_frames
90          self.replay_memory_start_size = replay_memory_start_size
91          self.max_frames = max_frames
92          self.slope = -(self.eps_initial - self.eps_final)/self.eps_annealing_frames
93          self.intercept = self.eps_initial - self.slope*self.replay_memory_start_size
94          self.slope_2 = -(self.eps_final - self.eps_final_frame)/
95                         (self.max_frames - self.eps_annealing_frames - self.replay_memory_start_size)
96          self.intercept_2 = self.eps_final_frame - self.slope_2*self.max_frames
97          self.DQN = DQN
98
```

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.

```
        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
```

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.

```
    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
```

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’.

```
LAYER_IDS = ["conv1", "conv2", "conv3", "conv4", "denseAdvantage",
             "denseAdvantageBias", "denseValue", "denseValueBias"]

```

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.

```
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)

```

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.

```
    with tf.Session() as sess:
        sess.run(init)
        frame_number = 0
        rewards = []
        loss_list = []

```

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.

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.

PYTHON + TENSORFLOW 2.X

TensorFlow

A biblioteca a qual iremos nos focar totalmente neste pequeno livro é a biblioteca TensorFlow, biblioteca esta desenvolvida por engenheiros da Google Brain, tendo sua primeira versão distribuída para domínio público em meados de 2015.

Tratando-se de modelos de redes neurais artificiais, a biblioteca TensorFlow veio com uma proposta diferente das demais se baseando em estruturas lógicas chamadas tensores, onde por meio destes são realizados processamentos sobre fluxos de dados.

Diferentemente de outras bibliotecas com outras arquiteturas de redes neurais artificiais, o TensorFlow se mostrou notavelmente mais eficiente quando comparado com as demais, não somente em questões de performance, mas fácil remodelagem para diferentes tipos de redes neurais assim como grande eficiência indiscriminada, ou em outras palavras, alta taxa de precisão seja para um simples perceptrons, seja para problemas computacionais de alta complexidade tratados em deep learning.

Como exploraremos nos capítulos subsequentes, é bastante fácil e intuitivo criarmos nossos modelos de redes neurais usando do TensorFlow e de suas ferramentas.

Para maiores detalhes sobre todo referencial teórico em torno de ciência de dados e aprendizado de máquina, recomendo a leitura de meu livro Redes Neurais Artificiais.

O intuito deste livro é de forma prática e descomplicada entender todos os detalhes envolvidos na criação de uma rede neural artificial por meio da biblioteca TensorFlow, sendo assim, toda teoria envolvida será abordada diretamente contextualizada para o código usado como exemplo.

Instalação do TensorFlow 2

A biblioteca TensorFlow, embora já muito bem estabelecida e muito usada para criação de redes neurais artificiais, não faz parte das bibliotecas, módulos e pacotes nativos do Python.

Sendo assim, é necessário realizar a instalação desta biblioteca para nosso ambiente virtualizado de desenvolvimento, independentemente do sistema operacional o qual você esteja usando.

O processo de instalação também pode variar de acordo com seu ambiente virtualizado o qual será usado.

Em outras palavras, basicamente antes mesmo de partir para a instalação de fato devemos considerar se usaremos do TensorFlow no ambiente virtualizado padrão do Windows, onde sua instalação se dará via pip ou em um ambiente virtualizado pela suíte Anaconda, onde a instalação se dará via o gerenciador de pacotes conda.

Por fim, a biblioteca TensorFlow em todas suas versões estáveis a partir da 2.X possui seus pacotes integrados independentemente da a forma de processamento, seja ela via CPU ou via GPU, e para nossos exemplos estaremos usando de versões a partir desta. Apenas contextualizando, para versões 1.X da biblioteca era necessário instalar versões diferentes para CPU e/ou para GPU.

Para instalação no ambiente virtualizado padrão do Windows basta executar no Prompt de Comando (com privilégios de

Administrador) o comando **pip install tensorflow** e o processo de instalação será iniciado, não só para o TensorFlow em sua última versão estável (no momento em que escrevo este livro, versão 2.4.0, de dezembro de 2020) mas todas suas dependências.

```
Administrador: Anaconda Powershell Prompt (anaconda3)
(base) PS C:\Windows\system32> conda create -n tensorflow python=3.7 anaconda
Collecting package metadata (current_repodata.json): done
Solving environment: failed with repodata from current_repodata.json, will re
Collecting package metadata (repodata.json): done
Solving environment: done

## Package Plan ##

  environment location: C:\Users\Fernando\anaconda3\envs\tensorflow
```

Para instalação em um ambiente virtualizado Anaconda, é recomendável a criação do mesmo do zero para que tenhamos uma instalação limpa livre de conflitos e incompatibilidades.

Para isso, via Anaconda Powershell Prompt (com privilégios de Administrador) basta executar o comando **conda create -n nome_do_seu_ambiente python=3.7 anaconda**.

Dessa forma um ambiente virtualizado será criado totalmente isolado dos demais, contando com um núcleo Python exclusivo além é claro, das bibliotecas, módulos e pacotes que serão instalados neste ambiente.

```
Administrador: Anaconda Powershell Prompt (anaconda3)
(tensorflow) PS C:\Windows\system32> conda install -c anaconda tensorflow
Collecting package metadata (current_repodata.json): done
Solving environment: failed with initial frozen solve. Retrying with flex
Solving environment: failed with repodata from current_repodata.json, wil
Collecting package metadata (repodata.json): done
Solving environment: done

## Package Plan ##

  environment location: C:\Users\Fernando\anaconda3\envs\tensorflow

  added / updated specs:
    - tensorflow
```

Em seguida, após ativar seu novo ambiente de desenvolvimento (via comando: **conda activate nome_do_seu_ambiente**) podemos fazer a devida instalação da biblioteca TensorFlow.

Como aqui estamos usando do repositório CondaForge, o processo é feito pelo comando **conda install -c anaconda tensorflow**. Executando esse comando será iniciado o download do TensorFlow assim como de suas dependências.

Ao término da instalação já é possível abrir sua IDE de preferência e executar seus primeiros códigos para o TensorFlow.

Usando TensorFlow via Google Colaboratory

Uma das formas comumente usadas para aplicações de pequena complexidade é usar da plataforma Colab do Google, dessa forma, em nuvem temos a disposição processamento via GPU assim como integração com nosso Google Drive.

Para usar desta plataforma basta acessar o Google Colaboratory estando logado com sua conta Google, em seguida criar um novo notebook para a partir de células executar seus blocos de código individualmente.

```
1  !pip install tensorflow-gpu==2.1.0-rc0

Requirement already satisfied: tensorflow-gpu==2.1.0-rc0 in /usr/local/lib/
Requirement already satisfied: tensorflow-estimator<2.1.0,>=2.0.0 in /usr/l
Requirement already satisfied: absl-py>=0.7.0 in /usr/local/lib/python3.6/c
Requirement already satisfied: opt-einsum>=2.3.2 in /usr/local/lib/python3.
Requirement already satisfied: wheel>=0.26; python_version >= "3" in /usr/l
Requirement already satisfied: tensorboard<2.1.0,>=2.0.0 in /usr/local/lib/
Requirement already satisfied: keras-preprocessing>=1.1.0 in /usr/local/lib
Requirement already satisfied: keras-applications>=1.0.8 in /usr/local/lib/
Requirement already satisfied: google-pasta>=0.1.6 in /usr/local/lib/pythor
Requirement already satisfied: astor>=0.6.0 in /usr/local/lib/python3.6/dis
Requirement already satisfied: gast==0.2.2 in /usr/local/lib/python3.6/dist
```

Por meio do comando **!pip install tensorflow** é feita a instalação assim como nos ambientes virtualizados para Windows.

Apenas como exemplo, para instalar uma versão específica do TensorFlow no Colab basta especificar que versão deverá ser instalada.

```
1  import tensorflow as tf
2
```

```
1  tf.__version__
2
```

```
'2.1.0-rc0'
```

Uma vez terminada a instalação basta importar normalmente a biblioteca tensorflow por meio do comando import tensorflow. Neste caso, apenas por convenção, estamos referenciando a mesma como tf.

Executando em uma nova célula o comando **tf.__version__** nos é retornada a versão a qual estaremos fazendo uso.

CONCEITOS BÁSICOS

Tensores

Quando estamos falando de redes neurais artificias temos uma vasta gama de arquiteturas desenvolvidas para os mais diversos propósitos, cada uma com suas particularidades no que diz respeito aos meios e métodos usados para resolver certos problemas computacionais.

Os modelos criados a partir do TensorFlow não fogem a essa regra, uma vez que sua arquitetura é única se comparado com outras bibliotecas de matemática computacional. Usando de arquiteturas de código desenvolvidas pelos engenheiros da Google Brain baseados em tensores para fluxos de informação, sendo que nesses moldes a biblioteca TensorFlow se sobressaiu em praticidade e performance quando comparada a outras.

Sendo assim, um dos primeiros conceitos que devemos explorar é justamente esse sistema de tensores, estruturas lógicas para processamento de dados de forma modularizada.

Bibliotecas científicas normalmente usam de tipos e dados em padrão escalar, vetorial e/ou matricial dependendo de outras ferramentas para padronizar e permitir o cruzamento de dados. Já a biblioteca TensorFlow usa destes tipos de dados internamente sem distinções, apenas os tratando como matrizes multidimensionais.

Dessa forma o mapeamento de certos dados para dimensões diretamente aplicada aos tensores do modelo acaba por realizar o processamento de tais dados em busca de seus padrões e todo aprendizado de máquina de modo mais eficiente tanto em performance quanto em margem de precisão em processos de classificação e/ou regressão.

Apenas fazendo uma ressalva, o modelo teórico de um processamento de dados em camadas de matrizes não é algo novo, exames de tomografia computadorizada, por exemplo, trabalham desde a década de 60 com imagens geradas em pixels e voxels para renderizar volume em imagem. O que outras bibliotecas não haviam feito até então era usar deste modelo teórico para

transformar tal estrutura em camadas de processamento em uma rede neural artificial.

Sendo assim, em resumo podemos inicialmente entender que o sistema de tensores nada mais é do que uma organização de dados modulares executados em sessões (inclusive permitindo computação paralela) onde um fluxo progressivo de informações é processado por meio de tensores, invólucros independentes que podem conter diversas estruturas de dados e funções a serem executadas.

Os tipos de dados que serão apresentados em seguida, variáveis e constantes, guardarão dados/valores atribuídos para si e toda e qualquer operação com estes tipos de dados serão feitas por meio de tensores. Este conceito pode parecer um tanto quanto confuso inicialmente, porém na prática você entenderá facilmente estas distinções.

Como nosso foco neste livro é uma abordagem prática, muitos dos conceitos envolvidos também serão devidamente explicados à medida que aplicamos os mesmos em nossos exemplos.

Constantes e Variáveis

Uma vez que temos nossos contêineres de processamento de dados em fluxo chamados tensores, outras estruturas de dados que estarão interagindo diretamente com nossos tensores são as chamadas constantes e variáveis.

Por hora, raciocine que no processo de criação de nossa rede neural artificial via tensorflow teremos sempre alguns dados de entrada, estes por sua vez podem ter valores predefinidos fixos (constantes) assim como valores iniciais dinâmicos (variáveis), que sofrerão alterações à medida que são instanciados por algum tensor.

Existe uma terceira estrutura relacionada a variáveis chamada placeholder, que por sua vez nada mais é do que um marcador que reserva um espaço o qual será alimentado com novos dados no decorrer da execução da rede neural.

Sendo assim, nas estruturas de código teremos tais tipos de dados demarcados em nós que irão interagir com/em tensores conforme a rede demande o uso destes dados.

Podemos fazer um paralelo com uma arquitetura de rede neural convencional apenas para que fique claro tais nomenclaturas. Raciocine que em um modelo convencional, equivalente aos neurônios da camada de entrada de uma rede neural artificial teremos nossas constantes.

Da mesma forma, entre camadas de neurônios, onde eram realizados os cálculos e ajustes dos pesos, no tensorflow teremos nossas variáveis.

Por fim, em redes neurais que guardam um estado inicial zerado/nulo, por meio de mecanismos de aprendizado de máquina como aprendizado por reforço, a rede retroalimenta essa memória a cada fase de execução da rede guardando o último estado apenas atualizando o mesmo, inicialmente vazio, com novos dados conforme a rede neural aprende.

Sessões

Encerrando o básico do conceitual teórico inicial do TensorFlow, uma das estruturas de processamento de dados mais básicas desta biblioteca é a chamada sessão.

Novamente, este é apenas um conceito inicial para um primeiro contato, porém, na prática será muito mais interessante de se entender de fato o que é uma sessão em TensorFlow.

Basicamente podemos dizer que uma sessão, desculpe a redundância, é de fato uma sessão criada para que neste espaço sejam de fato executadas as ações de nossos blocos de código.

As estruturas entendidas até então como tensores, variáveis e constantes, são estruturas de código onde abstraímos tudo o que é necessário em forma de código e internamente o interpretador do TensorFlow organiza tais estruturas de dados como elementos de uma equação a ser executada, porém o processamento de tais dados efetivamente acontece ao abrirmos uma sessão que instancia os mesmos.

Em outras palavras, graças a estrutura modular do TensorFlow, uma determinada arquitetura pode ser desmembrada em pequenos blocos que podem inclusive serem processados separadamente e paralelamente por um ou mais hardwares dedicados.

Neste processo de se modularizar tudo, cada estrutura de dado, cada método, cada interação entre dados pode ser separada não somente em blocos de código, mas módulos de processamento individual, de forma que apenas quando necessário, os módulos são ativados e postos para interagir uns com os outros.

Imagine um daqueles brinquedos modulares muito usados para o aprendizado básico em robótica, você tem um núcleo básico e ele funciona sozinho por si só, a medida que quer implementar maiores funcionalidades ao robô basta adicionar as devidas peças e elas

imediatamente se integram ao sistema e começam a executar suas ações.

A arquitetura interna desenvolvida pelos engenheiros da Google Brain funciona muito pautada neste tipo de modelo, onde podemos ter um modelo de rede neural artificial ao mesmo tempo muito simples como muito robusta, apenas adicionando mais funcionalidade modularizadas a arquitetura da rede neural.

Apenas encerrando essa linha de raciocínio, tenha em mente que para as versões a partir da 2.X da biblioteca TensorFlow todo esse processo é realizado de forma implícita, versões anteriores a esta demandavam que o usuário criasse manualmente cada sessão para cada fluxo de dado.

PRÁTICA COM TENSORFLOW 2

Ajuda Inline

Em nossos exemplos, a cada recurso sendo implementado será feita a devida explicação sobre o mesmo, porém, é de se esperar que as bibliotecas, módulos e pacotes que faremos uso ofereçam uma grande quantidade de ferramentas que na maioria das vezes não são utilizadas.

Das ferramentas utilizadas, cada uma delas terá também seus próprios parâmetros, sendo boa parte deles implícitos ou pré-definidos pela própria função.

Dependendo da IDE que você estiver usando será possível consultar em tempo real um campo com breve explicações sobre quais parâmetros tal método/função possui assim como sua descrição, de modo que para certas aplicações será interessante consultar este tipo de informação para realizar configurações adicionais e/ou ajustes finos em nossos modelos de redes neurais artificiais.

```
pd.DataFrame()
```

```
def __init__(data=None, index: Optional[Axes]=None, columns:
Optional[Dtype]=None, copy: bool=False)
```

Two-dimensional, size-mutable, potentially heterogeneous tabular data.

Data structure also contains labeled axes (rows and columns). Arithmetic operations align on both row and column labels. Can be thought of as a dict-like container for Series objects. The primary pandas data structure.

Parameters

data : ndarray (structured or homogeneous), Iterable, dict, or DataFrame
    Dict can contain Series, arrays, constants, or list-like objects.

index : Index or array-like
    Index to use for resulting frame. Will default to RangeIndex if
    no indexing information part of input data and no index provided.

columns : Index or array-like
    Column labels to use for resulting frame. Will default to
    RangeIndex (0, 1, 2, ..., n) if no column labels are provided.

dtype : dtype, default None

Apenas como exemplo, instanciando a biblioteca Pandas, chamando sua função DataFrame( ) é possível ver todos possíveis parâmetros e suas respectivas descrições.

```
train_test_split()
```

```
def train_test_split(*arrays, **options)
```

Split arrays or matrices into random train and test subsets

Quick utility that wraps input validation and `next(ShuffleSplit().split(X, y))` and application to input data into a single call for splitting (and optionally subsampling) data in a oneliner.

Read more in the `User Guide <cross_validation>` .

Parameters

*arrays : sequence of indexables with same length / shape[0]
Allowed inputs are lists, numpy arrays, scipy-sparse matrices or pandas dataframes.

test_size : float, int or None, optional (default=None)
If float, should be between 0.0 and 1.0 and represent the proport of the dataset to include in the test split. If int, represents the absolute number of test samples. If None, the value is set to the complement of the train size. If `train_size` is also None, it will be set to 0.25

Outro exemplo, consultando os parâmetros e suas descrições da função train_test_split( ) da biblioteca SKLearn.

Operações Básicas com Tensores

Partindo para prática, vamos entender por meio de exemplos como criamos certas estruturas de dados usando da biblioteca TensorFlow para o processamento dos mesmos. Lembrando que muito do que será visto se dará pela sintaxe usual do Python uma vez que estamos integrando o uso do TensorFlow com bibliotecas já conhecidas como Numpy, por exemplo.

Definindo uma Constante

```
1  tensor = tf.constant([[23, 4], [5, 71]])
2
```

Para definir uma constante em TensorFlow usaremos do método tf.constant( ) que deverá ser parametrizado com o devido dado/valor, tendo seu conteúdo atribuído a uma variável.

Inicialmente declaramos uma variável de nome tensor que por sua vez por meio do método tf.constant( ) cria uma matriz 2X2 nos moldes de uma array numpy.

```
1  print(tensor)

tf.Tensor(
[[23  4]
 [ 5 71]], shape=(2, 2), dtype=int32)
```

Por meio de nossa função print( ) parametrizada com o conteúdo da variável tensor, podemos visualizar o conteúdo da mesma.

Note que o retorno em si possui algumas particularidades, primeira delas é que nos é retornado uma classe tf.Tensor( ), com os dados inseridos anteriormente, seguido de shape=( ) que nos mostra o formato desta matriz e dtype= que nos retornará o tipo de dado em si.

Perceba que criamos uma constante, ao menos na forma como declaramos tal tipo de dado por meio do método referente, porém a partir do momento que este dado é criado, passa a estar encapsulado como um tensor, para que se possa fazer uso do mesmo para funções ou operações entre tensores.

Tudo em TensorFlow que de alguma forma permitirá ser instanciado e usado será internamente tratado por nosso interpretador como um tensor.

```
1  print(type(tensor))

<class 'tensorflow.python.framework.ops.EagerTensor'>
```

Verificando o tipo de dado por meio da função base do Python type( ) nos é retornado o tipo de objeto em questão, nesse caso, um EagerTensor.

Por hora raciocine que este, apesar de estruturalmente muito parecido com uma array do tipo numpy, é um tipo de dado interno exclusivo ao TensorFlow, que integra compatibilidade entre as versões 1.X e 2.X do TensorFlow além de dados matriciais importados de bibliotecas como Pandas e Numpy.

Acessando as Dimensões de um Tensor

```
1  tensor.shape
2

TensorShape([2, 2])
```

Partindo do pressuposto que temos um tensor qualquer, podemos verificar o formato do mesmo diretamente pelo método tensor.shape. Neste caso o retorno é um TensorShape([]) com o referido tamanho, no nosso exemplo, uma matriz de duas linhas e duas colunas.

Acessando os Dados de um Tensor

```
tensor.numpy()

array([[23,  4],
       [ 5, 71]], dtype=int32)
```

Até o momento você deve ter reparado que nossos retornos em si não mostraram de fato o conteúdo de nosso tensor de modo que possarmos interagir com o mesmo.

Pois bem, raciocine que até então, todo e qualquer tipo de dado tratado como um tensor ele permanece estático, de modo que não incorre em nenhum processamento sobre o mesmo. Isso se dá porque via TensorFlow, para de fato executar uma determinada ação/função com nossos tensores os mesmos deverão ser instanciados dentro de uma outra estrutura chamada sessão.

Por hora, enquanto estamos apenas entendendo as estruturas de dados em si, visualize as mesmas em seu contexto padrão.

Supondo que tenhamos um tensor genérico qualquer, uma das formas que temos de visualizar o conteúdo do mesmo é o tratando como uma array do tipo numpy.

Por meio do método numpy( ) aplicado a nossa variável tensor é possível visualizar os dados dessa forma. Lembrando que dessa forma estamos apenas visualizando os dados, não os convertendo para uma array do tipo numpy.

O retorno, como esperado, é o retorno padrão que teríamos via biblioteca numpy, onde os dados são exibidos em formato array.

Convertendo uma Array Numpy em Tensor

```
tensorn = np.array([[23,  4], [32, 51]])
```

Como visto no exemplo anterior, é possível visualizar o conteúdo de um tensor como se o mesmo fosse uma array do tipo

numpy. Para transformar um tipo de dado em outro, efetivamente e permanentemente é necessário realizar a conversão.

Supondo que temos nossa variável tensorn e atribuído a ela uma array numpy de duas dimensões com alguns valores genéricos.

```
1  tensor_numpy = tf.constant(tensorn)
2
```

Podemos facilmente realizar a conversão por meio do método tf.constant( ). Ao parametrizar este método com o conteúdo de nossa variável tensorn, automaticamente o mesmo é convertido para uma constante que por sua vez estará encapsulado em um tensor para que possamos fazer o uso do mesmo cruzando dados com outros tensores.

```
1  tensor_numpy
2
```

```
<tf.Tensor: shape=(2, 2), dtype=int64, numpy=
array([[23,  4],
       [32, 51]])>
```

Exibindo em tela o conteúdo de nosso tensor_numpy, agora temos o conteúdo oriundo de tensorn em formato tf.Tensor.

Definindo uma Variável

```
1  tf_variavel = tf.Variable([[1., 2., 3.], [4., 5., 6.]])
2
3  tf_variavel
4
```

De forma parecida como definimos uma constante, realizaremos a criação de uma variável TensorFlow. Lembrando que este tipo de dado terá algumas particularidades, sendo a principal dela a flexibilização da manipulação de seus dados a qualquer momento seja no escopo global ou por meio de um tensor em sua sessão.

Inicialmente criamos uma variável de nome tf_variavel que por sua vez instancia o método tf.Variable( ), parametrizando o mesmo com os dados em forma matricial em questão.

Em seguida já podemos visualizar estes dados diretamente chamando a variável por seu nome ou colocando-a como parâmetro de uma função print( ).

```
1  tf_variavel
2
```

```
<tf.Variable 'Variable:0' shape=(2, 3) dtype=float32, numpy=
array([[1., 2., 3.],
       [4., 5., 6.]], dtype=float32)>
```

O retorno, neste caso, é uma classe tf.Variable. Repare que neste caso em particular, os dados são guardados diretamente em formato array numpy.

Em outras palavras, o que nos permitirá uma fácil manipulação destes dados é o fato de que internamente uma variável em TensorFlow guarda seus dados em formato array, permitindo dessa forma toda e qualquer operação aplicável a uma array.

Realizando apenas um pequeno adendo, repare que no processo de criação de uma variável, por questões de compatibilidade, o tipo de dado ao qual os dados/valores são convertidos passa a ser float32.

```
1  tf_variavel.numpy()
```

```
array([[1., 2., 3.],
       [4., 5., 6.]], dtype=float32)
```

Haja visto que temos os dados de uma variável já em formato array numpy, podemos perfeitamente visualizar seus dados por meio do método numpy( ) aplicado a nossa variável tf_variavel.

Como retorno temos os devidos dados apresentados como dados matriciais do tipo array numpy.

Manipulando Dados/Valores em uma Variável

```
1  tf_variavel
```

```
<tf.Variable 'Variable:0' shape=(2, 3) dtype=float32, numpy=
array([[1., 2., 3.],
       [4., 5., 6.]], dtype=float32)>
```

Continuando com a mesma variável do exemplo anterior, vamos entender como se dá o processo de manipulação dos dados/valores desta estrutura de dados.

```
1  tf_variavel[0, 2].assign(100)
```

```
<tf.Variable 'UnreadVariable' shape=(2, 3) dtype=float32, numpy=
array([[  1.,   2., 100.],
       [  4.,   5.,   6.]], dtype=float32)>
```

Uma vez que temos em mãos dados em um formato matricial, podemos normalmente instanciar estes dados para todo e qualquer uso desde que tenhamos o referencial de seu índice.

Trabalhando diretamente com as funções da biblioteca TensorFlow, podemos facilmente com o número de índice e a função assign( ) alterar um determinado dado/valor. Apenas salientando que neste caso, por meio do método assign( ) estaremos alterando permanentemente este determinado dado/valor.

Em nossa variável tf_variavel, instanciando a posição de índice [0, 2], ou seja, linha zero coluna 2, aplicando o método assign( ) parametrizado com o valor 100 estaremos substituindo o valor padrão que inicialmente era 3 pelo novo valor 100.

Operações Matemáticas com Tensores

```
1  tensor = tf.constant([[1, 2], [3, 4]])
2  tensor
```

```
<tf.Tensor: shape=(2, 2), dtype=int32, numpy=
array([[1, 2],
       [3, 4]], dtype=int32)>
```

Como já comentado anteriormente, basicamente os tipos de dados que teremos em grande parte de nossas aplicações serão dados numéricos em formato escalar, vetorial ou matricial (porém existirão aplicações com outros tipos de dados que veremos nos capítulos subsequentes), e para estes dados numéricos logicamente podemos fazer uso de certas operações matemáticas.

Para toda e qualquer operação aritmética usaremos a simbologia da sintaxe Python para realizar as devidas operações.

Adição Entre um Escalar e um Tensor

```
1  print(tensor + 2)
2
```

```
<tf.Tensor: shape=(2, 2), dtype=int32, numpy=
array([[3, 4],
       [5, 6]], dtype=int32)>
```

Para realizar uma simples adição entre os dados de um tensor e um determinado número, podemos realizar esta operação diretamente como parâmetro de nossa função print( ).

Neste caso, como estamos realizando a soma entre um escalar e os elementos de uma matriz interna a um tensor, o que ocorrerá é que o valor a ser somado fará a adição para cada elemento que compõe a array.

Usando da mesma variável do exemplo anterior, aqui temos o valor 2 somado ao elemento de índice [0,0], também ao de índice

[0,1], assim como ao de índice [1,0] e por fim ao de índice [1,1]. Em matrizes de maior número de linhas e colunas o processo se repetirá para todas elas.

O retorno, como esperado, é uma array de valores [3,4] para a linha 1 e [5,6] para a linha 2.

```
1  tensor_soma = tensor + 2
2
3  print(tensor_soma)
4
```

Lembrando novamente que, por meio da função print( ) temos um retorno que imediatamente após exibido será descartado da memória.

Para que tenhamos um resultado permanente da operação matemática executada, devemos atribuir seu resultado a uma nova variável ou atualizar uma já existente.

Subtração Entre um Escalar e um Tensor

```
1  print(tensor - 1)
2
```

```
tf.Tensor(
[[0 1]
 [2 3]], shape=(2, 2), dtype=int32)
```

Da mesma forma, é perfeitamente possível realizar a subtração de um determinado valor pelos elementos de nosso tensor, usando da simbologia adequada para esse caso.

Multiplicação Entre um Escalar e um Tensor

```
1  tensor * 5
```

```
<tf.Tensor: shape=(2, 2), dtype=int32, numpy=
array([[ 5, 10],
       [15, 20]], dtype=int32)>
```

Outra operação muito comumente utilizada é a de multiplicação entre um determinado valor escalar e os elementos de nosso tensor.

Divisão Entre um Escalar e um Tensor

```
1  tensor / 2
```

```
<tf.Tensor: shape=(2, 2), dtype=float64, numpy=
array([[0.5, 1. ],
       [1.5, 2. ]])>
```

Por fim, das operações aritméticas mais básicas podemos realizar a divisão entre os valores de cada elemento de nosso tensor por um valor escalar.

Interação Entre Tensores

```
1  tensor + tensor
```

```
<tf.Tensor: shape=(2, 2), dtype=int32, numpy=
array([[2, 4],
       [6, 8]], dtype=int32)>
```

Uma vez que se tratando de tensores temos dados de mesmo tipo, podemos perfeitamente realizar operações entre os conteúdos de nossos tensores diretamente gerando a expressão matemática instanciando suas variáveis.

```
1  tensor * tensor
```

```
<tf.Tensor: shape=(2, 2), dtype=int32, numpy=
array([[ 1,  4],
       [ 9, 16]], dtype=int32)>
```

Nos mesmos moldes podemos aplicar toda e qualquer expressão matemática entre tensores.

```python
tensor1 = tf.constant([[1, 2], [3, 4]])
tensor2 = tf.constant([[1, 2], [7, 8]])
tensor3 = tf.constant([[1, 2], [5, 6]])

print(tensor1 + (tensor2 * tensor3))

```

```
tf.Tensor(
[[ 2  6]
 [38 52]], shape=(2, 2), dtype=int32)
```

Apenas encerrando este tópico, a compatibilidade de operações entre tensores de estende também para operações matemáticas compostas, desde que se respeite a sintaxe correta ao escrever a equação.

Operações Lógicas com Tensores

Da mesma forma como temos liberdade para usar de toda e qualquer expressão matemática sobre os dados/valores de nossos tensores, também é possível para certos casos usar de operações lógicas entre os mesmos, de modo que tais operações, assim como as matemáticas, serão realizadas elemento a elemento de cada tensor.

```
tensor1 = tf.constant([[1, 2], [3, 4]])
tensor2 = tf.constant([[5, 6], [7, 8]])

```

Criando agora dois novos tensores distintos, apenas para exemplo, com dados diferentes entre si, podemos perfeitamente usar de proposições lógicas para obter este tipo de retorno de cada elemento dos tensores.

```
tensor1 >= tensor2

```

```
<tf.Tensor: shape=(2, 2), dtype=bool, numpy=
array([[False, False],
       [False, False]])>
```

Em nosso exemplo, escrevendo a proposição lógica tensor1 >= tensor2 (o conteúdo de tensor1 é maior ou igual que o conteúdo de tensor2?) temos como retorno, mantendo o formato de array, False para cada proposição falsa.

```
tensor1 = tf.constant([[1, 2], [3, 4]])
tensor2 = tf.constant([[1, 2], [7, 8]])

tensor1 >= tensor2

```

```
<tf.Tensor: shape=(2, 2), dtype=bool, numpy=
array([[ True,  True],
       [False, False]])>
```

Em outro contexto, agora onde os respectivos elementos de cada posição de índice eram iguais ou maiores quando comparados os do tensor1 com de tensor2, obtivemos para os elementos da primeira linha True, e para os elementos da segunda linha False.

```
1  tensor1 = tf.constant([[1, 2], [3, 4]])
2  tensor2 = tf.constant([[1, 2], [7, 8]])
3  tensor3 = tf.constant([[1, 2], [5, 6]])
4
5  print(tensor1 >= tensor2 or tensor3 <= tensor1)
6
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-41-dc421f03610b> in <module>()
      3 tensor3 = tf.constant([[1, 2], [5, 6]])
      4
----> 5 print(tensor1 >= tensor2 or tensor3 <= tensor1)

/usr/local/lib/python3.6/dist-packages/tensorflow_core/python/framework/ops.py
    863
    864   def __bool__(self):
--> 865     return bool(self._numpy())
    866
    867   __nonzero__ = __bool__

ValueError: The truth value of an array with more than one element is ambiguous.
```

Apenas como exemplo, simulando algo que não é permitido, ao tentar realizar uma operação lógica composta, onde duas proposições são colocadas em teste, teremos um erro pois o retorno gerado para ambas proposições teria um formato diferente do formato original dos tensores. *O mesmo ocorre ao tentar usar de estruturas condicionais sobre estes tipos de dados.

Lembre-se que sempre que estamos cruzando dados de matrizes, devemos respeitar o formato das mesmas. Neste caso, as matrizes 2X2 que estamos usando, teriam como retorno uma matriz 1X1, pois o interpretador faria a comparação literal entre as mesmas retornando True ou False para cada proposição lógica.

Funções da Biblioteca Numpy em Tensores

Uma vez que a estrutura de dados de nossos tensores enquanto constantes ou variáveis é praticamente idêntica a de uma array numpy, outra possibilidade que temos é a de importar a biblioteca numpy e usar de suas funções diretamente sobre estes dados.

Essa prática será muito usada posteriormente pois em toda rede neural artificial, antes de codificar a arquitetura da mesma propriamente dita, existe sempre uma fase de tratamento dos dados para que eles possam ser processados pelos métodos da rede neural.

```
tensor = tf.constant([[1, 2], [3, 4]])

np.square(tensor)

array([[ 1,  4],
       [ 9, 16]], dtype=int32)
```

Por exemplo, usando do mesmo tensor dos exemplos anteriores, usando do método np.square( ) podemos elevar um determinado elemento ao quadrado, neste caso, cada elemento em sua devida posição da matriz será elevado ao quadrado.

```
np.sqrt(tensor)

array([[1.        , 1.41421356],
       [1.73205081, 2.        ]])
```

Do mesmo modo, por meio do método np.sqrt( ) podemos obter a raiz quadrada de um determinado elemento, aqui novamente se aplicando a cada elemento da matriz.

```
tensor1 = tf.constant([[1, 2], [3, 4]])
tensor2 = tf.constant([[5, 6], [7, 8]])

np.dot(tensor1, tensor2)

```

```
array([[19, 22],
       [43, 50]], dtype=int32)
```

Supondo dois tensores 2X2 onde abstraímos suas dimensões como camadas, podemos realizar o cálculo do produto escalar entre os elementos que coincidem e se equivalem entre camadas.

Lembrando apenas que aqui neste caso em particular não estamos multiplicando uma matriz por um número escalar, mas sim usando outra matriz inteira como incógnita a ser usada para multiplicação e produto.

Apenas finalizando esta linha de raciocínio, a biblioteca Numpy oferece uma vasta gama de funções aplicáveis a dados em formato matricial.

Raciocine que todos métodos os quais se apliquem individualmente a cada elemento em sua respectiva posição na matriz pode ser perfeitamente aplicado a um tensor.

Como dito anteriormente, desde que não se altere o formato de linhas e colunas de uma matriz, podemos usar de toda e qualquer operação lógica e/ou aritmética sobre estes dados.

Strings Como Tensores

Uma possível aplicação que pode, embora não muito comum, ocorrer em nossos modelos é quando temos dados de entrada em forma de strings.

Pela lógica quando estamos trabalhando com dados em formato escalar, vetorial ou matricial estamos majoritariamente trabalhando com elementos numéricos.

Contornando essa situação, lembre-se que quando trabalhamos com strings em Python, cada caractere de uma string pode ser desmembrado como um elemento independente, assim como todo e qualquer elemento de uma string é internamente mapeado e indexado por nosso interpretador.

Dessa forma, ao se trabalhar com dados que inicialmente estão em formato string para que os mesmos sejam tratados como numéricos para uma variável ou constante em um tensor, é necessário realizar uma pequena decodificação.

```
1  tf_string = tf.constant("Fernando")
2
3  print(tf_string)
4
```

```
tf.Tensor(b'Fernando', shape=(), dtype=string)
```

Apenas para fins de exemplo, inicialmente criamos uma variável de nome tf_string que recebe como atributo uma constante por meio da função tf.constant( ) onde parametrizamos a mesma com a string “Fernando”, respeitando é claro, a sintaxe Python.

Por meio da função print( ) podemos exibir o conteúdo de nossa variável, em outras palavras, o conteúdo de nosso tensor, e podemos perceber que nossa string internamente é tratada como uma b’string.

```
tf.strings.length(tf_string)

<tf.Tensor: shape=(), dtype=int32, numpy=8>
```

Usando do método TensorFlow tf.strings.length( ) parametrizando o mesmo com o conteúdo de nossa tf_string, podemos ver que o tamanho desta string é, como esperado, 8 caracteres.

```
tf.strings.unicode_decode(tf_string, "UTF8")

<tf.Tensor: shape=(8,), dtype=int32,
 numpy=array([ 70, 101, 114, 110,  97, 110, 100, 111], dtype=int32)>
```

Finalmente, usando do método rf.strings.unicode_decode( ) parametrizado com a variável que será a fonte e qual padrão de formato converter, obtemos como resultado uma nova array com valores equivalentes aos caracteres em UTF8 mapeados, indexados e atribuídos para cada elemento que anteriormente compunha a string “Fernando”.

Existem outros métodos, inclusive mais práticos, onde estaremos realizando estes e outros tipos de conversões, porém veremos em exemplos dos capítulos seguintes.

```
string2 = tf.constant(["Fernando", "Python", "TensorFlow"])

for i in string2:
  print(i)

tf.Tensor(b'Fernando', shape=(), dtype=string)
tf.Tensor(b'Python', shape=(), dtype=string)
tf.Tensor(b'TensorFlow', shape=(), dtype=string)
```

Apenas complementando o exemplo anterior, lembrando que quando temos dados, principalmente oriundos de uma lista onde mais de um elemento da mesma está em formato string, ao iterar

sobre tais elementos os desmembraremos elemento por elemento (string por string / palavra por palavra) e não caractere por caractere.

Percorrendo o tensor atribuído a string2 por meio de um laço for, pedindo que a cada laço exiba em tela o elemento lido, o resultado que temos são 3 novos tensores compostos por b’strings.

Trabalhando com Sessões

Buscando entender agora de forma prática como se faz o uso de uma sessão, apenas vamos retomar rapidamente o referencial teórico deste tipo de estrutura.

Na biblioteca TensorFlow, em sua proposta de arquitetura tanto para redes neurais artificiais quanto para propósito geral, os dados em si são codificados em blocos modulares instanciados apenas quando necessário, permanecendo em standby quando fora de uso.

O processamento destes dados se dá por fluxos de informação organizados assim como um fluxograma em um grafo, de forma que os blocos são rearranjados e organizados conforme a demanda por uma determinada ação, alimentados com dados de entrada, funções de ativação e estruturas de saída de acordo com o contexto e propósito.

Pois bem, dessa forma, raciocine que tudo visto até então nos exemplos anteriores, em uma aplicação real nada mais seria do que simples execuções de código descartados em seguida da memória.

Para o efetivo fluxo de processamento de dados pelo núcleo Python, usando do interpretador base e do TensorFlow, nas versões antigas da biblioteca era necessário criar algumas estruturas de código adicionais para que o modelo fosse de fato criado e alocado em memória em forma de sessão.

A partir das versões 2.X do TensorFlow este processo passou a ser implícito, não havendo a necessidade de criar manualmente uma sessão, o que facilitou muito a elaboração do modelo podendo dar mais ênfase nos dados, seus fluxos e interação.

Apenas para entender como é realizado o processo de uma sessão, vamos emular algumas funcionalidades da versão 1.X do

TensorFlow e criar manualmente toda estrutura de fluxo de informação.

```
1  import tensorflow.compat.v1 as tf
2  tf.disable_v2_behavior()
3
```

Inicialmente importamos o módulo compat.v1 da biblioteca tensorflow e a referenciamos como tf. Em seguida executamos a função tf.disable_v2_behavior( ) que como o próprio nome já sugere, desabilita a forma como o interpretador iria se comportar (usando do núcleo 2.X do TensorFlow) para que se possa ser criado nosso exemplo.

```
1  num1 = tf.constant(8)
2  num2 = tf.constant(15)
3
4  soma_num = num1 + num2
5
```

Na sequência criamos por meio do método tf.constant( ) duas constantes quaisquer com números do tipo int atribuídos as variáveis num1 e num2 respectivamente.

Em seguida criamos uma nova variável de nome soma_num que simplesmente realiza a soma dos conteúdos de num1 e num2.

```
1  print(soma_num)
2
3  print(type(soma_num))
4
```

```
Tensor("add:0", shape=(), dtype=int32)
<class 'tensorflow.python.framework.ops.Tensor'>
```

Por meio da função print( ) note que ao tentar exibir em tela o resultado da operação realizada em soma_num é nos retornando um tensor, retornando “add:0”.

Isso se dá pois até o momento, ao criar estas estruturas para nossas variáveis, internamente o interpretador do TensorFlow 1.X

reconheceria estes tipos de dados apenas como tensores para um fluxo de operação, mas até então estáticos.

```
1  with tf.Session() as sess:
2      soma = sess.run(soma_num)
3
4  print(soma)
5
```

```
23
```

Para criar a sessão, aos moldes do TensorFlow 1.X, era necessário criar a estrutura da imagem acima, com with ts.Session( ) tendo indentado para si um bloco de código onde soma_num é instanciada e executada ao ser chamada a função sess.run( ), para que finalmente o fluxo de dados da operação matemática dessa variável fosse posta em operação.

Repare que a estrutura em si lembra um pouco a de uma estrutura condicional, porém, neste caso essa estrutura é criada para que tais dados e seu fluxo de informação sejam executados dentro dessa instância.

Na sequência, já fora dessa estrutura, novamente no escopo global do código, ao tentar exibir em tela o conteúdo de soma por meio da função print( ) finalmente teríamos o devido resultado da soma.

Por sorte, para as versões a partir da 2.X do TensorFlow não é mais necessário criar este tipo de estrutura para cada operação. Não que estas estruturas internamente não existam, mas para as versões atuais da biblioteca foram totalmente remodeladas de modo a ficarem implícitas ao usuário, bastando ter em mente que sempre que um fluxo de dados for ativado, internamente o mesmo está ocorrendo em um bloco modular de sessão.

REDES NEURAIS ARTIFICIAIS

Para darmos início ao nosso primeiro tópico dedicado à prática de aplicação do TensorFlow, vamos tecer alguns breves comentários que se fazem necessários nesse momento.

Primeiro deles é que é preciso entender que a biblioteca TensorFlow é uma biblioteca bastante robusta, tanto para interação com outras bibliotecas quanto para bibliotecas que ela incorpora para si.

De modo reduzido podemos dizer que ao longo dos anos de desenvolvimento do TensorFlow foram sendo incluídas para a mesma uma série de outras bibliotecas, módulos e pacotes anteriormente independentes, permitindo ter uma série de recursos à disposição para uso conforme o contexto.

Como nosso propósito neste pequeno livro é entender gradualmente os conceitos e aplicações de uma rede neural artificial via TensorFlow, vamos começar do básico, aumentando de complexidade conforme avançamos em nossos exemplos.

Classificação Binária

Uma das aplicações mais comuns de uma rede neural artificial é criar e treinar a mesma para que, por processos de aprendizado de máquina, realize ações como classificação a partir de dados.

Nesse contexto, a grande maioria das vezes usamos de modelos de aprendizado supervisionado para ensinar a nossa rede neural os padrões corretos a serem levados em conta quando a mesma estiver processando dados.

Raciocine que a arquitetura de uma rede neural artificial criada para fins de classificação, por si só, embora estruturalmente seja bastante robusta, sozinha não realiza nenhuma ação pois somente o modelo não possui capacidade de abstração.

Logo, temos que realizar um breve tratamento de nossos dados, em seguida treinando a rede com os padrões certos a serem identificados para que a mesma a partir destes consiga processar e classificar dados sozinha.

O processo chamado de supervisionado nada mais é do que uma das etapas iniciais de toda codificação desse tipo de rede neural artificial onde literalmente mostramos para a rede neural a base de dados com todas suas amostras e características, em seguida mostrando das mesmas amostras quais padrões deverão ser identificados, mapeados e posteriormente usados como parâmetro para alguma função de ativação, que no caso de uma rede neural artificial para classificação pode ser classificar objetos quanto a seu tipo, formato, tamanho, preço, etc... quanto as suas características que o diferem de outros objetos.

Para nosso exemplo inicial usaremos de um modelo de rede neural artificial elaborado para realizar uma classificação binária, onde partindo de uma base de dados com dados de exames de mama treinaremos a rede de forma que uma vez treinada, a mesma possa ser utilizada para interpretar novos exames, julgando a partir

de seu aprendizado, destes exames quais revelam lesões (câncer de mama) ou apresentam estruturas sadias.

Este mesmo tipo de aplicação, onde o resultado é “uma coisa ou outra”, pode ser perfeitamente adaptado para classificação e diferenciação de outros objetos de estudo (classificar a partir de imagens cães e gatos, por exemplo) assim como existem modelos de classificação mais complexos chamados de classificadores multiclasse onde um determinado grupo de objetos de estudo pode ser identificado e separado quanto sua classificação em mais de 2 tipos.

Por hora, vamos nos ater ao básico para entender de fato como esses conceitos são implementados em uma aplicação real.

```
1  import tensorflow as tf
2  import pandas as pd
3  import numpy as np
4  import sklearn as sk
5  import matplotlib.pyplot as plt
6
```

Inicialmente realizaremos as importações das bibliotecas, módulos e pacotes que serão utilizados ao longo de nosso código.

Nesse caso, realizamos a importação da biblioteca tensorflow da qual usaremos seu núcleo e sub-bibliotecas para criação do modelo de nossa rede neural artificial, o referenciando como tf, da mesma forma importamos as bibliotecas pandas e numpy que nos possibilitarão importar e trabalhar com nossos dados matriciais realizando o tratamento dos mesmos.

Também importamos a biblioteca sklearn que oferecerá algumas ferramentas para tratamento dos dados.

Por fim importamos a biblioteca matplotlib que nos permitirá gerar gráficos para de forma didática e visual apresentar certos resultados.

```
1   import tensorflow as tf
2   from tensorflow import keras
3   from tensorflow.keras import Sequential
4   from tensorflow.keras.layers import Flatten, Dense, Dropout
5   from tensorflow.keras.layers import Conv1D, MaxPool1D
6   import pandas as pd
7   import numpy as np
8   import matplotlib.pyplot as plt
9   from tensorflow.keras.optimizers import Adam
10  from sklearn import datasets, metrics
11  from sklearn.model_selection import train_test_split
12  from sklearn.preprocessing import StandardScaler
13
```

Lembrando que no parágrafo anterior, realizamos as importações das bibliotecas com todo seu conteúdo, o que não é uma boa prática de programação assim como em questão de performance teríamos muita coisa carregada em memória desnecessariamente.

Aprimorando o processo de importação, podemos de fato importar somente o que iremos fazer uso das respectivas bibliotecas, aumentando algumas linhas de código no processo de declarar tais comandos, porém ganhando muito em performance ao executar nosso modelo.

Veja que das bibliotecas anteriormente citadas agora importamos módulos e pacotes bastante específicos. Aproveitando a deixa, podemos já introduzir o que tais módulos e pacotes oferecerão de funcionalidades implementadas para nosso modelo.

Da biblioteca tensorflow importamos keras, sub-biblioteca esta que em outros momentos era independente mas foi incorporada ao tensorflow para oferecer modelos de arquitetura de redes neurais tanto simples quanto complexas (deep learning) de forma bastante intuitiva e eficiente.

Da própria biblioteca keras importamos Sequential, modelo pré-configurado para criação das camadas da rede neural artificial assim como suas interconexões permitindo processos como feedforward e

retropropagation, onde os fluxos de dados são processados de forma sequencial e progressiva da camada de entrada em sentido a camada de saída, mas permitindo que a cada execução da rede, o resultado final possa retroalimentar a mesma para que seja possível o mecanismo de aprendizado de máquina.

Do módulo layers da biblioteca keras importamos Flatten, Dense, Dropout, Conv1D e MaxPool1D. Vamos procurar entender o que cada estrutura dessa realiza. Inicialmente Flatten usa de uma série de métricas para pegar uma determinada matriz, independentemente de seu formato, e a converter para uma matriz unidimensional onde cada elemento é usado como um neurônio da camada de entrada da rede neural.

Dense nos permite a criação de quantas camadas forem necessárias assim como quantos neurônios por camada e suas interconexões forem necessárias no projeto.

Dropout aplica algumas métricas onde aleatoriamente alguns neurônios das camadas são arbitrariamente excluídos (ou tem seus valores alterados para zero absoluto, não sendo assim atualizados durante o treinamento da rede neural) e é testado assim a consistência da rede sem os mesmos, muito utilizado para detectar vícios de processamento da rede (também chamado de overfitting) que possam estar afetando negativamente os resultados.

Conv1D nos permitirá gerar e manipular camadas de uma rede convolucional, ou seja, camadas com dados oriundos de imagens ou que em outro momento foram convertidos e mapeados a partir de dados de imagens.

MaxPool1D, complementar ao Conv1D, nos permite aplicar alguns métodos de polimento dos dados convolutivos, de modo a padronizar os mesmos de acordo com o padrão de todos os outros dados da rede.

Em seguida do módulo optimizers da biblioteca keras importamos Adam. Pela própria documentação da biblioteca keras é possível ver que a mesma possui um grande número de otimizadores, sendo cada um deles de aplicabilidade específica,

aqui, para nosso exemplo de um classificador binário, o otimizador Adam tende a oferecer uma margem de precisão muito alta ao transformar dados de saída da rede neural em gatilhos de uma função de ativação. *Ao longo dos próximos exemplos, de acordo com seus propósitos usaremos de outros otimizadores.

Na sequência da biblioteca sklearn importamos datasets e metrics. Datasets são justamente bases de dados de amostra para livre uso no processo de aprendizado, metrics nos oferece algumas funções para que possamos avaliar a performance do modelo.

Do módulo model_selection da biblioteca sklearn importamos train_test_split, ferramenta fundamental para uma rede neural artificial onde, após ter feito o tratamento inicial de nossa base de dados, por meio de train_test_split a separaremos em dados distintos para treino e para teste de nossa rede neural artificial, prática muito comum usada em conjunto ao treinamento supervisionado da rede.

Por fim, do módulo preprocessing da biblioteca sklearn importamos StardardScaler, ferramenta de normalização dos dados para padrões na prática mais eficientes.

```
1  print(tf.__version__)
2
```

```
2.4.0
```

Por meio da nossa já conhecida função print( ), parametrizada com tf.__version__ podemos verificar rapidamente qual a versão da biblioteca tensorflow em uso para nosso exemplo.

Neste caso, no momento em que escrevo este livro, estou fazendo o uso da versão 2.4.0 da biblioteca TensorFlow.

```
1  data = datasets.load_breast_cancer()
2
```

Na sequência criamos uma variável de nome data que chama a função datasets.load_breast_cancer( ), dessa forma realizamos o

carregamento de nossa base de dados, atribuindo seu conteúdo a variável data.

```
1   print(data.DESCR)
2
```

Parametrizando nossa função print( ) com data.DESCR obtemos como retorno um texto resumo com informações importantes desta base de dados a qual iremos explorar.

```
.. _breast_cancer_dataset:

Breast cancer wisconsin (diagnostic) dataset
--------------------------------------------

**Data Set Characteristics:**

    :Number of Instances: 569

    :Number of Attributes: 30 numeric, predictive attributes and the class

    :Attribute Information:
        - radius (mean of distances from center to points on the perimeter)
        - texture (standard deviation of gray-scale values)
        - perimeter
        - area
        - smoothness (local variation in radius lengths)
        - compactness (perimeter^2 / area - 1.0)
        - concavity (severity of concave portions of the contour)
        - concave points (number of concave portions of the contour)
        - symmetry
        - fractal dimension ("coastline approximation" - 1)
```

Realizando uma breve leitura deste texto informativo podemos ver que seu conteúdo trata-se de um compendio contendo informações de mapeamento de 569 exames de mama, obtendo a partir de cada amostra cerca de 30 características, apresentadas como, por exemplo, o raio, a textura, o perímetro, a área, regiões de borramento, regiões de consolidações, regiões de cavidades e concavidades, dados de simetria, dimensões fractais, entre outras características para esse estudo relevantes.

```
The mean, standard error, and "worst" or largest (mean of the three
largest values) of these features were computed for each image,
resulting in 30 features.  For instance, field 3 is Mean Radius, field
13 is Radius SE, field 23 is Worst Radius.

- class:
        - WDBC-Malignant
        - WDBC-Benign
```

Também é nos fornecido a informação de que para as amostras mapeadas existem algumas margens de desproporcionalidades que podem afetar negativamente os resultados dos processamentos uma vez que se tratando de mamas cada uma, para cada amostra tem seu respectivo tamanho, formato e volume, assim como todas características descritas anteriormente internas.

Outro marcador importante é que, para esta base de dados, o propósito geral é classificar tais amostras como lesões benignas ou malignas de acordo com suas características.

```
:Summary Statistics:

===================================== ====== ======
                                       Min    Max
===================================== ====== ======
radius (mean):                        6.981  28.11
texture (mean):                       9.71   39.28
perimeter (mean):                     43.79  188.5
area (mean):                          143.5  2501.0
smoothness (mean):                    0.053  0.163
compactness (mean):                   0.019  0.345
concavity (mean):                     0.0    0.427
concave points (mean):                0.0    0.201
symmetry (mean):                      0.106  0.304
fractal dimension (mean):             0.05   0.097
radius (standard error):              0.112  2.873
texture (standard error):             0.36   4.885
perimeter (standard error):           0.757  21.98
area (standard error):                6.802  542.2
smoothness (standard error):          0.002  0.031
compactness (standard error):         0.002  0.135
concavity (standard error):           0.0    0.396
concave points (standard error):      0.0    0.053
symmetry (standard error):            0.008  0.079
fractal dimension (standard error):   0.001  0.03
radius (worst):                       7.93   36.04
```

Em seguida é mostrado um extenso sumário das características quanto os intervalos considerados para as mesmas dentro das margens médias de erro permitidas.

```
:Missing Attribute Values: None

:Class Distribution: 212 - Malignant, 357 - Benign

:Creator:  Dr. William H. Wolberg, W. Nick Street, Olvi L. Mangasarian

:Donor: Nick Street

:Date: November, 1995
```

Na sequência temos mais algumas informações como em que proporção as amostras estão divididas.

Também são dados os devidos créditos aos autores e colaboradores da criação dessa base de dados.

```
This is a copy of UCI ML Breast Cancer Wisconsin (Diagnostic) datasets.
https://goo.gl/U2Uwz2

Features are computed from a digitized image of a fine needle
aspirate (FNA) of a breast mass.  They describe
characteristics of the cell nuclei present in the image.

Separating plane described above was obtained using
Multisurface Method-Tree (MSM-T) [K. P. Bennett, "Decision Tree
Construction Via Linear Programming." Proceedings of the 4th
Midwest Artificial Intelligence and Cognitive Science Society,
pp. 97-101, 1992], a classification method which uses linear
programming to construct a decision tree.  Relevant features
were selected using an exhaustive search in the space of 1-4
features and 1-3 separating planes.
```

Encerrando essa parte, são apresentadas algumas últimas informações relacionadas a publicação científica desta base de dados.

```
X = pd.DataFrame(data = data.data,
                 columns=data.feature_names)

print(X.head())

```

```
   mean radius  mean texture  ...  worst symmetry  worst fractal dimension
0        17.99         10.38  ...          0.4601                  0.11890
1        20.57         17.77  ...          0.2750                  0.08902
2        19.69         21.25  ...          0.3613                  0.08758
3        11.42         20.38  ...          0.6638                  0.17300
4        20.29         14.34  ...          0.2364                  0.07678

[5 rows x 30 columns]
```

Dando continuidade, começamos a partir deste momento explorar de fato o conteúdo de nossa base de dados.

Para isso, inicialmente criamos uma variável de nome X que por meio da função pd.DataFrame( ) realiza a leitura do conteúdo de nossa variável data, extraindo da mesma as informações de

data.data assim como de data.feature_names, gerando a partir destes dados um DataFrame, estrutura de dado bastante comum para este tipo de contexto por ser de certa forma parecida com uma estrutura matricial e ao mesmo tempo oferecer grande compatibilidade de integração com outras bibliotecas.

Por meio de nossa função print( ) parametrizada com X.head( ), obtemos como retorno as 5 primeiras linhas deste dataframe para que possamos visualizar a distribuição destes tipos de dados assim como sua indexação.

Visualizando as 5 primeiras linhas, temos uma tabela com 5 linhas e 30 colunas respectivamente de acordo com as características mapeadas e apresentadas anteriormente.

```
1  y = data.target
2
3  print(y)
4
```

```
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 1 0 0 0 0 0 0 0 0 1 0 1 1 1 1 1 0 0 1 0 0 1 1 1 1 0 1 0 0 1 1 1 1 0 1 0 0
 1 0 1 0 0 1 1 1 0 0 1 0 0 0 1 1 1 0 1 1 0 0 1 1 1 0 0 1 1 1 1 0 1 1 0 1 1
 1 1 1 1 1 1 0 0 0 1 0 0 1 1 1 0 0 1 0 1 0 0 1 0 0 1 1 0 1 1 0 1 1 1 1 0 1
 1 1 1 1 1 1 1 1 0 1 1 1 1 0 0 1 0 1 1 0 0 1 1 0 0 1 1 1 1 0 1 1 0 0 0 1 0
 1 0 1 1 1 0 1 1 0 0 1 0 0 0 0 1 0 0 0 1 0 1 0 1 1 0 1 0 0 0 0 1 1 0 0 1 1
 1 0 1 1 1 1 1 0 0 1 1 0 1 1 0 0 1 0 1 1 1 1 0 1 1 1 1 1 0 1 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 1 1 1 1 1 1 0 1 0 1 1 0 1 1 0 1 0 0 1 1 1 1 1 1 1 1 1 1 1 1
 1 0 1 1 0 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 0 1 0 1 1 1 1 0 0 0 1 1
 1 1 0 1 0 1 0 1 1 1 0 1 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 1 1 1 1 1 0 0 1 0 0
 0 1 0 0 1 1 1 1 1 0 1 1 1 1 1 0 1 1 1 0 1 1 0 0 1 1 1 1 1 1 0 1 1 1 1 1 1
 1 0 1 1 1 1 1 0 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 0 1 0 1 1 1 1 1 0 1 1
 0 1 0 1 1 0 1 0 1 1 1 1 1 1 1 1 0 0 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 0 1
 1 1 1 1 1 1 0 1 0 1 1 0 1 1 1 1 1 0 0 1 0 1 0 1 1 1 1 1 0 1 1 0 1 0 1 0 0
 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 0 1 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 0 0 0 0 0 0 1]
```

De forma parecida criamos uma variável de nome y que por sua vez recebe atribuída para si própria os dados de data.target.

Em outras palavras, como dito anteriormente, neste modelo de rede neural artificial já temos os dados de entrada baseado nas amostras assim como as possíveis saídas (lesão benigna ou lesão

maligna), para que a rede neural encontre os padrões corretos que liguem cada amostra ao seu respectivo resultado, aprendendo com esse processo.

O retorno como pode ser visto é uma grande lista com elementos 0 e 1, sendo estes definidos para 0 quando a amostra é de uma lesão benigna e 1 quando a amostra apresenta uma lesão maligna.

```
print(data.target_names)

['malignant' 'benign']
```

O que pode ser confirmado se parametrizarmos nossa função print( ) com data.target_names. O retorno obtido é em formato de lista, apresentando dois elementos ‘malignant’ (maligno) ou ‘benign’ (benigno).

```
print(X.shape)

(569, 30)
```

Verificando o formado de X, via função print( ) podemos ver que aqui temos uma base com 569 linhas e 30 colunas.

```
X_treino, X_teste, y_treino, y_teste = train_test_split(X,
                                                        y,
                                                        test_size = 0.2,
                                                        random_state = 0,
                                                        stratify = y)
```

Uma vez encerradas as etapas de tratamento de nossos dados base, próximo passo é dividir nossa base de dados em grupos que servirão para treinar e testar nossa rede neural artificial.

Apenas lembrando novamente que para este modelo estamos usando de um modelo de aprendizado de máquina supervisionado, logo, estes grupos seriam como se tivéssemos em mãos as perguntas e as respostas, bastando apenas treinar a rede

mostrando o caminho para que ela associe as devidas perguntas a suas respectivas respostas.

Para esta etapa inicialmente criamos 4 variáveis, X_treino, X_teste, y_treino e y_teste que chamam a função train_test_split( ) a parametrizando com algumas métricas.

```
def train_test_split(*arrays, **options)
```

Consultando a documentação inline desta função podemos ver que a mesma espera alguns argumentos cm *args e **kwargs.

Sendo assim, inicialmente a parametrizamos de modo que os dados de origem serão obtidos das variáveis X e y, respectivamente.

Na sequência definimos o parâmetro test_size, neste caso para 0.2, o que em outras palavras dignifica que da base original, 80% dos dados serão direcionados para treino enquanto 20% dos mesmos serão alocados para teste da rede neural.

Também definimos um segundo parâmetro chamado random_state para zero pois como estamos trabalhando com uma base de dados muito pequena, a aleatoriedade das escolhas das amostras que serão direcionadas para treino e para teste não surtirão diferença considerável por parte de eficiência na margem de precisão dos resultados.

Em outros contextos, usando de bases de dados maiores, é importante que tal parâmetro seja definido para que justamente divida e distribua as amostras aleatoriamente, evitando assim treinar uma rede neural viciada.

Por fim, definindo o parâmetro stratify com y estamos “forçando” a rede neural a dividir as amostras para treino e teste em uma proporção justa para que a mesma aprenda com maior eficiência.

Em modelos diferentes de uma classificação binária este parâmetro não se aplica, aqui, neste exemplo, queremos garantir que de fato dos 80% e 20% definidos como base para treino e para teste, em cada grupo tenhamos uma margem proporcional entre si de amostras com saídas 0 e 1.

```
print(X_treino.shape)

print(X_teste.shape)

```

```
(455, 30)
(114, 30)
```

Verificando os respectivos formatos de X_treino e X_teste podemos observar a proporção indicada anteriormente.

```
escalonador = StandardScaler()

X_treino = escalonador.fit_transform(X_treino)
X_teste = escalonador.transform(X_teste)

```

Continuando com nosso exemplo, uma prática bastante comum de ser feita para que a partir desta obtenhamos melhor performance por parte de processamento é realizar o escalonamento de alguns dados.

Lembre-se que nossos dados de saída estão em formato 0 ou 1, dentro dessa lógica, para ganho de performance podemos transformar nossos dados de entrada em uma escala entre 0 e 1.

Para isso simplesmente criamos uma variável de nome escalonador que por sua vez chama a função StandardScaler( ).

Em seguida é preciso atualizar os valores de nossas variáveis X_treino e X_teste, chamando nosso escalonador e usando as funções fit_transform( ) e transform( ), respectivamente.

```
X_treino = X_treino.reshape(455, 30, 1)
X_teste = X_teste.reshape(114, 30, 1)

```

Agora que temos todos dados escalonados dentro de um intervalo que vai de 0 a 1, precisamos também transformar o formato original destes dados para um novo formato onde os

mesmos possam ser usados como neurônios da camada de entrada de nossa rede neural artificial.

Isso é feito de forma bastante simples, bastando atualizar novamente tais variáveis, dessa vez por meio da função reshape( ) passando a referência do número de linhas, seguido do número de colunas, seguido do número de dimensões, que neste caso, será uma dimensão.

```
1   modelo = Sequential()
2   modelo.add(Conv1D(filters=32,
3                     kernel_size=2,
4                     activation='relu',
5                     input_shape = (30, 1)))
6   modelo.add(BatchNormalization())
7   modelo.add(Dropout(0.2))
8
9   modelo.add(Conv1D(filters=64,
10                    kernel_size=2,
11                    activation='relu'))
12  modelo.add(BatchNormalization())
13  modelo.add(Dropout(0.5))
14
15  modelo.add(Flatten())
16  modelo.add(Dense(64,
17                   activation='relu'))
18  modelo.add(Dropout(0.5))
19
20  modelo.add(Dense(1,
21                   activation='sigmoid'))
22
```

Prosseguindo com nosso código, hora de criar a arquitetura da rede neural artificial. Nesta etapa que definimos o esqueleto da rede neural artificial de acordo com o propósito a ser empregado.

Inicialmente criamos uma variável de nome modelo que por sua vez instancia e executa a classe Sequential( ), inicializando a mesma.

Na sequência criamos a primeira camada de nossa rede neural artificial, ou como rotineiramente é chamada, aqui temos nossa camada de entrada.

O processo de criação é bastante simples, por meio da função add( ), inicialmente parametrizada com Conv1D( ) já que para este exemplo temos dados obtidos e mapeados a partir de imagens. Como parâmetros para Conv1D( ) inicialmente definimos filters=32, que representa o número de filtros a serem aplicados para cada amostra em seu processo interno de convolução.

Em seguida definimos kernel_size=2, o que em outras palavras significa que uma janela é definida com um número específico da matriz detectora de características e mapeamento. Raciocine que o processamento de dados obtidos a partir de uma imagem é algo bastante complexo, independentemente do tamanho da imagem e da quantidade de canais de cor, funções internas como kernel_size dividem a imagem em blocos menores mapeando as características daquele bloco específico e gerando uma indexação para tais dados.

Na sequência definimos para activation ‘relu’, sigla para *rectified linear units* em tradução livre, que nada mais é do que uma função de ativação onde ao final do processo de cruzamento dos dados de entrada para as camadas subsequentes será retornado um valor 0 ou 1.

*Para outras aplicações, onde usamos de um classificador multiclasse, é bastante comum usar de outras funções de ativação como a softmax, por exemplo, que retorna valores entre 0 e 1 (0 e todos intermediários até 1), gerando uma probabilidade de uma saída tender para zero ou para 1. Aqui em nosso exemplo inicial, como só temos duas possíveis saídas, 0 ou 1, usamos do ativador relu.

Por fim, para esse primeiro bloco, referente a camada de entrada de nossa rede neural, o último parâmetro a ser definido é o input_shape, que especifica o tamanho da camada de entrada, ou quantos neurônios compõe a camada de entrada. Nesse caso, 30 neurônios.

Terminada a criação da primeira camada, é aplicado sobre a mesma uma camada de normalização via BatchNormalization( ) assim como via Dropout( ) parametrizado em 0.2 definimos que desta camada para a próxima camada intermediária, 20% das amostras serão aleatoriamente removidas, forçando que a rede neural assim identifique os padrões contornando esses “erros”.

Isso é uma prática muito comum a ser feita para esse tipo de modelo pois assim conseguimos obter maior precisão no tratamento dos dados nos processos de interconexões e processamento entre camadas.

Na sequência criamos uma nova camada via função add( ) parametrizada com Conv1D( ) por sua vez parametrizada com duas diferenças em relação a camada de entrada. Primeira delas o número de filtros passou de 32 para 64, assim como não é mais necessário definir um input_size pois esta é uma camada intermediária e não de entrada.

Novamente é aplicado uma normalização via BatchNormalization( ) e uma nova remoção aleatória de dados via Dropout( ), agora eliminando 50% das amostras.

Em seguida é aplicada sobre essa camada, via função Flatten( ), um novo redimensionamento dos dados.

Dando sequência criamos mais uma camada para nossa rede neural via add( ) agora parametrizada com Dense( ) que por sua vez possuirá 64 neurônios em sua estrutura, gerando mais uma vez um dado em formato 0 ou 1.

Repare que diferentemente das camadas anteriores onde a estrutura era definida por Conv1D realizando convoluções, agora temos dados que podem ser usados como neurônios sem necessidade de novas conversões.

Desta camada também será eliminado 50% das amostras de forma aleatória.

Por fim criamos a camada de saída de nossa rede neural, esta com apenas um neurônio, uma vez que queremos obter como

resposta apenas um valor 0 ou 1. Também definimos para a mesma como método de ativação ‘sigmoid’, o que basicamente indica que após todos os processamentos anteriores, para saída serão feitos alguns cálculos internos sobre os 64 neurônios da camada densa, usando desta referência para definir que de acordo com os resultados a saída será 0 ou 1.

A função sigmóide basicamente realizará este tipo de interpretação, dado um grupo de amostras, conforma o número de amostras que retornaram 0 e 1, a mesma usará este dado como métrica para definir se após todo processamento da rede neural para aquele canal a maioria das observações indicam um resultado final o ou 1.

```
1   modelo.summary()
2
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv1d (Conv1D)              (None, 29, 32)            96
_________________________________________________________________
batch_normalization (BatchNo (None, 29, 32)            128
_________________________________________________________________
dropout (Dropout)            (None, 29, 32)            0
_________________________________________________________________
conv1d_1 (Conv1D)            (None, 28, 64)            4160
_________________________________________________________________
batch_normalization_1 (Batch (None, 28, 64)            256
_________________________________________________________________
dropout_1 (Dropout)          (None, 28, 64)            0
_________________________________________________________________
flatten (Flatten)            (None, 1792)              0
_________________________________________________________________
dense (Dense)                (None, 64)                114752
_________________________________________________________________
dropout_2 (Dropout)          (None, 64)                0
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 65
=================================================================
Total params: 119,457
Trainable params: 119,265
Non-trainable params: 192
```

Uma vez criada a estrutura da rede neural artificial, por meio da função summary( ) podemos ver um esboço de como a arquitetura da rede neural em si, camada por camada, está definida.

Note que internamente cada Layer corresponde a uma camada, todas elas com um determinado número de conexões. Interessante reparar que mesmo com uma base de dados pequena, alimentando uma rede neural com poucas camadas e poucos neurônios por camada, ainda assim existirão durante o processo 119,457 interconexões, cada uma delas cruzando dados e aplicando sobre os mesmos algumas métricas matemáticas, tudo isso para ligar

nosso ponto A ao ponto B, nossa pergunta a sua devida resposta, nosso dado a sua respectiva classificação.

```
1  modelo.compile(optimizer=Adam(lr=0.00005),
2                 loss ='binary_crossentropy',
3                 metrics=['accuracy'])
4
```

Próxima etapa, compilar o modelo, processo este onde definimos alguns parâmetros diretamente ligados ao processo de aprendizado de máquina realizado em nossa rede neural a medida que a mesma entra em ação e começa a processar dados.

Para nosso modelo simplesmente chamamos a função compile( ), parametrizando a mesma com optimizer = Adam( ) por sua vez parametrizado com lr=0,00005. Em outras palavras, aqui especificamos que a taxa de aprendizado é de 0,00005, ou seja, os valores das amostras serão atualizadas a cada 0,0005% das vezes em que os dados da rede neural como um todo forem processados.

Tenha em mente que os dados contidos nas interconexões entre neurônios e entre camadas de neurônios são totalmente dinâmicos, sendo corrigidos conforme definido para que aos poucos a rede neural aprenda com o processo.

Inicialmente a rede tentará fazer todas as ligações dos dados de entrada com seus respectivos dados de saída da rede neural, porém com margem de acertos muito baixa, então, são aplicadas uma série de métricas matemáticas sobre as mesmas e a medida que a margem de acertos vai aumentando, esses dados são guardados, atualizando as interconexões da rede execução após execução.

A tendência é que, após muitas execuções de nossa rede neural, fase após fase a mesma tenha dados com margem de acertos progressivamente maiores.

É altamente recomendável que para aplicações reais, estes tipos de parâmetro como o de taxa de aprendizado sejam testados com diferentes valores para que se encontre um ponto ideal entre tempo de processamento, performance e margem de acertos ao final dos processos.

Dando sequência, em nosso compilador também é definido loss com ‘binary_crossentropy’, em outras palavras, aqui definimos um parâmetro que internamente será usado pela rede neural para identificar seus erros (margem de erro) com base em uma entropia binária, nomenclatura essa para como o resultado final do processamento de dados em seu grupo gera probabilidade de um ou outro desfecho haja visto que estamos visando uma classificação binária.

Por fim, também especificamos que a métrica a ser levada em consideração para como a rede neural aprende deverá ser baseada em margem de precisão.

Para outros modelos existirão outras métricas, para esse, não buscamos dúvida, mas valores finais conclusivos sobre um tipo ou outro de lesão de nossa amostra.

```
1  history = modelo.fit(X_treino,
2                       y_treino,
3                       epochs=1000,
4                       validation_data=(X_teste, y_teste),
5                       verbose=1)
6
```

Finalizando nosso código, criamos uma variável de nome history que sobre nosso modelo aplica a função fit( ), basicamente a partir deste ponto nossa rede neural artificial entrará em execução.

Para função fit( ) repassamos alguns parâmetros, primeiro deles o conteúdo de X_treino assim como de y_treino, em epochs definimos o número de vezes que a rede neural deverá ser executada, para validation_data repassamos os dados de X_teste e y_teste, para que a rede tenha estes dados como parâmetro para identificar se está aprendendo com o processo chegando aos resultados certos, finalizando com verbose=1 apenas para que após o término da execução seja exibido em tela ou terminal o resultado do processamento.

```
Epoch 1/1000
15/15 [==============================] - 0s 24ms/step - loss: 1.1305 - accuracy: 0.4615
Epoch 2/1000
15/15 [==============================] - 0s 7ms/step - loss: 0.7612 - accuracy: 0.6264
Epoch 3/1000
15/15 [==============================] - 0s 6ms/step - loss: 0.5765 - accuracy: 0.7495
Epoch 4/1000
15/15 [==============================] - 0s 6ms/step - loss: 0.4033 - accuracy: 0.8022
Epoch 5/1000
15/15 [==============================] - 0s 6ms/step - loss: 0.4490 - accuracy: 0.7978
Epoch 6/1000
15/15 [==============================] - 0s 6ms/step - loss: 0.3927 - accuracy: 0.8374
Epoch 7/1000
15/15 [==============================] - 0s 6ms/step - loss: 0.2960 - accuracy: 0.8637
Epoch 8/1000
15/15 [==============================] - 0s 6ms/step - loss: 0.2890 - accuracy: 0.8703
Epoch 9/1000
15/15 [==============================] - 0s 7ms/step - loss: 0.2696 - accuracy: 0.8879
```

Como mencionado anteriormente, quando o interpretador lê o bloco de código onde a função fit( ) é chamada a rede neural começa de fato seu processamento.

Repare que nas fases iniciais a mesma apresenta um resultado pouco satisfatório, com margem de acertos inicial em 46%, subindo para 62%, posteriormente para 74%, seguindo com essa progressão justamente pelo processo de aprendizado de máquina, onde se toda a estrutura da rede neural está correta, a mesma aprenderá a cada fase de processamento.

```
Epoch 993/1000
15/15 [==============================] - 0s 7ms/step - loss: 0.0071 - accuracy: 0.9978
Epoch 994/1000
15/15 [==============================] - 0s 8ms/step - loss: 0.0094 - accuracy: 0.9956
Epoch 995/1000
15/15 [==============================] - 0s 7ms/step - loss: 0.0066 - accuracy: 1.0000
Epoch 996/1000
15/15 [==============================] - 0s 7ms/step - loss: 0.0069 - accuracy: 1.0000
Epoch 997/1000
15/15 [==============================] - 0s 7ms/step - loss: 0.0060 - accuracy: 1.0000
Epoch 998/1000
15/15 [==============================] - 0s 7ms/step - loss: 0.0103 - accuracy: 0.9956
Epoch 999/1000
15/15 [==============================] - 0s 7ms/step - loss: 0.0044 - accuracy: 1.0000
Epoch 1000/1000
15/15 [==============================] - 0s 8ms/step - loss: 0.0080 - accuracy: 0.9978
```

Após concluído o processamento para as 1000 épocas definidas anteriormente, é possível ver como retorno uma margem de acertos por volta de 99% para esse modelo.

Note que existem algumas épocas onde o retorno foi 100% mas isto obviamente é um falso positivo, levando em consideração uma margem de tolerância de 1% para mais ou para menos podemos dizer que a rede atingiu uma margem média de acertos de 99%.

```python
def CurvaAprendizado(historico, epoca):
    epoca = range(1, epoca + 1)
    plt.plot(epoca, historico.history['accuracy'])
    plt.plot(epoca, historico.history['val_accuracy'])
    plt.title('Precisão do Modelo')
    plt.ylabel('Margem de Acertos')
    plt.xlabel('Épocas')
    plt.legend(['Train', 'Val'], loc='upper left')
    plt.show()

    plt.plot(epoca, historico.history['loss'])
    plt.plot(epoca, historico.history['val_loss'])
    plt.title('Margem de Erro')
    plt.ylabel('Erros')
    plt.xlabel('Épocas')
    plt.legend(['Train', 'Val'], loc='upper left')
    plt.show()

```

Realizando uma codificação adicional por meio de nossa biblioteca matplotlib podemos transformar os resultados em gráficos para que possamos de um modo mais didático visualizar os resultados de processamento de nossa rede neural artificial.

Eis que temos o primeiro gráfico referente a precisão do modelo, condizendo perfeitamente com os resultados visualizados anteriormente via terminal, onde a rede começa seu processamento com uma margem de acertos baixa, aumentando consideravelmente estes valores a medida que aprende.

E como segundo gráfico temos a representação visual referente a margem de erro, que começa muito alta e diminui consideravelmente estabilizando com uma baixa margem de erro se levado em consideração todos falsos positivos que podem haver assim como a própria margem de acertos obtida que é extremamente satisfatória.

Classificação Multiclasse

No exemplo anterior pudemos ver um modelo bastante simples e ao mesmo tempo eficiente no que diz respeito a realização de uma classificação binária.

Fazendo uso do método de aprendizado supervisionado treinamos uma rede neural artificial para que identificasse os padrões corretos para ao final do processo ser capaz de classificar uma nova amostra para os critérios de lesão benigna ou maligna de um exame de mama.

Avançando um pouco mais, vamos buscar entender que para o mesmo tipo de rede neural artificial, podemos adaptá-la para realizar uma classificação multiclasse.

Diferentemente do exemplo anterior onde o resultado final do processamento da rede neural tendia a um desfecho ou outro, aqui trabalharemos com um modelo voltado para classificar amostras em 10 possíveis categorias.

Para isso, usaremos de outra base de dados de amostra disponível para estudos da própria biblioteca TensorFlow, chamada Fashion Mnist. A partir da mesma criaremos um modelo de rede neural capaz de detectar padrões que classificam imagens de roupas e acessórios de vestimenta onde o resultado final do processamento é um dado que mostra, de acordo com os 10 possíveis objetos, qual a maior probabilidade de nossa amostra ser de um deles especificamente.

Em outras palavras, feita a leitura de uma nova imagem de amostra a mesma será processada de acordo com as características aprendidas anteriormente pela rede neural para que ofereça uma probabilidade de ser um dos 10 tipos de itens, por exemplo, 94% de chance de ser uma camisa.

```
1  import numpy as np
2  import datetime
3  import tensorflow as tf
4  from tensorflow.keras.datasets import fashion_mnist
5
```

Partindo para a prática, todo processo sempre se inicia com as devidas importações das bibliotecas, módulos e pacotes que usaremos ao longo de nosso código.

Para esse exemplo em particular, do pacote datasets do módulo keras da biblioteca tensorflow importamos fashion_mnist.

```
1  (X_treino, y_treino), (X_teste, y_teste) = fashion_mnist.load_data()
2
3  print(X_treino)
4
```

Na sequência criamos nossas variáveis X_treino, y_treino, X_teste e y_teste para onde serão desempacotados e atribuídos os dados de fashion_mnist por meio da função load_data( ).

```
[[[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 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 0 0]
  [0 0 0 ... 0 0 0]]

 ...

 [[0 0 0 ... 0 0 0]
```

Por meio de nossa função print( ) parametrizada com X_teste podemos ver que para esta variável estão carregados uma série de dados em forma matricial.

```
(X_treino, y_treino), (X_teste, y_teste) = fashion_mnist.loa

print(X_treino[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   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   0   0  13  73   0
    0   1   4   0   0   0   0   1   1   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   3   0  36 136 127  62
   54   0   0   0   1   3   4   0   0   3]
 [  0   0   0   0   0   0   0   0   0   0   0   0   6   0 102 204 176 134
  144 123  23   0   0   0   0  12  10   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0 155 236 207 178
  107 156 161 109  64  23  77 130  72  15]
 [  0   0   0   0   0   0   0   0   0   0   0   1   0  69 207 223 218 216
  216 163 127 121 122 146 141  88 172  66]
 [  0   0   0   0   0   0   0   0   0   1   1   1   0 200 232 232 233 229
  223 223 215 213 164 127 123 196 229   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0 183 225 216 223 228
  235 227 224 222 224 221 223 245 173   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0 193 228 218 213 198
  180 212 210 211 213 223 220 243 202   0]
 [  0   0   0   0   0   0   0   0   0   1   3   0  12 219 220 212 218 192
  169 227 208 218 224 212 226 197 209  52]
 [  0   0   0   0   0   0   0   0   0   0   6   0  99 244 222 220 218 203
  198 221 215 213 222 220 245 119 167  56]
```

Lembrando que, como temos um objeto em forma matricial, é perfeitamente possível consultar elemento por elemento desta matriz apenas referenciando seu índice.

Dessa forma temos a projeção dos dados referentes a cada pixel que compõe um determinado elemento.

A partir dessa projeção, podemos inclusive identificar que tais dados se encontram em uma escala que vai de 0 a 255, representando um canal de cor apenas.

```
X_treino = X_treino / 255.0
X_teste = X_teste / 255.0

print(X_treino[0])

```

Sendo assim, por uma questão tanto de compatibilidade quanto de performance, é interessante realizar uma normalização desses dados em uma escala menor.

Atualizando cada variável dividindo o valor do conteúdo da mesma por 255 estamos convertendo os valores iniciais de uma escala 0-255 para uma escala 0-1.

```
[[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.         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.
  0.00392157 0.         0.         0.05098039 0.28627451 0.
  0.         0.00392157 0.01568627 0.         0.         0.
  0.         0.00392157 0.00392157 0.        ]
 [0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.         0.         0.
  0.01176471 0.         0.14117647 0.53333333 0.49803922 0.24313725
  0.21176471 0.         0.         0.         0.00392157 0.01176471
  0.01568627 0.         0.         0.01176471]
 [0.         0.         0.         0.         0.         0.
```

Visualizando o mesmo elemento do parágrafo anterior é possível ver que de fato em seu mapa de características seus valores foram escalonados entre 0 e 1.

```python
print(X_treino.shape)

X_treino = X_treino.reshape(-1, 28*28)

print(X_treino.shape)

X_teste = X_teste.reshape(-1, 28*28)

```

```
(60000, 28, 28)
(60000, 784)
```

Outro possível tratamento dos dados é realizar uma remodelagem do formato da base de dados, mesclando duas dimensões da mesma apenas com características do formato das imagens para uma dimensão só.

Nesse caso, atualizando o conteúdo das variáveis por meio da função reshape( ), realizamos tal remodelagem.

Note que inicialmente tínhamos uma matriz com 60000 amostras, cada uma delas em tamanho 28x28 (28 pixels de altura por 28 pixels de largura).

Após a conversão, onde simplesmente multiplicamos largura por altura, temos uma nova base de dados bidimensional, com as mesmas 60000 amostras, agora com 784 pontos de características referentes a cada pixel de sua composição.

```
1   modelo = tf.keras.models.Sequential()
2
3   print(modelo)
4
5   modelo.add(tf.keras.layers.Dense(units = 128,
6                                    activation = 'relu',
7                                    input_shape = (784, )))
8   modelo.add(tf.keras.layers.Dropout(0.2))
9   modelo.add(tf.keras.layers.Dense(units = 10,
10                                   activation = 'softmax'))
11  modelo.compile(optimizer = 'adam',
12                 loss = 'sparse_categorical_crossentropy',
13                 metrics = ['sparse_categorical_accuracy'])
14
```

Uma vez terminada a fase de tratamento dos dados podemos partir para a elaboração da arquitetura da rede neural artificial.

De forma parecida com o exemplo anterior, por meio de algumas estruturas pré-configuradas da biblioteca keras iremos realizar este processo.

Inicialmente criamos uma variável de nome modelo que atribuída para si possui a função Sequential( ), nesta linha inicializando esta classe assim como todas suas funções.

Em seguida por meio da função add( ) criamos uma camada de entrada onde por meio de seus parâmetros definimos que a mesma terá 128 neurônios, usará do método de ativação ‘relu’, bastante eficiente para modelos de classificação, finalizando com o formato da camada de entrada com base no formato de nossos dados normalizados anteriormente.

Na sequência criamos uma camada Dropout( ), que como visto no exemplo anterior, estipula um número de amostras a serem eliminadas entre camadas para forçar a rede neural a trabalhar compensando a falta de dados das mesmas. Parametrizando Dropout( ) com 0.2 estamos definindo que nesta etapa 20% das amostras serão removidas aleatoriamente desta para a camada subsequente.

Por fim criamos nossa camada de saída, onde por meio de Dense( ) agora parametrizado com units = 10 estamos definindo o número de saídas, assim como para activation = ‘softmax’ estamos usando dessa função de ativação para que o resultado do processamento dos dados gere uma probabilidade entre as 10 categorias definidas.

Sempre que terminada a criação das camadas de nossa rede neural, é necessário realizar a compilação do modelo. Para isso simplesmente usamos da função compile( ) parametrizando a mesma com optimizer = ‘adam’, loss = ‘sparse_categorical_crossentropy’, que em outras palavras significa que o processamento dos dados deverá considerar como métrica a proximidade/semelhança dos dados para diferenciá-los em possíveis categorias, finalizando com metrics = [‘sparse_categorical_accuracy’] definimos que a métrica usada como parâmetro para medir a eficiência do processo de aprendizado de máquina desta rede neural deve se basear em maior precisão possível para o processo de categorização.

```
<tensorflow.python.keras.engine.sequential.Sequential object at 0x7fdf64a5b198>
```

Apenas por curiosidade, como retorno de nossa função print( ) parametrizada com nosso modelo, é retornado que o mesmo é um objeto alocado em memória, pronto para uso a partir desta etapa de criação.

```
1  print(modelo.summary())
2
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 128)               100480
_________________________________________________________________
dropout (Dropout)            (None, 128)               0
_________________________________________________________________
dense_1 (Dense)              (None, 10)                1290
=================================================================
Total params: 101,770
Trainable params: 101,770
Non-trainable params: 0
_________________________________________________________________
None
```

Por meio da nossa função print( ), agora parametrizada com modelo.summary( ) podemos ver em resumo a estrutura de nossa rede neural artificial, que para esse modelo, apesar de possuir poucas camadas, em suas camada intermediárias chega a realizar cerca de 100480 interconexões entre neurônios destas camadas.

```
1  modelo.fit(X_treino,
2             y_treino,
3             epochs = 100)
4
```

```
Epoch 1/100
1875/1875 [==============================] - 4s 2ms/step - loss: 0.6858 - sparse_categorical_accuracy: 0.7610
Epoch 2/100
1875/1875 [==============================] - 3s 2ms/step - loss: 0.4117 - sparse_categorical_accuracy: 0.8524
Epoch 3/100
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3708 - sparse_categorical_accuracy: 0.8636
Epoch 4/100
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3445 - sparse_categorical_accuracy: 0.8729
Epoch 5/100
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3308 - sparse_categorical_accuracy: 0.8770
Epoch 6/100
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3215 - sparse_categorical_accuracy: 0.8802
Epoch 7/100
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3094 - sparse_categorical_accuracy: 0.8858
Epoch 8/100
```

Encerrando essa etapa, por meio da função fit( ) alimentamos nossa rede neural também definindo para a mesma quantas execuções deverão ser realizadas.

Nesse caso, alimentamos a mesma com os dados de X_treino e y_treino, estipulando que processe esses dados cem vezes.

Ao executar esse bloco de código é possível acompanhar via terminal cada fase de execução.

```
Epoch 93/100
1875/1875 [==============================] - 3s 2ms/step - loss: 0.1366 - sparse_categorical_accuracy: 0.9463
Epoch 94/100
1875/1875 [==============================] - 3s 2ms/step - loss: 0.1318 - sparse_categorical_accuracy: 0.9486
Epoch 95/100
1875/1875 [==============================] - 3s 2ms/step - loss: 0.1394 - sparse_categorical_accuracy: 0.9457
Epoch 96/100
1875/1875 [==============================] - 3s 2ms/step - loss: 0.1366 - sparse_categorical_accuracy: 0.9467
Epoch 97/100
1875/1875 [==============================] - 3s 2ms/step - loss: 0.1353 - sparse_categorical_accuracy: 0.9468
Epoch 98/100
1875/1875 [==============================] - 3s 2ms/step - loss: 0.1372 - sparse_categorical_accuracy: 0.9467
Epoch 99/100
1875/1875 [==============================] - 3s 2ms/step - loss: 0.1361 - sparse_categorical_accuracy: 0.9485
Epoch 100/100
1875/1875 [==============================] - 3s 2ms/step - loss: 0.1352 - sparse_categorical_accuracy: 0.9473
```

Após alguns minutos de execução, via terminal podemos ver que, como esperado, após o processamento dos dados por todas as épocas definidas a rede neural atinge uma margem de acertos por volta de 94%.

```
1  modelo_json = modelo.to_json()
2  with open("fashion_modelo.json", "w") as json_file:
3      json_file.write(modelo_json)
4
```

Uma prática comum e não abordada até o momento é o de salvar sua rede neural, tanto em estrutura quanto os dados de aprendizado de máquina para que possamos importar tais dados para outros exemplos.

Raciocine que o processo de criação de uma rede neural artificial se dá de modo que após criada, configurada e testada obtendo boa margem de acertos, a mesma possa ser salva em seu estado para que para novas implementações não seja necessário criar e testar toda a rede neural.

Muito pelo contrário, todo cientista de dados possui seus modelos prontos para uso conforme a necessidade.

Para salvar o modelo basta que instanciemos o mesmo, salvando seu conteúdo para um arquivo por meio da função to_json( ).

Repare que indentado ao corpo da função to_json( ) por meio da função open( ) definimos um nome para o arquivo assim como repassamos o parâmetro “w” para que o mesmo possa ser lido sem incompatibilidades, tudo isso como um arquivo do tipo json_file.

Finalizando esta etapa por meio da função json_file.write( ) atualizamos o o conteúdo de modelo_json.

```
1  modelo.save_weights("fashion_modelo.h5")
2
```

Um processo parecido é feito para salvar os pesos, ou em outras palavras, os padrões de dados de aprendizagem salvos para aquele modelo.

Nesse caso, basta aplicar sobre nosso modelo o método save_weights( ) especificando um nome para o arquivo e definindo a extensão do mesmo como .h5.

Para reutilização desses dados de aprendizagem em um novo modelo, ou para um novo teste, basta realizar o processo de leitura destes pesos o associando a uma variável, e o processo de leitura é feito por meio da função modelo.load_weights( ), por sua vez parametrizada com o nome do artuivo .h5.

*Apenas fazendo um breve adendo, é interessante sempre ter em mente que o tempo de execução de uma rede neural está diretamente ligado com o hardware no qual a mesma está rodando assim como o volume de dados e o número de épocas de execução definida para a rede neural.

ANÁLISE FINANCEIRA COM PYTHON

* Nível Básico

Visualização de Dados

Como mencionado anteriormente, é esperado que você tenha um certo nível de conhecimento da linguagem Python para abstrair e entender cada bloco de código que estaremos criando na sequência.

Mesmo familiarizado com a linguagem Python, talvez você nunca tenha precisado fazer uso de bibliotecas científicas como Numpy e Pandas, por exemplo, ou com plotagem de gráficos para apresentação de certos tipos de dados.

Porém, iniciaremos nossas implementações já fazendo uso destas e de outras bibliotecas, desmistificando muita coisa desse meio e construindo uma bagagem de conhecimento sobre mercado financeiro ao mesmo tempo, realizamos as devidas análises sobre os dados oriundos deste meio.

Vamos fazer isso de forma gradual, começando por conceitos bastante básicos e avançaremos passo-a-passo desde uma simples análise de dados até a criação de um Agente autônomo dotado de inteligência artificial.

Boa parte do que chamamos de Ciência de Dados (Data Science) é justamente obter acesso a uma série de dados brutos, realizar o polimento e processamento sobre os mesmos para por fim ter em mãos dados consistentes de onde é possível extrair informações ou até mesmo realizar previsões.

Tenha em mente que agora, inicialmente, estaremos realizando uma simples importação e leitura de dados, nos capítulos subsequentes estaremos fazendo uso de “tecnologias” mais avançadas como redes neurais artificiais. É importante você fazer as coisas no seu tempo, recomendo fortemente que você só avance para os capítulos seguintes quando de fato se sentir seguro para avançar pois gradualmente à medida que avançamos trabalharemos com estruturas de dados de maior complexidade.

Para darmos início a parte prática de nossos estudos, é interessante fazer alguns esclarecimentos iniciais, entendendo ao menos a lógica do que será visto nas páginas subsequentes.

Quando estamos falando de visualização de dados, estamos falando diretamente da maneira como iremos (desculpe a redundância) visualizar os mesmos, mas de forma mais intuitiva.

Tenha em mente que ao criar suas ferramentas você terá domínio sobre cada ponto que compõe a mesma, porém, uma prática comum a todo desenvolvedor é imaginar como esses dados serão mostrados para seu cliente, um leigo no assunto.

Seguindo essa lógica, estaremos realizando processos de importação e análise de dados, de forma a para cada ponto de interesse criar uma maneira de representar visualmente os retornos destes dados.

Criar toda uma arquitetura de código voltada ao mercado financeiro, não diferentemente das outras áreas, envolve uma série de procedimentos onde o resultado final de cada aplicação deve ser gerada e exibida por meios e métodos que tornem tal apresentação concisa e informativa para o público comum.

Como mencionado anteriormente, gradativamente iremos criar nossas ferramentas de análise, entendendo qual a aplicação das mesmas no mercado financeiro, assim como realizar as devidas apresentações e interpretações dos resultados.

Partindo para prática:

```
import numpy as np
import pandas as pd
from pandas_datareader import data as wb
import matplotlib.pyplot as plt
```

Todo processo de construção de nossos códigos começa com as importações das bibliotecas, módulos e demais ferramentas que utilizaremos ao longo do código e que não são pré-carregadas em nossa IDE.

Lembrando que a esse ponto essas ferramentas já devem estar devidamente instaladas conforme explicado em capítulos anteriores.

Para que seja possível realizar nossas análises, inicialmente importamos algumas bibliotecas, a primeira delas, a biblioteca Numpy, é uma biblioteca que nos permitirá trabalhar com dataframes (arrays/matrizes, mas do tipo numpy) aplicando uma série de funções lógicas e aritméticas sobre nossos dados.

Também é feita a importação da biblioteca Pandas, que por sua vez oferecerá uma série de ferramentas para importação e manipulação de nossos dados.

Em seguida é feita a importação da biblioteca pandas_datareader, não da biblioteca em si, mas as funções wb contidas no módulo data da mesma, isso nos possibilitará realizar a importação de bancos de dados diretamente de algum site.

Por fim é importada a biblioteca MatPlotLib, biblioteca composta de uma coleção de ferramentas que nos permitirá uma visualização mais intuitiva de nossos dados por meio de gráficos e outras representações visuais.

```
data = wb.DataReader('PG', data_source='yahoo', start='2000-1-1')
```

Em seguida iniciamos o processo de importação de nossa base de dados, que poderia ser local a partir de qualquer arquivo, mas estaremos usando dos dados importados do site Yahoo Finanças, por uma questão preferencial de carregar os dados mais atualizados o possível para nossa análise.

Para isso inicialmente criamos nossa variável data que recebe como atributo a função DataReader( ) da biblioteca pandas_datareader. Como parâmetros repassamos para mesma ‘PG’, que nada mais é do que os dados referentes a companhia Procter & Gamble; data_source = ‘yahoo’, que em outras palavras

significa que estaremos carregando tais dados a partir do site Yahoo Finanças; start = ‘2000-1-1’ define um intervalo de tempo, começando no dia 1 de janeiro de 2000 até o dia atual.

```
data.head()
```

Não havendo nenhum erro no carregamento dos dados podemos começar a explorar os mesmos de forma básica. Por exemplo, por meio da função head( ) é exibido em tela/console os 5 primeiros registros encontrados.

	High	Low	Open	Close	Volume	Adj Close
Date						
2000-01-03	53.96875	52.84375	53.90625	53.59375	4275000.0	28.391300
2000-01-04	54.18750	51.87500	53.18750	52.56250	4270800.0	27.844984
2000-01-05	52.68750	51.25000	52.43750	51.56250	5098400.0	27.315243
2000-01-06	54.71875	52.53125	52.56250	53.93750	6524200.0	28.573389
2000-01-07	58.50000	54.59375	54.78125	58.25000	9832000.0	30.857954

Começando o entendimento desses dados, note que o que nos é apresentado nada mais é do que os dados dispostos num formato muito parecido com o de uma tabela Excel, organizados num padrão de tabela composta por linhas e colunas.

A primeira coluna, lendo da esquerda para a direita, Date, é a coluna referente ao dia apresentado, iniciando no primeiro dia útil do ano 2000 que logicamente foi o dia 3 de janeiro.

Na sequência temos a coluna High, apresentando o maior valor alcançado pelas ações da P&G naquele dia. Da mesma forma Low exibe os dados de menor valor alcançado, Open mostra o valor das ações no momento da abertura do mercado, Close mostra o valor das ações no momento do fechamento do mercado, Volume mostra o número de ações negociadas naquele dia e finalmente Adj Close mostra os dados no momento do fechamento do mercado considerando a aplicação de todas taxas envolvidas nas operações.

No mercado financeiro se trabalha com esses valores de modo geral, uma vez que a “bolsa de valores” tem um expediente definido, onde durante o expediente, conforme novas operações sejam processadas, os valores dos ativos oscilam de acordo com os resultados das operações.

```
data.tail()
```

	High	Low	Open	Close	Volume	Adj Close
Date						
2020-05-28	116.989998	114.900002	115.529999	116.059998	6586700.0	116.059998
2020-05-29	116.720001	114.620003	116.129997	115.919998	7987900.0	115.919998
2020-06-01	117.500000	115.260002	116.000000	117.250000	5105600.0	117.250000
2020-06-02	118.139999	116.820000	117.070000	118.059998	6327900.0	118.059998
2020-06-03	119.190002	117.949997	118.099998	118.529999	6169800.0	118.529999

Da mesma forma como exibimos certos dados via função head( ), podemos ver o outro extremo desta base de dados por meio da função tail( ) que por sua vez exibirá os 5 últimos resultados encontrados na base de dados.

Estes por sua vez, contendo os valores em cada coluna da tabela referentes ao dia atual, ao último dia lido no carregamento dos dados do Yahoo Finanças. (No momento da escrita deste livro, dados referentes ao dia 3 de junho de 2020).

Outro ponto a destacar é que neste momento estamos fazendo o uso de todos os dados existentes, diferentemente do que será visto nos capítulos subsequentes onde para treinamento de uma rede neural artificial iremos descartar boa parte dos dados que julgamos desnecessários naquele contexto. Por hora, toda e qualquer informação é útil quando estamos a fazer uma análise simples dos dados.

```
data['Resultado'] = (data['Adj Close'] / data['Adj Close'].shift(1)) -1
print(data['Resultado'])
```

Em seguida criamos uma nova coluna chamada ‘Resultado’ onde serão repassados os dados da diferença entre os valores de fechamento ajustado do dia atual em relação ao dia anterior.

Repare que para isto, basta usar a notação de [ ] diretamente na variável base, desde que haja algo a ser atribuído para esse novo campo. Se não houver nenhum erro de sintaxe nesse processo a coluna será criada como o esperado.

Por meio da função print( ) parametrizada com os dados de ‘Resultado’, o retorno esperado é:

	High	Low	Open	Close	Volume	Adj Close	Resultado
Date							
2000-01-03	53.96875	52.84375	53.90625	53.59375	4275000.0	28.391300	NaN
2000-01-04	54.18750	51.87500	53.18750	52.56250	4270800.0	27.844984	-0.019242
2000-01-05	52.68750	51.25000	52.43750	51.56250	5098400.0	27.315243	-0.019025
2000-01-06	54.71875	52.53125	52.56250	53.93750	6524200.0	28.573389	0.046060
2000-01-07	58.50000	54.59375	54.78125	58.25000	9832000.0	30.857954	0.079954

Novamente por meio da função head( ) ou tail( ) é possível ver que foi de fato adicionada uma nova coluna ao final da tabela.

```
Date
2000-01-03         NaN
2000-01-04   -0.019242
2000-01-05   -0.019025
2000-01-06    0.046060
2000-01-07    0.079954
                 ...
2020-05-28    0.019053
2020-05-29   -0.001206
2020-06-01    0.011473
2020-06-02    0.006908
2020-06-03    0.003981
Name: Resultado, Length: 5137, dtype: float64
```

Retomando nosso raciocínio, por meio da função print( ) parametrizada com data[‘Resultado’] podemos visualizar a diferença entre os valores de fechamento corrigidos.

Note que o primeiro valor está representado com NaN, sigla para quando não há dados naquele campo, neste exemplo, a criação da base de dados se dá pelos valores a partir do dia 3 de janeiro, não existe uma relação desses valores com o dia anterior pois o mercado estava fechado, ficando uma lacuna representada por NaN.

Importante lembrar também que nesse exemplo existe apenas um dado faltante, em outras bases de dados onde houverem ocorrência de muitos dados faltantes é necessário realizar o polimento desses dados removendo-os para que não atrapalhem o processamento dos demais dados da base, principalmente no âmbito das redes neurais artificiais.

```
data['Resultado'].plot(figsize=(8,5))
plt.show()
```

Seguindo com o código, a partir deste momento já é possível realizar a visualização dos dados de forma mais intuitiva. Lembre-se que ao início do código realizamos a importação da biblioteca MatPlotLib, pois bem, é por meio dela que estaremos gerando nosso primeiro gráfico. Recomendo fortemente que você leia a documentação desta biblioteca, pois lá estão os modelos aplicados para cada tipo de gráfico que é possível ser gerado por meio dessa biblioteca.

Instanciando apenas a coluna ‘Resultado’ dos dados de data, podemos simplesmente acrescentar o sufixo .plot( ) chamando esta função e lhe passando como parâmetro, por hora, apenas o tamanho do gráfico.

Para exibir o gráfico chamamos a função plt.show( ).

Eis que temos um gráfico gerado com base nos dados contidos em ‘Resultado’, onde podemos ver ao longo de 20 anos qual foi a variação média destes dados.

```
media_simples = data['Resultado'].mean()
print(media_simples)
```

Na sequência é possível visualizar um valor médio de todos os valores lidos anteriormente por meio da função .mean( ). Para isso criamos uma variável de nome media_simples que recebe os dados de ‘Resultado’ de data, aplicando essa métrica que nos retorna o valor médio de todos resultados.

Nesse caso o valor retornado é:

0.00037451045789188374

Porém este não é um dado muito interessante quando lido dessa forma, uma vez que é um valor médio para 20 anos de dados. Na verdade, um número apresentado dessa forma pode ser

facilmente interpretado da maneira errada, então, vamos aplicar outras métricas para tornar nossos retornos mais lógicos, úteis e atraentes a leitura.

```
media_anual = data['Resultado'].mean() * 250
print(media_anual)
```

Realizando um processo muito parecido com o anterior, usando dos mesmos dados, é possível realizar a multiplicação deste valor por 250, o que em outras palavras estamos corrigindo a apresentação desses valores entendendo que os mesmos são referentes aos 250 (média de) dias úteis do ano.

Neste caso o retorno será:

0.09362761447297094

O que ainda é um dado suscetível a erro de leitura para o leigo. Podemos melhorar esta apresentação ainda mais.

```
print(str(round(media_anual, 5)*100)+'%')
```

Por meio da função print( ) podemos fazer um simples procedimento de “arredondamento” dos valores encontrados anteriormente por meio da função round( ) assim como podemos transformar estes dados em um valor percentual os multiplicando por 100.

Neste caso o retorno será:

9.363%

E assim, finalmente temos um dado claro dizendo que ao longo do período de 20 anos, o valor de um determinado ativo teve uma variação média de 9%. Em outras palavras, a taxa de retorno deste ativo variou 9% em média durante todo o período de forma geral.

Dessa forma, realizando um simples tratamento dos dados obtivemos o que neste meio é chamado de retorno aritmético, ou retorno simples de um determinado ativo.

```
data['RetLogaritmico'] = np.log(data['Adj Close'] / data['Adj Close'].shi
ft(1))
print(data['RetLogaritmico'])
```

Na sequência, usando de lógica parecida, é possível calcular o retorno logarítmico ao longo de um período de tempo, essa métrica, diferentemente da anterior, considera a variação do(s) ativo(s) em períodos de tempo específicos.

Raciocine que para você, por exemplo, estimar o valor de uma ação, usando de um dado registrado em um período de tempo muito longo irá prejudicar sua análise, uma vez que os mercados mudam constantemente e dados antigos são defasados.

Dentro dessa lógica, um retorno que de fato nos ajudará a estimar novos valores deve ser obtido por meio de intervalos de tempo próximos e de mesmo padrão estatístico.

Repare no código que, da mesma forma como feito anteriormente, é criada uma nova coluna para data, agora de nome 'RetLogaritmico' que recebe os dados da função np.log( ) parametrizado com a diferença dos dados de fechamento com o período de tempo definido manualmente por meio da função shift( ).

```
Date
2000-01-03         NaN
2000-01-04   -0.019430
2000-01-05   -0.019208
2000-01-06    0.045031
2000-01-07    0.076919
                 ...
2020-06-04   -0.021145
2020-06-05    0.019456
2020-06-08    0.006066
2020-06-09   -0.005982
2020-06-10    0.007493
Name: RetLogaritmico, Length: 5142, dtype: float64
```

Por meio da função print( ) podemos visualizar estes dados passando como parâmetro os dados/valores da coluna

‘RetLogaritmico’.

```
Date
2000-01-03         NaN
2000-01-04   -0.019242
2000-01-05   -0.019025
2000-01-06    0.046060
2000-01-07    0.079954
                 ...
2020-05-28    0.019053
2020-05-29   -0.001206
2020-06-01    0.011473
2020-06-02    0.006908
2020-06-03    0.003981
Name: Resultado, Length: 5137, dtype: float64
```

```
Date
2000-01-03         NaN
2000-01-04   -0.019430
2000-01-05   -0.019208
2000-01-06    0.045031
2000-01-07    0.076919
                 ...
2020-06-04   -0.021145
2020-06-05    0.019456
2020-06-08    0.006066
2020-06-09   -0.005982
2020-06-10    0.007493
Name: RetLogaritmico, Length: 5142, dtype: float64
```

Realizando uma simples comparação entre os valores obtidos como retorno simples e como retorno logarítmico é possível ver que, embora mínima, há uma diferença entre os valores, e já que estamos trabalhando sobre dados que impactarão nossas decisões no mercado financeiro precisamos levar em consideração a confiabilidade desses dados.

```
data['RetLogaritmico'].plot(figsize = (8,5))
plt.show()
```

Exatamente da mesma forma como feito anteriormente, também é possível plotar tais dados em um gráfico para fácil visualização.

Da mesma forma, realizar as devidas comparações.

```
medialog = data['RetLogaritmico'].mean() * 250
print(str(round(medialog, 5) * 100) + '%')
```

Assim como aplicar a métrica que transforma os dados em um valor percentual.

Neste caso o retorno estimado será:

6.978%

O que condiz mais com a realidade, sendo um dado de maior relevância se comparado ao valor de retorno simples.

Até o momento tudo o que fizemos foi simplesmente criar as estruturas básicas de importação dos dados para construção de nossas bases de dados e a aplicação de algumas simples métricas para o valor de um ativo fictício de uma empresa.

Porém, uma prática comum é trabalharmos usando de múltiplas carteiras que podem conter inúmeros ativos dentro de seu portfólio. Sendo assim, podemos criar uma estrutura em código que realize a importação de diferentes carteiras.

Raciocine que um investidor realiza essa leitura e com base nas tendências apresentadas passa a investir proporcionalmente nas carteiras que melhor lhe recompensarão. A essa altura você deve entender o porquê de tanto tempo é aplicado por parte do operador em suas carteiras, imagine acompanhar e manualmente realizar uma série de procedimentos para estimar qual melhor carteira naquele momento de especulação investir.

Nosso foco aqui é justamente automatizar todo esse processo, tendo inclusive uma margem de precisão muito superior e em muito menos tempo do que a maneira tradicional de corretagem.

```
carteiras = ['PG', 'MSFT', 'F', 'GE', 'AAPL']
```

```
database = pd.DataFrame()
```

Nesta etapa, inicialmente criamos uma variável de nome carteiras que receberá em forma de lista os dados de 5 empresas, também é criada uma segunda variável de nome database que por hora apenas chama a função pd.DataFrame( ) da biblioteca Pandas. Dessa forma, teremos múltiplos dados oriundos de diferentes carteiras, organizados em forma de matrizes onde podem ser realizadas as devidas leituras e aplicações de operações aritméticas.

```
for i in carteiras:
```

```
  database[i] = wb.DataReader(i, data_source='yahoo', start='2000-1-1')['Adj Close']

database.info()
database.head()
database.tail()
```

Para criação dessa nova base de dados, inicialmente por meio de um laço de repetição for realizamos a leitura a partir do Yahoo Finanças de cada carteira definida em carteiras. Aqui, para sermos mais práticos, realizamos a importação apenas dos dados de fechamento corrigido, atribuindo os mesmos para database.

Em seguida, se não houver nenhum erro durante o processo de importação dos dados, são apresentados um sumário, as 5 primeiras e as 5 últimas linhas de database.

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 5142 entries, 2000-01-03 to 2020-06-10
Data columns (total 5 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   PG      5142 non-null   float64
 1   MSFT    5142 non-null   float64
 2   F       5142 non-null   float64
 3   GE      5142 non-null   float64
 4   AAPL    5142 non-null   float64
dtypes: float64(5)
memory usage: 241.0 KB
```

Date	PG	MSFT	F	GE	AAPL
2020-06-04	116.050003	182.919998	6.57	7.74	322.320007
2020-06-05	118.330002	187.199997	7.34	7.88	331.500000
2020-06-08	119.050003	188.360001	7.53	8.46	333.459991
2020-06-09	118.339996	189.800003	7.24	8.02	343.989990
2020-06-10	119.230003	196.839996	6.81	7.61	352.839996

```
ret_carteiras = (database / database.shift(1)) -1
ret_carteiras.head()

pesos = np.array([0.2, 0.2, 0.2, 0.2, 0.2])
np.dot(ret_carteiras, pesos)

mediacarteiras = ret_carteiras.mean() * 100
print(mediacarteiras)

mediaretornoanual = ret_carteiras.mean() * 250
print(mediaretornoanual)

np.dot(mediaretornoanual, pesos)

portfolio = str(round(np.dot(mediaretornoanual, pesos), 5) * 100) + '
%'
print(portfolio)

pesos2 = np.array([0.3, 0.3, 0.15, 0.05, 0.2])
portfolio2 = str(round(np.dot(mediaretornoanual, pesos2), 5) * 100) +
 '%'
print(portfolio2)
```

Da mesma forma com que foi feito anteriormente sobre apenas um ativo, agora seguindo a mesma lógica podemos gerar uma estimativa de várias carteiras, onde as projeções nos indicarão quais carteiras (e todos seus ativos) serão mais lucrativas a longo prazo.

Repare que a nível de código temos uma diferença estrutural interessante, existe uma variável de nome pesos que inicialmente tem atribuída para si uma lista composta de valores iguais 0.2. Em outras palavras isso significa que inicialmente todas as carteiras serão lidas e processadas com o mesmo critério de importância.

Raciocine que ao início de nossas leituras não temos noção de qual ou quais carteiras serão mais lucrativas, logo, elas devem ter

igual importância no processamento. O parâmetro 0.2 por sua vez nada mais é do que 5 carteiras divididas igualmente para processamento, cada uma representando 20% do total de 5 carteiras (100%). Caso você inclua ou exclua carteiras desse portfolio este parâmetro deve ser alterado, caso contrário você estará gerando um erro a nível das dimensões dessa matriz.

Note que o código segue exatamente igual aos exemplos feitos anteriormente sobre apenas um ativo, agora gerando projeções simples e logarítmicas sobre todo conteúdo de todas as carteiras.

```
PG      0.036959
MSFT    0.050298
F       0.017561
GE     -0.003338
AAPL    0.123378
dtype: float64
PG      0.092398
MSFT    0.125744
F       0.043904
GE     -0.008345
AAPL    0.308444
dtype: float64
11.243%
13.33%
```

Fazendo uma leitura rápida do retorno obtido, é possível ver não só a projeção do retorno simples e logarítmico dessas carteiras como a margem de retorno individual das mesmas. Repare que nesse contexto, apenas como exemplo, os ativos da GE em ambas projeções geraram prejuízo, este compensado pelos outros ativos que geraram lucro, o que pode nos orientar a simplesmente não aplicar sobre os ativos desta carteira, eliminando ela do portfólio de investimentos ou a substituindo por outra.

```
ind_carteiras = ['^GSPC', '^IXIC', '^GDAXI']
indicadores = pd.DataFrame()

for t in ind_carteiras:
  indicadores[t] = wb.DataReader(t, data_source='yahoo', start='2000
-1-1')['Adj Close']
```

```
indicadores.head()
indicadores.tail()

(indicadores / indicadores.iloc[0] * 100).plot(figsize = (8,5))
plt.show()

retindicadores = (indicadores / indicadores.shift(1)) -1
retindicadores.tail()

retindicadoresanuais = retindicadores.mean() * 250
print(retindicadoresanuais)
```

Outra prática muito comum, é a do cálculo dos índices de retorno sobre uma carteira ou sobre um ativo. Diferentemente dos dados obtidos anteriormente, este índice representa o potencial de lucro de um investimento sobre uma empresa, proporcional ao seu investimento.

Entenda que quando você está disposto a investir uma determinada quantia de dinheiro sobre um ativo, ou uma carteira, ou um portfólio de uma empresa, sua participação nos lucros será proporcional ao seu investimento, obviamente. Logo, é interessante contextualizarmos para nossa realidade dessa forma, porém, com auxílio dessas ferramentas científicas podemos gerar estimativas de acordo com o ativo.

Note que a nível de código estamos fazendo praticamente o mesmo que anteriormente, porém dessa vez obtendo dados de ^GSPC, ^IXIC e ^GDAXI. Em outras palavras, estamos buscando índices em ^GSPC das 500 empresas com histórico mais confiável, de ^IXIC, também conhecido como Nasdaq, índices históricos em geral e finalmente de ^GDAXI os dados das 30 melhores empresas (em performance) com base nos seus índices na bolsa de Frankfurt.

O interessante a se reparar aqui é que, novamente, estamos fazendo em poucas linhas de código o que um estatístico poderia demorar semanas coletando dados e gerando os prospectos sobre os mesmos.

S&P 500 (^GSPC)
SNP - SNP Real Time Price. Currency in USD
Add to watchlist
3,002.10 -188.04 (-5.89%)
At close: June 11 5:13PM EDT
Summary
Chart
Conversations
Historical Data
Options
Components
Previous Close 3,190.14
Open 3,123.53
Volume 4,123,277,874
Day's Range 2,999.49 - 3,123.53
52 Week Range 2,191.86 - 3,393.52
Avg. Volume 6,214,697,968
1D 5D 1M 6M YTD 1Y 5Y Max
Full screen
3,220.00
3,190.14
3,133.333
3,046.667
3,001.87
2,960.00
10 AM
12 PM
2 PM
4 PM

^GSPC

Nasdaq Composite
INDEXNASDAQ: .IXIC
+ Seguir
9.492,73 −527,62 (5,27%)
11 de jun. 17:15 GMT-4 · Exoneração de responsabilidade
Um dia
5 dias
1 mês
6 meses
YTD
Um ano
5 anos
Máx
9.491,42 16:00
Pré-fechamento
10.020,35
10.200
10.000
9.800
9.600
9.400
10:00
12:00
14:00
16:00
Abertura 9.791,24
Alta 9.868,08
Baixa 9.491,30

^IXIC

^GDAXI

Realizadas as devidas importações e executado esse bloco de código, vamos ao retorno gerado.

```
^GSPC     0.055013
^IXIC     0.072683
^GDAXI    0.041469
dtype: float64
```

Realizando uma rápida leitura do gráfico e da legenda gerada podemos ver facilmente que, por exemplo, investir sobre os ativos do escopo ^GSPC trouxe no intervalo de tempo definido anteriormente um retorno médio de 5%, ^IXIC 7% e ^GDAXI 4%.

Lembrando que este tipo de índice é o retorno médio proporcional ao investimento, e que como você pode visualizar no gráfico, por meio de suas linhas ascendentes, houveram épocas em que tais carteiras de ativos tiveram maior valorização, aqui apenas como exemplo, em um intervalo de 20 anos, onde houveram literalmente muitos altos e baixos dentro do mercado financeiro, um valor investido por você em tais carteiras teria ao final de todo esse período obtido um lucro médio de acordo com os resultados.

Investimentos devem ser considerados de acordo com o prazo ao qual se espera um retorno positivo, existem tipos de investimentos mais rápidos, porém de maior risco, assim como

outros mais conservadores onde o retorno é a longo prazo, porém mais garantido.

Estimando os Riscos de um Investimento

```
carteiras = ['PG', 'AAPL']
database = pd.DataFrame()

for i in carteiras:
  database[i] = wb.DataReader(i,
                              data_source='yahoo',
                              start='2010-1-1')['Adj Close']
database.tail()

retorno = np.log(database / database.shift(1))
print(retorno)
```

Outra prática muito comumente adotada por quem opera no mercado financeiro é a de realizar diferentes formas de medir os riscos envolvidos em um investimento. Para não investir às cegas em um determinado grupo de ativos, nada mais normal do que fazer uma leitura de seu histórico, do cenário atual e tentar realizar estimativas com base nesses dados.

Usando de mais um exemplo diferente, vamos supor que temos a convicção de que o melhor a se fazer no momento é investir sobre os ativos da P&E e da Apple, porém vamos imaginar um cenário onde inicialmente iremos escolher em apenas uma delas para aplicar um capital.

Sendo assim, podemos realizar uma leitura de seus dados históricos e realizar algumas estimativas para orientar nossa escolha.

A nível de código note que importamos os dados exatamente da mesma forma como feito anteriormente, com a única diferença que estamos buscando dados em um intervalo de tempo menor, de

cerca de 10 anos, já usando somente dos dados de fechamento corrigidos e seu retorno logarítmico.

```
retorno['PG'].mean()
retorno['PG'].mean() * 250
retorno['PG'].std()
retorno['PG'].std() * 250 ** 0.5

retorno['AAPL'].mean()
retorno['AAPL'].mean() * 250
retorno['AAPL'].std()
retorno['AAPL'].std() * 250 ** 0.5
```

Em seguida realizamos tanto para os dados obtidos para PG quanto para AAPL os cálculos baseados nos valores médios, assim como o chamado desvio padrão por meio da função std( ). O chamado desvio padrão é uma métrica de leitura de dados estatísticos onde se mensura o quanto os dados estão agrupados em relação à média.

Em outras palavras, já que temos um registro histórico, é possível avaliar os valores médios ao longo do período assim como em determinados momentos essa média oscilou, com dados que se dispersaram muito ou pouco da média de acordo com o contexto.

Essa métrica é muito levada em consideração para medir os riscos pois o mercado oscila constantemente de acordo com inúmeros fatores, e essa oscilação pode atrapalhar nossos modelos no que diz respeito a confiabilidade dos dados interpretados e consequentemente das projeções realizadas.

```
print(f'Média PG', retorno['PG'].mean() * 250)
print(f'Média AAPL', retorno['AAPL'].mean() * 250)

print(f'Média entre PG e AAPL', retorno[['PG', 'AAPL']].mean() * 250)
print(f'Desvio Padrão entre PG e AAPL', retorno[['PG', 'AAPL']].std() *
 250 ** 0.5)
```

Realizando as devidas comparações, vamos ao retorno.

```
                   PG      AAPL
Date
2010-01-04       NaN       NaN
2010-01-05  0.000327  0.001728
2010-01-06 -0.004755 -0.016034
2010-01-07 -0.005438 -0.001850
2010-01-08 -0.001323  0.006626
...               ...       ...
2020-06-08  0.006066  0.005895
2020-06-09 -0.005982  0.031090
2020-06-10  0.007493  0.025402
2020-06-11 -0.025225 -0.049201
2020-06-12  0.005319  0.031300

[2629 rows x 2 columns]
Média PG 0.09281038979357492
Média AAPL 0.24469379688661624
Média entre PG e AAPL PG      0.092810
AAPL    0.244694
dtype: float64
Desvio Padrão entre PG e AAPL PG      0.171967
AAPL    0.275736
dtype: float64
```

Realizando uma simples leitura do retorno obtido, podemos ver o carregamento dos dados históricos, em seguida os retornos médios e o desvio padrão entre os mesmos. Este tipo de resultado, por hora nos diz que basicamente as ações baseadas na P&G foram um tanto quanto mais conservadoras, valorizaram menos ao longo do tempo porém dentro da média, enquanto as ações da Apple valorizaram mais se comparado com o mesmo período de tempo, porém com um desvio padrão mais acentuado.

Trazendo para nossa realidade sabemos que essa grande valorização da Apple se deu por uma série de eletrônicos que a mesma lançou de 2010 para cá, o que justifica a valorização, o que você deve ficar atento é quando uma determinada empresa valoriza seus ativos de forma que não condiz com a realidade, confirmando essas características com um desvio padrão alto pode ser o indicador de que a empresa está trabalhando sobre uma volatilidade muito grande, em outras palavras, assim como ela valorizou muito

repentinamente, a mesma pode desvalorizar muito de forma rápida também, levando junto consigo o seu investimento.

O mercado de ações é também baseado nestes tipos de características, se você estudar o mercado financeiro ou ao menos obter alguma consultoria de alguém que trabalha diretamente nesse meio, verá que o operador está sempre atento a tudo o que pode interferir na valorização ou desvalorização dos ativos de uma determinada empresa, realizando essa leitura diariamente, várias vezes durante o dia, para tentar identificar o momento certo de investir em um determinado portfólio.

Nesse exemplo apenas realizamos os entendimentos iniciais deste universo, colocando em prática os meios de como se obtém dados para que a partir dos mesmos possamos gerar informações relevantes para nossos investimentos. Conforme formos avançando os capítulos estaremos usando de meios mais complexos para realizar análise financeira.

Outros métodos e funções interessantes de serem implementadas:

Caso você já esteja familiarizado com Python deve saber bem que uma determinada ação pode ser realizada de diferentes formas, cabendo ao desenvolvedor escolher os meios e métodos aos quais se sente mais à vontade para trabalhar.

O mesmo ocorre com as bibliotecas desenvolvidas pela comunidade, não é difícil encontrar na estrutura de uma biblioteca ou módulo diferentes estruturas de função com o mesmo propósito ou com intercompatibilidade. Vamos ver mais algumas dessas possibilidades que podem tornar nossos códigos mais robustos.

No processo de importação de nossa base de dados, usamos da biblioteca pandas_datareader para realizar tal importação diretamente do site Yahoo Finanças, e fizemos isso usando da classe DataReader( ) da mesma. Visando melhor performance, é possível chamar apenas a função específica para importação do site Yahoo Finanças, por meio da função get_data_yahoo( ).

```
data = wb.get_data_yahoo('AAPL',
                         start=datetime.datetime(2019, 1, 1),
                         end=datetime.datetime(2020, 1, 1))
```

Note que, já que estamos usando da função específica, não é mais necessário passar a fonte ‘Yahoo’ como parâmetro de função, apenas qual carteira e que intervalo de tempo, neste caso, usando da própria biblioteca datetime do sistema.

Vimos em outro momento que é perfeitamente possível criar novas colunas em nosso dataframe, e usando da própria gama de possibilidades nativas do Python, podemos fazer este tipo de criação inclusive por meio de operadores.

Supondo que estamos a criar uma nova coluna com os valores da diferença entre os valores de abertura e do fechamento do mercado, isso pode ser feito diretamente aplicando um operador aritmético sobre tais colunas de nosso dataframe.

```
data['diferenca'] = data.Open - data.Close
print(data)
```

Note que estamos usando a notação de referenciar uma coluna de nome ‘diferenca’ em nossa variável data, o valor a ser atribuído para a mesma nada mais é do que os valores de Open – Close, sendo possível realizar essa subtração diretamente sobre tais colunas.

```
                 High         Low        Open  ...   Adj Close  diferenca
Date                                           ...
2019-01-02  158.850006  154.229996  154.889999  ...  154.794983  -3.029999
2019-01-03  145.720001  142.000000  143.979996  ...  139.376251   1.789993
2019-01-04  148.550003  143.800003  144.529999  ...  145.326126  -3.729996
2019-01-07  148.830002  145.899994  148.699997  ...  145.002686   0.770004
2019-01-08  151.820007  148.520004  149.559998  ...  147.766861  -1.190002
...                ...         ...         ...  ...         ...        ...
2019-12-24  284.890015  282.920013  284.690002  ...  282.831299   0.420013
2019-12-26  289.980011  284.700012  284.820007  ...  288.442780  -5.089996
2019-12-27  293.970001  288.119995  291.119995  ...  288.333313   1.320007
2019-12-30  292.690002  285.220001  289.459991  ...  290.044617  -2.059998
2019-12-31  293.679993  289.519989  289.929993  ...  292.163818  -3.720001
```

Como retorno da função print( ) podemos ver a coluna diferença juntamente das outras demais.

Dependendo o contexto de seus estudos, pode ser necessário realizar a aplicação de algumas funções sobre amostras selecionadas aleatoriamente da base de dados, e isso pode ser feito facilmente por meio da função sample( ).

```
amostras = data.sample(10)
print(amostras)
```

Repare que inicialmente foi criado uma variável de nome amostras, atribuído a ela está a função sample, parametrizada em 10, o que significa que serão colhidas 10 amostras aleatórias de toda a base de dados.

```
                 High         Low        Open  ...   Adj Close       dif
Date                                           ...
2019-03-08  173.070007  169.500000  170.320007  ...  170.215256 -2.589996
2019-12-06  271.000000  267.299988  267.480011  ...  269.339935 -3.229980
2019-11-21  264.010010  261.179993  263.690002  ...  260.683990  1.679993
2019-12-03  259.529999  256.290009  258.309998  ...  258.136963 -1.140015
2019-05-07  207.419998  200.830002  205.880005  ...  199.698502  3.020004
2019-10-21  240.990005  237.320007  237.520004  ...  238.576508 -2.989990
2019-01-29  158.130005  154.110001  156.250000  ...  151.619080  1.570007
2019-06-17  194.960007  192.169998  192.899994  ...  191.603317 -0.990005
2019-03-01  175.149994  172.889999  174.279999  ...  172.243118 -0.690002
2019-04-10  200.740005  198.179993  198.679993  ...  197.493378 -1.940002
```

Por meio da função print( ) é possível visualizar as amostras selecionadas aleatoriamente.

Rede Neural Artificial Recorrente Aplicada a Séries Temporais

* Nível Intermediário

Para entendermos de fato, e na prática, o que é uma rede neural artificial recorrente (nomenclatura usual para modelos de redes neurais artificiais aplicadas ao estudo analítico e estatístico de dados), vamos usar de uma das principais aplicações deste modelo, a de previsão de valores a partir de dados históricos (neste caso, valores de ações).

Em função de estarmos agora trabalhando com redes neurais artificiais, o que faremos, embora de lógica parecida, será diferente do modelo anterior onde usamos apenas de bibliotecas científicas para processar nossos dados.

O modelo de rede neural artificial 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 a leitura dos dados históricos assim como previsões de um futuro próximo. Para isso, usaremos de uma base real e atual extraída diretamente do site Yahoo Finanças. Para esse exemplo, usaremos das ações Petrobrás.

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.

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).

```
normalizador = MinMaxScaler(feature_range = (0,1))
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 previsto 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.

```
39 regressor.compile(optimizer = 'rmsprop',
40                   loss = 'mean_squared_error',
41                   metrics = ['mean_absolute_error'])
42 regressor.fit(previsores,
43               preco_real,
44               epochs = 100,
45               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.

```
previsores_teste.mean()
preco_real_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.

VISÃO COMPUTACIONAL

O que é Visão Computacional?

Dentre as inúmeras áreas da computação, uma das que têm ganhado muito destaque ao longo dos anos devido as suas aplicações é a chamada Visão Computacional, onde a partir de imagens, extraímos características, detectamos padrões e aplicamos funções com os mais diversos propósitos abstraindo o sentido da visão de forma computacional.

Você certamente já deve ter se deparado com aplicações de visão computacional, mesmo que de forma indireta, pois em determinados contextos temos situações onde uma imagem pode ser obtida de um arquivo ou até mesmo a partir de uma câmera em tempo real, onde a mesma é processada de forma a realizar a identificação de objetos, de pessoas, rastreamento, reconhecimento de rostos, leitura de ambientes para veículos autônomos, realidade aumentada, etc...

Para tal fim, diversas ferramentas foram criadas e aperfeiçoadas ao longo dos anos para que se permitisse de forma simples implementar mecanismos do que hoje é chamado de Visão Computacional.

Neste pequeno livro, estaremos entendendo de forma direta e prática as principais funcionalidades de algumas bibliotecas como OpenCV, SKImage e Pillow, estas, com uma série de ferramentas dedicadas a visão computacional.

A biblioteca OpenCV, criada no início dos anos 90 para linguagens de programação como Python e C++, sofreu ao longo dos anos significativas melhorias em suas funções, além de implementação de novas funcionalidades, tornando-a hoje uma das mais completas e robustas ferramentas de visão computacional. Não podemos esquecer de mencionar que tal biblioteca também acabou por servir como base para outras de mesmo propósito.

Por exemplo, a biblioteca Pillow, inicialmente nomeada de PIL (Python Imaging Library), mais moderna e natura da linguagem Python oferece ferramentas equivalentes às do OpenCV, em certos casos, oferecendo melhor performance.

Se tratando de bibliotecas específicas para visão computacional, o ponto é que não só as citadas anteriormente, mas outras menos relevantes, à sua maneira oferecem ferramentas para que possamos de forma descomplicada criar nossas aplicações.

No âmbito da pesquisa científica em tratamento e processamento de imagens, com uma rápida pesquisa podemos encontrar uma série de artigos científicos publicados e revisados ao longo dos últimos anos onde é possível se obter referências para criação de algoritmos que codificados por meio de funções nas bibliotecas anteriormente citadas.

Neste pequeno livro, não vamos nos ater a discussão de tais artigos científicos, sempre que necessário estaremos entendendo e revisitando tópicos importantes do referencial teórico, porém nosso propósito é focar na implementação de tais ferramentas, métodos e meios de processamento de imagem.

Também abordaremos outras modalidades de visão computacional, como o processamento de imagens a partir de redes neurais artificiais, por exemplo, de modo a explorar diferentes técnicas e métodos complementares de aquisição de dados a partir de imagens.

Processo de Formação de Imagem

Quando estamos falando em imagem, basicamente estamos falando da interação da luz visível com algum objeto qualquer. Biologicamente falando, como mamíferos placentários temos olhos

desenvolvidos e adaptados de modo a captar certas frequências do espectro de luz chamado espectro visível.

Por parte da física, este espectro é muito limitado se comparado com outras frequências de energia aos quais nossos olhos simplesmente não são capazes de captar.

Anatomicamente nossos olhos são ferramentas evolutivas espetaculares, formados por uma órbita com parte dedicada a formar um sistema de lente, assim como todo um sistema de células fotorreceptoras conectadas a um quiasma ótico que transpõe toda uma inervação para uma região cerebral especializada em interpretar os sinais enviados por tais células, posteriormente processados por nossas redes neurais.

Você se lembra do filme Matrix? Em um certo momento do filme Neo questiona se o lugar e o contexto onde está situado é real. Morpheus responde que, se o que ele pode enxergar, tocar, sentir ou experenciar através de seus sentidos é o que ele considera real, está enganado pois tudo são sinais nervosos enviados de nossos órgãos sensoriais e interpretados pelo cérebro.

De fato, em neurociência, do que conhecemos até o momento sobre certos mecanismos de nosso cérebro, por mais incrível que tal estrutura anatômica seja, ela é simplesmente incapaz de diferenciar a realidade vivida da realidade simulada. Até mesmo em coisas banais como os mecanismos de indução a um sonho é algo que, talvez por falta de propósito, nosso cérebro não distingue da realidade.

Em outras palavras, estudos apontam que se você, por exemplo, tocar uma música em um violão, imaginar que está tocando uma música em um violão, ou sonhar que está tocando uma música em um violão, as mesmas regiões do cérebro se ativam como se estivesse praticando o ato de tocar uma música em um violão.

Na computação, muito do que abstraímos e transformamos em tecnologias ao longo da história são, seja direta ou indiretamente,

mecanismos naturais que estudamos e entendemos de aspectos biológicos de um ser vivo e da forma como ele experencia o mundo ao seu redor.

A própria eletrônica analógica, com seus componentes, meios e métodos de manipular circuitos eletrônicos em aparelhos, assim como a eletrônica digital onde circuitos lógicos realizam chaveamentos binários para execução de tarefas, em muito condiz com a forma como nosso cérebro cria e executa sinapses nervosas.

Inclusive áreas como redes neurais artificiais e aprendizado de máquina literalmente abstraem neurônios biológicos em perceptrons (neurônios lógicos), sinapses em camadas de processamento de dados de entrada, interconexões e funções de ativação... simulando uma rede neural biológica para processar sinais em algoritmos que simulam nossos mecanismos de aprendizado.

Voltando ao básico, na computação, desde seus primórdios, mais do que automatizar funções com maquinários específicos, se criaram tecnologias de equipamentos programáveis e em paralelo a isso sistemas lógicos e interfaces de interação, onde por meio de dispositivos de entrada tínhamos controle visual das saídas por meio de dados apresentados em telas.

As interfaces de tela, hoje tão banais por serem comuns a praticamente qualquer coisa, no passado foi uma grande sacada da eletrônica para montar dispositivos que de acordo com certos padrões de ativação dos componentes de um circuito, exibiriam em displays informações sobre o mesmo.

Um display, por mais simples que seja, basicamente é composto por lâmpadas organizadas em certos padrões que permitem a ativação de grupos personalizados de lâmpadas de acordo com a necessidade. A configuração de lâmpadas em um display pode ser programada a partir de uma matriz, em diferentes combinações de lâmpadas acesas e apagadas conforme o contexto, gerando imagens lógicas como projeção de caracteres alfanuméricos com suas formas e características.

Claro que devido aos avanços da eletrônica, tais displays se tornaram muito mais robustos uma vez que se desenvolveram tecnologias como as lâmpadas coloridas em três canais de cores independentes assim como a miniaturização de tais componentes.

Hoje quando você adquire uma televisão com uma resolução de 4K (apenas como exemplo), esta informação nada mais diz que, dentro do tamanho de tela escolhido em polegadas, existe uma densidade de pixels na escala de 3840 pixels de largura por 2160 pixels de altura, numa proporção de 16:9, gerando em suas possíveis combinações um total de 8294400 pixels para compor uma imagem, algo imperceptível ao olho humano e ao mesmo tempo uma interface capaz de exibir imagens com uma riqueza de detalhes excelente.

Cada frame exibido em tela, independentemente da imagem que esteja sendo apresentada, é uma combinação única de ativação de lâmpadas assim como sua intensidade de cor, pixel a pixel, de modo a a partir de um dispositivo eletrônico, se criar uma janela para o mundo.

A experiência que temos ao enxergar o mundo a nossa volta por nossos meios naturais ou a partir de dispositivos eletrônicos é uma das experiências de vida mais sensacionais quando levada em consideração toda a complexidade envolvida.

Visão Computacional

Revisitado o básico do processo de formação de imagem, podemos avançar um pouco em sua lógica, entendendo o que se tornou todo um nicho dentro da computação chamado Visão Computacional.

Aproveitando parte dos conceitos do tópico anterior, se tratando de visão computacional claro que estaremos falando sempre a nível de imagem, porém, muito do que caracteriza esse nicho se dará por meios e métodos de abstrair sentidos de captação e interpretação de imagens assim como o processamento paralelo das mesmas.

Em outras palavras, grande parte do que criamos e aperfeiçoamos em termos de tecnologia são ferramentas onde conseguimos abstrair, por exemplo, as funcionalidades de um olho humano como detector de luz para geração de imagens.

Enquanto um olho biológico é composto de estruturas celulares que detectam diferentes frequências de luz visível, transformando as mesmas em pulsos elétricos para certas regiões do cérebro, todo dispositivo de entrada capaz de captar imagens as fará de modo a transformar todas suas características em dados aos quais conseguimos mapear e gerar imagens digitais a partir dos mesmos.

A computação gráfica é uma das áreas as quais sofreram maior aperfeiçoamento ao longo dos anos, graças ao exponencial aumento de poder computacional e desenvolvimento de hardware específico para tal fim, o processo de processamento de imagens evoluiu de tal forma que a manipulação de imagens hoje em dispositivos adequados nos faz ter dificuldades em distinguir uma projeção gerada por computador de uma projeção do mundo real captada e exibida em forma de imagem digital.

Você já deve ter se deparado com apresentações de modelos que por meio de uma interface de vídeo exibem em tela desde uma simples frase composta de símbolos e caracteres alfanuméricos até sistemas de holografia e realidade aumentada.

Em épocas passadas, uma simples fotografia demandava uma série de processos manuais que demandavam tempo e manipulação de materiais para obtenção do conteúdo final em forma de imagem. Hoje ler um livro a partir de uma tela de computador é algo tão banal, se comparado a tudo o que conseguimos fazer a partir do mesmo computador através da manipulação de imagens, que por muitas vezes nos vemos nos questionando quais serão os próximos passos em tecnologia já que de fato, conseguimos abstrair de forma plena tudo o que está no contexto de imagens.

Tudo isso é computação gráfica, tudo isso direta ou indiretamente é visão computacional.

Algoritmos de Detecção e Rastreamento

Partindo para uma outra abordagem do mesmo assunto, obviamente quando estamos falando de imagens digitais estamos falando em diferentes camadas de processamento de dados para que uma simples informação seja apresentada em uma interface de tela para o usuário.

Falando mais especificamente da parte de programação, tudo o que envolve captação, manipulação, transformação, processamento, reprodução, etc... está codificado em linguagens de alto nível para que a apresentação e interação com o usuário seja algo intuitivo e de fácil implementação.

Nos capítulos subsequentes partiremos de fato para prática, onde simplesmente o que faremos é trabalhar por meio de uma linguagem de programação e algumas bibliotecas da mesma, de modo a ter acesso e controle aos dispositivos de entrada, importar dados a partir dos mesmos e realizar os devidos processamentos por meio de algoritmos bastante simples, quando comparado a toda complexidade envolvida.

Em programação, como você bem sabe, todo e qualquer processo parte (explícito ou implícito) de algoritmos que por sua vez são as instruções lógicas de como uma determinada ação deve ser codificada, de modo que o interpretador de tal linguagem de programação reconheça e execute as instruções para geração de um determinado retorno.

Falando especificamente de Python e suas bibliotecas dedicadas a visão computacional, felizmente não precisamos criar tais métodos e meios do zero, praticamente tudo o que será feito será por meio de combinações de funções nativas do Python assim como funções pré-programadas das bibliotecas, tudo de forma simples e objetiva.

Redes Neurais Artificiais Convolucionais

Apenas concluindo nossa linha de raciocínio, não podemos de forma alguma deixar de falar sobre redes neurais artificiais.

Avançando um pouco conceitualmente, desde a década de 50 grandes cientistas da computação elaboraram e desenvolveram ferramentas que visavam abstrair a maneira como nosso sistema nervoso central funciona.

Dentro dessa lógica foram criamos modelos que transliteravam, por exemplo, um neurônio biológico em um neurônio artificial, adaptando para o mesmo praticamente todas funcionalidades da sua versão biológica, uma vez que ambos neurônios funcionam basicamente em processos de conexão e comunicação por meio de pulsos elétricos.

Nesse contexto, se idealizaram modelos de redes neurais complexas capazes de reconhecer padrões realizando processos de aprendizado de máquina para os mais diversos fins, desde classificação, regressão até redes neurais artificiais intuitivas capazes de simular uma IA em tempo real, aprendendo constantemente por diferentes técnicas.

Não distante disso, também há todo um nicho dentro da inteligência artificial que usa de visão computacional como ferramenta de obtenção de dados de entrada. Importante salientar que quando falamos em diferentes nichos nada impede que na prática combinemos estes para um propósito em comum.

Um veículo autônomo em execução possui diversos núcleos operando tanto isolada quanto simultaneamente com dados de entrada, processamento e dados de saída. Raciocine que um automóvel autônomo por sua vez, usa de visão computacional para “enxergar” o tracejado de seu caminho e os objetos que estão no mesmo, em paralelo a isso existe toda uma IA que lê e processa esses dados por meio de uma rede neural, por sua vez treinada com

todas informações relevantes do veículo, do trajeto e de tudo o que está envolvido nesse contexto.

Uma rede neural convolucional nada mais é do que um modelo de rede neural artificial dedicada ao processamento de imagens para detecção de padrões, de modo que sua arquitetura lê e processa dados a partir de imagens, realizando diversos processos internos de mapeamento em matrizes e aplicação de funções matemáticas de acordo com seu propósito final.

INSTALAÇÃO DAS DEPENDÊNCIAS

Instalação do Python

Todo conteúdo deste livro foi elaborado e posto em prática sobre a versão 3.8.1 da linguagem Python, assim como todos exemplos foram rodados via Google Colab, e por fim toda representação gráfica complementar aos exemplos foi feita por meio do plugin PythonPreview da IDE Visual Studio Code.

Para instalação da linguagem Python em seu sistema operacional basta acessar o site oficial https://www.python.org/downloads/ e baixar a versão referente ao seu sistema operacional.

```
Administrador: Prompt de Comando

C:\Windows\system32>pip install more-itertools
Collecting more-itertools
  Downloading more_itertools-8.4.0-py3-none-any.whl (43 kB)
     |[redacted]| 43 kB 80 kB/s
Installing collected packages: more-itertools
Successfully installed more-itertools-8.4.0

C:\Windows\system32>
```

Para instalação de bibliotecas externas, seja no path ou em um ambiente virtualizado em seu sistema, busque o comando correto para a instalação da mesma em https://pypi.org/ .

```
[ ]  1  !pip install pandasdatareader

Collecting pandasdatareader
  Downloading https://files.pythonhosted.org/packages/8e/ed/19fa
Building wheels for collected packages: pandasdatareader
  Building wheel for pandasdatareader (setup.py) ...
Installing collected packages: pandasdatareader
    Running setup.py install for pandasdatareader ...
```

Caso você trabalhe no ambiente virtualizado do Google Colaboratory, use da sintaxe correta para instalação de bibliotecas adicionais (quando necessário).

Instalação do OpenCV

Trabalhar com a biblioteca OpenCV envolve algumas particularidades pois a mesma ao longo dos anos foi sofrendo uma série de modificações, de modo que alguns recursos sofreram melhorias enquanto outros foram descontinuados por conta de incompatibilidade.

Dessa forma, é possível baixar a última release por meio do site oficial https://opencv.org/releases/ assim como do portal SourceForge.

Em certos exemplos estaremos utilizando de versões específicas da biblioteca OpenCV, quando forem esses casos, haverá uma nota orientando a baixar a versão específica para aquele contexto pois nem todas ferramentas foram mantida de versões antigas para versões atuais.

Para esses casos em particular o processo de instalação deverá ser feito pelo método tradicional, via CMD do Windows ou PowerShell da suite Anaconda, tomando cuidado para instalar a referida biblioteca no ambiente virtualizado correto.

Demais instalações

Para a grande maioria dos exemplos estaremos utilizando a combinação da linguagem Python com três bibliotecas base. A própria OpenCV por suas ferramentas específicas para tratamento de imagem e visão computacional, Numpy que nos permitirá realizar certas operações a nível das matrizes das imagens e a MatPlotLib, que nos permitirá a plotagem de imagens e gráficos sobre as mesmas de forma simples e funcional.

O processo de instalação destas bibliotecas se dá pelo simples comando de instalação via pip.

pip install numpy

pip install matplotlib

pip install dlib

https://www.lfd.uci.edu/~gohlke/pythonlibs/

Unofficial Windows Binaries for Python Extension Packages

by **Christoph Gohlke, Laboratory for Fluorescence Dynamics, University of California, Irvine.**

Updated on 29 October 2020 at 14:16 UTC.

This page provides 32- and 64-bit Windows binaries of many scientific open-source extension packages for the official CPython distribution of the Python programming language. A few binaries are available for the PyPy distribution.

The files are unofficial (meaning: informal, unrecognized, personal, unsupported, no warranty, no liability, provided "as is") and made available for testing and evaluation purposes.

Most binaries are built from source code found on PyPI or in the projects public revision control systems. Source code changes, if any, have been submitted to the project maintainers or are included in the packages.

Refer to the documentation of the individual packages for license restrictions and dependencies.

If downloads fail, reload this page, enable JavaScript, disable download managers, disable proxies, clear cache, use Firefox, reduce number and frequency of downloads. Please only download files manually as needed.

Use pip version 19.2 or newer to install the downloaded .whl files. This page is not a pip package index.

Many binaries depend on numpy+mkl and the current Microsoft Visual C++ Redistributable for Visual Studio 2015, 2017 and 2019 for Python 3, or the Microsoft Visual C++ 2008 Redistributable Package x64, x86, and SP1 for Python 2.7.

Install numpy+mkl before other packages that depend on it.

The binaries are compatible with the most recent official CPython distributions on Windows >=6.0. Chances are they do not work with custom Python distributions included with Blender, Maya, ArcGIS, OSGeo4W, ABAQUS, Cygwin, Pythonxy, Canopy, EPD, Anaconda, WinPython etc. Many binaries are not compatible with Windows XP or Wine.

The packages are ZIP or 7z files, which allows for manual or scripted installation or repackaging of the content.

The files are provided "as is" without warranty or support of any kind. The entire risk as to the quality and performance is with you.

The opinions or statements expressed on this page should not be taken as a position or endorsement of the Laboratory for Fluorescence Dynamics or the University of California.

Index by date: numpy open3d pyrxp aiohttp psutil orange rapidjson numpy-stl boost-histogram jpype cx_freeze rasterio gdal grpcio cvxpy regex ets jupyter pywin32 cairocffi pycairo freetypepy pillow pymol-open-source pymatgen lxml netpbmfile scipy bitarray gmpy cellprofiler xgboost multidict zfec astropy fiax minepy recordclass sqlalchemy yarl zeroc-ice pydantic fontools ujson glfw pycuda python-igraph zipline yappi cobra pandas ta-lib spacy sounddevice reportlab partseg zope.interface aggdraw mercurial cheetah tifffile msvc_runtime pymssql freesasa pygit2 gevent pygresql leidenalg imimuit openpiv cvxcanon cytoolz lsqfit intbitset vispy tinyarray pyrsistent xylib-py annoy numcodecs coverage greenlet cffi mkl_random mkl_fft lfdfiles oiffile fcsfiles matplotlib jcc hmmlearn pyopencl fiona blosc blis moderngl dipy assimulo pyproj

Algumas versões descontinuadas de bibliotecas, em especial do OpenCV podem ser encontradas no site Unnoficial Windows

Binaries for Python Extensions Packages, disponível em: https://www.lfd.uci.edu/~gohlke/pythonlibs/

COMANDOS BÁSICOS DO OPENCV

Enquanto estivermos trabalhando diretamente sobre a biblioteca OpenCV, você notará que grande parte dos processos segue uma mesma sequência lógica de ações.

Sendo assim, basicamente podemos dizer em outras palavras que muito do que se faz por meio da biblioteca OpenCV envolve importar as bibliotecas que serão utilizadas em conjunto, importar uma mídia a partir de uma fonte (seja ela um arquivo de imagem, um vídeo ou até mesmo um dispositivo de entrada como uma webcam), a partir daí aplicar as devidas funções parametrizando as mesmas de modo a obter o efeito desejado, realizando também testes de parâmetros para por fim gerar a saída esperada.

```
1   # Importação da biblioteca
2   import cv2
3
4   # Lendo uma imagem
5   imagem = cv2.imread('nome_do_arquivo.jpg')
6
7   # Descobrindo a largura da imagem
8   print(imagem.shape[1])
9
10  # Descobrindo a altura da imagem
11  print(imagem.shape[0])
12
13  # Exibindo em tela a imagem
14  cv2.imshow('nome da janela', imagem)
15
16  # Salvando a imagem em disco
17  cv2.imwrite('nova_imagem.jpg', imagem)
18
```

Ajuda Inline

Uma vez carregadas as bibliotecas, módulos e pacotes que utilizaremos ao longo de nosso código, é possível (dependendo de sua IDE) consultar um tópico de ajuda no momento em que chamamos alguma função.

Para isso, basta a qualquer momento digitar uma vírgula no espaço dedicado aos parâmetros da função para que seja mostrado quais parâmetros podem ser definidos assim como uma breve explicação das funcionalidades dos mesmos.

```
cv2.imread(,)
```

```
cv2.imread

imread(filename[, flags]) -> retval
.   @brief Loads an image from a file.
.
.   @anchor imread
.
.   The function imread loads an image from the specified file and returns it. If the image cannot be
.   read (because of missing file, improper permissions, unsupported or invalid format), the function
.   returns an empty matrix ( Mat::data==NULL ).

.   Currently, the following file formats are supported:
.
.   -   Windows bitmaps - \*.bmp, \*.dib (always supported)
.   -   JPEG files - \*.jpeg, \*.jpg, \*.jpe (see the *Note* section)
.   -   JPEG 2000 files - \*.jp2 (see the *Note* section)
.   -   Portable Network Graphics - \*.png (see the *Note* section)
.   -   WebP - \*.webp (see the *Note* section)
.   -   Portable image format - \*.pbm, \*.pgm, \*.ppm \*.pxm, \*.pnm (always supported)
.   -   PFM files - \*.pfm (see the *Note* section)
.   -   Sun rasters - \*.sr, \*.ras (always supported)
.   -   TIFF files - \*.tiff, \*.tif (see the *Note* section)
.   -   OpenEXR Image files - \*.exr (see the *Note* section)
.   -   Radiance HDR - \*.hdr, \*.pic (always supported)
.   -   Raster and Vector geospatial data supported by GDAL (see the *Note* section)
.
.   @note
.   -   The function determines the type of an image by the content, not by the file extension.
.   -   In the case of color images, the decoded images will have the channels stored in **B G R** order.
.   -   When using IMREAD_GRAYSCALE, the codec's internal grayscale conversion will be used, if available.
.       Results may differ to the output of cvtColor()
.   -   On Microsoft Windows\* OS and MacOSX\*, the codecs shipped with an OpenCV image (libjpeg,
.       libpng, libtiff, and libjasper) are used by default. So, OpenCV can always read JPEGs, PNGs,
.       and TIFFs. On MacOSX, there is also an option to use native MacOSX image readers. But beware
.       that currently these native image loaders give images with different pixel values because of
.       the color management embedded into MacOSX.
.   -   On Linux\*, BSD flavors and other Unix-like open-source operating systems, OpenCV looks for
.       codecs supplied with an OS image. Install the relevant packages (do not forget the development
.       files, for example, "libjpeg-dev", in Debian\* and Ubuntu\*) to get the codec support or turn
.       on the OPENCV_BUILD_3RDPARTY_LIBS flag in CMake.
.   -   In the case you set *WITH_GDAL* flag to true in CMake and @ref IMREAD_LOAD_GDAL to load the image,
.       then the [GDAL](http://www.gdal.org) driver will be used in order to decode the image, supporting
.       the following formats: [Raster](http://www.gdal.org/formats_list.html),
.       [Vector](http://www.gdal.org/ogr_formats.html).
.   -   If EXIF information are embedded in the image file, the EXIF orientation will be taken into account
.       and thus the image will be rotated accordingly except if the flag @ref IMREAD_IGNORE_ORIENTATION is
passed.
.   -   Use the IMREAD_UNCHANGED flag to keep the floating point values from PFM image.
.   -   By default number of pixels must be less than 2^30. Limit can be set using system
.       variable OPENCV_IO_MAX_IMAGE_PIXELS
```

Por exemplo, ao chamar a função imread( ) da biblioteca OpenCV, é possível ver um tópico flutuante com a descrição da função, possíveis parâmetros e sua descrição e algumas notas importantes.

PROCESSAMENTO DE IMAGENS

Dadas as devidas introduções, vamos diretamente para a prática, inicialmente explorando os recursos mais básicos da biblioteca OpenCV entendendo suas aplicações e efeitos práticos.

Histograma de uma imagem

Um dos primeiros tópicos que abordaremos de forma prática é o de geração de histograma de uma imagem.

Caso você nunca teve nenhum contato com processamento de imagens não se preocupe pois o que será necessário ser entendido será contextualizado e aplicado de forma gradual e prática.

À medida que formos avançando entre os exemplos não nos ateremos mais a explicar repetidas vezes os mesmos conceitos iniciais.

Vez ou outra retomaremos certos pontos quando necessário, mas de modo geral, a medida em que progredimos em nossos estudos iremos cada vez mais direto ao ponto.

Basicamente, um histograma será uma representação visual das características de formação de uma imagem no que diz respeito

a sua configuração de cores primárias.

Em outras palavras, um histograma é uma ferramenta de interpretação onde poderemos ver em forma de gráfico parâmetros de uma imagem como a distribuição de suas cores em sua janela de brilho, uma vez que toda imagem colorida basicamente é formada pelo padrão RGB (sigla para red, green e blue, em tradução livre, vermelho, verde e azul), pois como dito anteriormente, a imagem é formada por diferentes combinações destas cores em uma matriz de pixels.

Equalização de Histograma

Se você é um amante de música, certamente já deve ter dedicado um tempo a entender as diferentes possíveis equalizações que podem serem aplicadas sobre o áudio para realçar certas frequências, que no caso de uma música, estão distribuídas em ondas que vão do grave ao agudo, deixando em destaque a voz ou alguns instrumentos de acordo com sua preferência.

Profissionais que trabalham com fotografia e/ou produção de vídeo também tem como parte de sua rotina o tratamento dessas mídias no que diz respeito a sua equalização, porém quando estamos falando de imagens, a equalização das mesmas está diretamente ligada com fatores como superexposição ou subexposição em relação a luz aplicada sobre a imagem e suas nuances de contraste, corrigindo ou realçando certas características da imagem.

Nessa linha de raciocínio, da mesma forma que equalizamos uma música para deixa-la mais de nosso agrado, um tratamento bastante comum aplicado sobre imagens é a equalização das mesmas por meio da interpretação de seus histogramas, garantindo assim uma correção adequada a imagem no seu pós-processamento.

De forma muito parecida com as barras de equalização de frequências de uma música, os gráficos de histograma nos representarão a real distribuição de cores em seus respectivos canais assim como a distribuição das mesmas de acordo com a exposição em seu espectro.

Em imagens, o padrão que se busca sempre é o equilíbrio destes fatores para que se tenha nitidez sob todos os aspectos da imagem em sua composição. Claro que, neste caso, estamos falando do tratamento corretivo de uma imagem, não levando em consideração a manipulação artística que também pode ser aplicada sobre a mesma.

Partindo para a prática, vamos criar nossas primeiras rotinas de importação das bibliotecas, módulos e pacotes necessários, assim como da mídia a ser processada e tratada.

Me refiro como rotina pois você notará que de fato muitas das aplicações que codificaremos segue uma sequência lógica muito semelhante, alterando basicamente funções e parâmetros aplicados as mesmas com seus respectivos testes.

Pois bem, sem mais delongas, vamos à prática!!!

```
1  import cv2
2  import numpy as np
3  import matplotlib.pyplot as plt
4
```

Todo processo se inicia com a importação das bibliotecas, módulos e pacotes necessários. Nesse caso, importamos a biblioteca OpenCV por meio do comando import cv2, da mesma forma importamos a biblioteca Numpy e a referenciamos como np.

Por fim, importamos apenas o módulo de plotagem da biblioteca MatPlotLib a referenciando como plt.

```
1  imagem = cv2.imread('stephenh.jpg', 1)
2
```

Na sequência criamos uma variável de nome imagem, que por sua vez recebe como atributo a função cv2.imread() da biblioteca opencv, como parâmetro passamos para a mesma em forma de string o nome da imagem com sua respectiva extensão.

O segundo parâmetro é opcional dependendo do contexto, definindo manualmente o mesmo como 1 estamos lendo a imagem em seus 3 canais de cor, definindo como 0 esse parâmetro, estaremos lendo a imagem em preto e branco.

```
plt.imshow(imagem)

<matplotlib.image.AxesImage at 0x7f58aabb9278>
```

Por meio da função plt.imshow() da biblioteca matplotlib, passando nossa variável imagem como parâmetro, teremos a plotagem da imagem em questão.

```
plt.hist(imagem.ravel(), 256, [0,256])
plt.show()
```

Para gerar um histograma geral, temos de combinar duas funções, a primeira delas a função plt.hist.( ) da biblioteca matplotlib, que como parâmetros tem nossa variável imagem chamando a função ravel(), assim como os respectivos valores das distribuições de canais de cor.

Dessa forma temos um gráfico representando a distribuição das cores em pixels dentro de um intervalo personalizado.

```
1  imagem2 = cv2.imread('stephenh.jpg')
2
3  azul, verde, vermelho = cv2.split(imagem2)
4
```

Para gerarmos histogramas específicos para cada canal de cor, podemos usar da função split() da biblioteca cv2, aplicando a mesma sobre a variável imagem2, separando os dados dos canais nas variáveis azul, verde e vermelho, respectivamente.

```
 1  fig = plt.figure(figsize=(20,5))
 2
 3  ax1 = fig.add_subplot(131)
 4  ax1.hist(azul.ravel(), 256, [0,256])
 5  plt.title('Histograma da cor Azul')
 6
 7  ax2 = fig.add_subplot(132)
 8  ax2.hist(verde.ravel(), 256, [0,256])
 9  plt.title('Histograma da cor Verde')
10
11  ax3 = fig.add_subplot(133)
12  ax3.hist(vermelho.ravel(), 256, [0,256])
13  plt.title('Histograma da cor Vermelho')
14
15  plt.show()
16
```

Uma vez que temos canais independentes mapeados, podemos criar uma representação visual um pouco mais robusta para gerar e visualizar tais dados de histograma em forma de gráfico.

Para isso, inicialmente criamos uma variável de nome fig que recebe a função plt.figure( ), especificando por meio de seus parâmetros que as imagens plotadas terão o tamanho de 20x5.

Em seguida criamos uma variável de nome ax1 que recebe a função fig.add_subplot(), uma vez que plotaremos os gráficos para comparação, parametrizando a mesma com o valor 131, que neste contexto significa uma linha, 3 colunas, referente a primeira imagem (primeiro canal de cor, que nesse caso é o azul).

Na sequência ax1 chama a função hist(), parametrizada com azul.ravel(), com a mesma configuração de intervalo de cores de nosso primeiro exemplo.

Por fim, também é atribuído uma legenda em forma de título para a figura por meio da função plt.title( ), passando em forma de string a mensagem em questão.

O mesmo processo é repetido para os outros dois canais de cor, alterando, claro, a variável que chama a função ravel() e a legenda.

Por meio da função plt.show() podemos ver o resultado.

Eis que temos os três histogramas gerados, sendo o da esquerda referente ao canal azul, o do meio ao canal verde e o da direita o canal vermelho.

Agora temos condições de realizar as devidas comparações e entender a composição da imagem no que diz respeito a sua distribuição de cores.

```
imagem3 = cv2.imread('stephenh.jpg', 0)

imagem_eq_hist = cv2.equalizeHist(imagem3)

```

Partindo para a chamada equalização de histograma, inicialmente criamos uma variável de nome imagem3 que recebe a mesma imagem dos exemplos anteriores, porém, agora carregada em preto e branco.

Também criamos a variável imagem_eq_hist que por sua vez chama a função cv2.equalizeHist(), parametrizada com imagem3.

```
fig = plt.figure(figsize=(20,5))
ax1 = fig.add_subplot(121)
plt.imshow(imagem3, cmap=plt.cm.gray)

ax2 = fig.add_subplot(122)
plt.imshow(imagem_eq_hist, cmap=plt.cm.gray)

plt.show()

```

De forma parecida como realizamos a plotagem anterior, agora apenas alteramos alguns parâmetros das mesmas funções usadas anteriormente.

Inicialmente fig.add_subplot() agora possui valor 121 e 122 respectivamente, referentes a uma linha, duas colunas e duas imagens que plotaremos.

Da mesma forma plt.imshow() agora leva o parâmetro cmap=plt.cm.gray, em outras palavras, não só estamos carregando a imagem em preto e branco mas a convertendo para tal para que seus valores de canais sejam escalonados em escalas de cinza padronizados, evitando erros no processo de plotagem.

Por fim, repare que para ax1 teremos a plotagem de imagem3, para ax2 teremos a plotagem de imagem_eq_hist.

Como esperado, temos as devidas plotagens, à esquerda, a imagem em sua forma natural de preto e branco, à direita, a imagem em preto e branco gerada pela função equalizadora.

```
fig = plt.figure(figsize=(20,5))
ax1 = fig.add_subplot(121)
plt.hist(imagem3.ravel(), 256, [0,256])

ax2 = fig.add_subplot(122)
plt.hist(imagem_eq_hist.ravel(), 256, [0,256])

plt.show()

```

Nos mesmos moldes de antes, por meio das funções plt.hist() e ravel() vamos gerar os histogramas.

Finalmente temos os gráficos de modo que podemos realizar nossas interpretações quanto a equalização, onde de modo similar a

uma equalização de uma música, aqui temos a nítida distribuição de tons de cinza claros, médios e escuros que compõe a imagem.

TRANSFORMAÇÕES GEOMÉTRICAS EM IMAGENS

Avançando alguns passos em cima das funções básicas da biblioteca OpenCV, podemos praticar algumas possíveis transformações básicas que podem ser aplicadas sobre uma imagem.

Lembrando que estas funções por sua vez são bastante básicas, na prática usadas quando ao ler alguma determinada imagem, como pré-processamento da mesma temos de realizar transformações geométricas para normalização.

Toda e qualquer transformação mais avançada deve ser feita a partir de um editor dedicado a este propósito.

```
1  import cv2
2  import numpy as np
3  import matplotlib.pyplot as plt
4
```

Como de costume, todo processo se inicia com as devidas importações das bibliotecas, módulos e pacotes que serão usados ao longo de nosso código.

Neste caso, novamente importamos as bibliotecas OpenCV e Numpy, assim como o módulo PyPlot da biblioteca MatPlotLib.

```
1  imagem = cv2.imread('einstein.jpg')
2
```

Na sequência criamos a variável imagem que recebe como atributo para si a função imread( ), parametrizada com o nome do arquivo de imagem, neste caso, ‘einstein.jpg’.

```
plt.imshow(imagem)

```

```
<matplotlib.image.AxesImage at 0x7f961db407b8>
```

Por meio da função imshow( ) da biblioteca matplotlib podemos visualizar a imagem a partir de sua variável.

```
linhas, colunas = imagem.shape
print(linhas)
print(colunas)

```

```
550
433
```

Para visualizarmos, de fato o tamanho desta imagem, criamos duas variáveis de nomes linhas e colunas ,respectivamente, que recebem como atributo o método imagem.shape.

O que este método fará é justamente ler o arquivo em suas configurações de pixels horizontais e verticais e guardar esses dados nas respectivas variáveis.

Por meio da função print( ), parametrizando a mesma com as variáveis em questão podemos ver as dimensões reais da imagem, neste caso 550 pixels de altura por 433 pixels de largura.

Rotacionando uma Imagem

```
matriz_rotacao = cv2.getRotationMatrix2D((linhas/2, colunas/2),
                                         45,
                                         1)

imagem_rotacionada = cv2.warpAffine(imagem,
                                    matriz_rotacao,
                                    (colunas, linhas))
```

Para rotacionar uma imagem sobre sei eixo central, temos de partir do princípio de que este é um objeto em duas dimensões, logo, temos de usar da função específica para este tipo de objeto.

Sendo assim, inicialmente criamos uma variável de nome matriz_rotacao, que recebe como atributo a função do opencv getRotationMatrix2D( ), que por sua vez irá gerar uma matriz numérica que permitirá mapear cada pixel e seu deslocamento em rotação sem deformar a imagem e suas características visuais.

Como parâmetros da mesma primeiro definimos que a matriz será gerada a partir dos dados de linhas e colunas divididos por 2, assim usando como referência o ponto médio horizontal e longitudinal como referência para torração.

Também é estipulado um segundo parâmetro com o número 45, este por sua vez diz respeito ao grau de rotação da imagem (numa escala de 0 a 360º), e por fim o terceiro parâmetro “obrigatório” aqui definido como 1 significa que queremos a imagem em seu tamanho real.

Em seguida é criada uma nova variável de nome imagem_rotacionada, que chama a função warpAffine( ), essa função internamente faz a combinação do formato original que a imagem possuía (tamanho e altura originais) e insere dentro deste espaço o conteúdo da imagem rotacionada, mantendo assim a proporção original, com o novo conteúdo rotacionado, preenchendo os espaços vazios com pixels pretos.

Para conseguir este resultado é repassado para a mesma como parâmetros os dados de imagem, imagem_rotacionada e os valores de colunas e linhas, respectivamente.

```
fig = plt.figure(figsize=(10,8))
plt.imshow(imagem_rotacionada, cmap=plt.cm.gray)

<matplotlib.image.AxesImage at 0x7f961faa23c8>
```

Plotando novamente a imagem, agora com base em nossa variável imagem_rotacionada, podemos ver de fato que a imagem original está rotacionada em 45º dentro do tamanho e formato original.

Translação de uma Imagem

Uma das transformações básicas um tanto quanto comuns de ser usada é a chamada translação de imagem, onde o que ocorre é o simples deslocamento de uma imagem ao longo de um eixo, a partir de sua forma original.

O processo é parecido com o feito no exemplo anterior, porém, com uma função dedicada a este tipo de transformação.

```
1  matriz_translacao = np.float32([[1,0,100],[0,1,100]])
2
3  imagem_deslocada = cv2.warpAffine(imagem,
4                                    matriz_translacao,
5                                    (colunas, linhas))
6
```

Para isso, inicialmente criamos uma variável de nome matriz_translacao, que chama a função float32 da biblioteca Numpy, Aqui parametrizamos a mesma criando uma array com os respectivos dados [1,0,100],[0,1,100], sendo a primeira referente ao eixo X da imagem, e a segunda referente ao eixo Y, deslocando 100 pixels em cada eixo de acordo com o terceiro valor definido.

Exatamente da mesma forma que no exemplo anterior, criamos uma variável imagem_deslocada que usa da função warpAffine( ) para sobrepor os dados da imagem original em nossa matriz de translação.

```
fig = plt.figure(figsize=(10,8))
plt.imshow(imagem_deslocada, cmap=plt.cm.gray)

<matplotlib.image.AxesImage at 0x7f961f586550>
```

Novamente plotamos a imagem ,agora mostrando o conteúdo de imagem_deslocada, que por sua vez possui o tamanho original de formato, com seu conteúdo visual deslocado 100 pixels na horizontal e 100 pixels na vertical.

Escalonando uma Imagem

Outra aplicação bastante comum é o escalonamento de uma determinada imagem, onde alteramos o tamanho da mesma sem perda de suas proporções originais de equivalência entre largura e altura.

```
1  imagem_escalonada = cv2.resize(imagem,
2                                 None,
3                                 fx = 0.4,
4                                 fy = 0.4,
5                                 interpolation = cv2.INTER_CUBIC)
6
```

O processo de escalonamento é bastante simples pois também conta com uma função dedicada a este propósito, bastando instanciar e parametrizar a mesma.

Dessa forma, inicialmente criamos uma variável de nome imagem_escalonada que recebe como parâmetro a função resize( ) da biblioteca opencv.

Como parâmetros da mesma, em primeiro lugar passamos o conteúdo de nossa variável imagem, o segundo parâmetro, caso definido, irá gerar uma nova imagem com as novas proporções, mantendo este parâmetro como None, o escalonamento será aplicado sobre a própria imagem original.

Em seguida fx e fy definem o fator de escalonamento horizontal e vertical, respectivamente. Lembrando que aqui estes valores devem ser definidos como iguais visando manter a proporção visual da imagem, não distorcendo a mesma.

Por fim é preciso especificar o método de interpolação, podendo ser INTER_CUBIC, INTER_LINEAR ou INTER_AREA, sendo destes, o que traz melhor eficiência para escalonamentos sem perda de qualidade, o INTER_BICUBIC.

```
1  cv2.imshow('Imagem Original', imagem )
2  cv2.imshow('Imagem Escalonada', imagem_escalonada)
3  cv2.waitKey(0)
4  cv2.destroyAllWindows()
5
```

Para fins de exemplo, por meio da função imshow( ) vamos exibir tanto a imagem original quanto a escalonada.

Neste caso também temos duas linhas de código adicionais, onde executamos as funções WaitKey( ) e destroyAllWindows, para ao pressionar qualquer botão, fechar as janelas das duas imagens.

Como esperado, temos as projeções das duas imagens, em escalas diferentes, sem deformidades ou perda de qualidade.

```
linhas, colunas = imagem_escalonada.shape
print(linhas)
print(colunas)

```

```
220
173
```

Apenas para fins de comparação, novamente aplicando o método .shape, agora em nossa variável imagem_escalonada, temos as dimensões de 220 pixels de altura por 173 de largura, enquanto nossa imagem original tem dimensões de 550 pixels de altura por 433 pixels de largura.

```
largura = imagem.shape[1]
altura = imagem.shape[0]
proporcao = float(altura/largura)
largura_nova = 200
altura_nova = int(largura_nova*proporcao)

fig = plt.figure(figsize=(8,5))
plt.imshow(imagem, cmap=plt.cm.gray)

```

```
matplotlib.image.AxesImage at 0x7fb0ab672198>
```

Apenas a nível de curiosidade, nada impede que façamos o escalonamento de uma imagem usando de regras matemáticas simples.

Como mostrado na imagem acima, podemos criar variáveis específicas para altura e largura da imagem original, dividindo os valores de uma por outra obtendo um dado da proporção da imagem.

Da mesma forma criamos uma variável de nome largura_nova que define manualmente qual será o valor de referência para o escalonamento baseado em sua largura.

Por fim definimos uma variável de nome altura_nova que realiza a multiplicação entre os valores de proporcao e largura_nova.

Recortando uma Imagem

Em certas situações pode ser necessário que se faça uso apenas de uma parte de uma imagem, e isto pode ser feito diretamente sobre a imagem original, bastando realizar uma delimitação de seu tamanho e posição diretamente em sua matriz.

```
1  recorte = imagem[100:200, 100:200]
2
3  fig = plt.figure(figsize=(8,5))
4  plt.imshow(recorte, cmap=plt.cm.gray)
5
```

```
matplotlib.image.AxesImage at 0x7fb0ab713908>
```

Para isso simplesmente criamos uma variável de nome recorte que recebe como atributo para si parte da imagem original, referenciadas as posições da linha 100 até a 200, da mesma forma que da coluna 100 até a 200.

Plotando a imagem podemos perceber que o conteúdo de nossa variável recorte trata-se do pedaço da imagem correspondente aos valores acima.

Espelhando uma Imagem

Por fim, mas não menos importante, em certos casos pode ser necessário o uso de uma determinada imagem em sua forma espelhada, e isso pode ser feito diretamente com a função cv2.flip( ) da biblioteca opencv.

```
imagem = cv2.imread('einstein.jpg', 0)

flip_horizontal = cv2.flip(imagem, 1)

fig = plt.figure(figsize=(8,5))
plt.imshow(flip_horizontal, cmap=plt.cm.gray)

<matplotlib.image.AxesImage at 0x7fb0ab6674a8>
```

Novamente trabalhando sobre a mesma imagem dos exemplos anteriores, o que temos de fazer é simplesmente chamar a função flip( ) da biblioteca opencv.

Para isso, criamos uma variável de nome flip_horizontal que chama a função cv2.flip( ) passando como parâmetros para a

mesma nossa imagem e um segundo parâmetro chamado flipCode que quando definido como 1 inverte a imagem em seu eixo horizontal, mantendo todas as proporções da imagem original.

Plotando a imagem temos, como esperado, nossa imagem espelhada em seu eixo horizontal.

```
imagem = cv2.imread('einstein.jpg', 0)

flip_vertical = cv2.flip(imagem, 0)

fig = plt.figure(figsize=(8,5))
plt.imshow(flip_vertical, cmap=plt.cm.gray)
```

```
<matplotlib.image.AxesImage at 0x7fb0ab5cb278>
```

O processo de espelhamento vertical se dá exatamente da mesma forma, apenas alterando o nome de nossa variável para flip_vertical e alterando o flipCode da função cv2.flip( ) de 1 para 0 nesse caso.

Plotando a imagem temos o resultado esperado.

OPERAÇÕES ARITMÉTICAS EM IMAGENS

Uma vez entendido que imagens são objetos transliterados para matrizes numéricas, podemos efetuar algumas operações aritméticas sobre as mesmas, de modo que funções matemáticas sejam aplicadas sobre as configurações dos pixels, o que refletirá em alterações na própria imagem.

Por essa afirmação também é possível presumir que se tratando de dados de imagens aplicaremos sobre os mesmos operações básicas de matemática, pois operações complexas iriam descaracterizar a imagem original.

Soma de Imagens

Começando pelo básico, quando estamos falando de soma em imagens estamos na verdade realizando a soma dos valores dos pixels em sua matriz, dessa forma, podemos por exemplo, somar duas imagens diferentes para obter uma terceira que visualmente é uma mescla das imagens originais.

```
1  import cv2
2  import numpy as np
3  import matplotlib.pyplot as plt
4
```

Como sempre, todo processo se inicia com as importações das bibliotecas, módulos e pacotes que usaremos ao longo de nosso código.

```
1  imagem_1 = cv2.imread('lamborghini_1.jpg', 1)
2  imagem_2 = cv2.imread('lamborghini_2.jpg', 1)
3
```

Na sequência realizamos as importações das imagens que iremos utilizar. Para isso, criamos as variáveis imagem_1 e imagem_2 que por meio da função cv2.imread( ) parametrizada com os nomes dos arquivos irá ler os mesmos em sua forma, escala e padrões de cor originais, atribuindo seus dados as respectivas variáveis.

```
fig = plt.figure(figsize=(20,5))
ax1 = fig.add_subplot(121)
plt.imshow(imagem_1)
ax2 = fig.add_subplot(122)
plt.imshow(imagem_2)
plt.show()
```

Plotando as imagens podemos ver que se tratam de duas imagens de duas Lamborghinis, imagens estas bastante diferentes na perspectiva de como mostram o carro em si.

```
soma = cv2.add(imagem_1, imagem_2)

```

Para realizar a soma destas imagens basicamente usaremos da função add( ) da própria biblioteca opencv. Sendo assim, criamos uma variável de nome soma que chama a função cv2.add( ) parametrizando a mesma com as variáveis que contém nossas imagens.

```
fig = plt.figure(figsize=(10,8))
plt.imshow(soma,)

<matplotlib.image.AxesImage at 0x7f5b854ef0b8>
```

Plotando a nossa variável soma temos o resultado em tela. Note que a imagem gerada para soma nada mais é do que a sobreposição das imagens originais, uma vez que houve uma soma direta de cada pixel da matriz de cada imagem.

Repare que o resultado, dependendo das imagens originais, pode gerar sobreposições boas ou ruins, no sentido de que a imagem da mescla pode ter regiões inteiras sub ou sobre expostas, perdendo características importantes na imagem nova gerada.

Subtração de Imagens

Para realizar a subtração entre duas imagens, realizaremos praticamente o mesmo processo que no exemplo anterior, porém dessa vez fazendo o uso da função subtract( ).

```
1  imagem_1 = cv2.imread('lamborghini_1.jpg', 1)
2  imagem_2 = cv2.imread('lamborghini_2.jpg', 1)
3
4  subtracao = cv2.subtract(imagem_1, imagem_2)
5
```

Da mesma forma que no exemplo anterior, criamos variáveis dedicadas a armazenar os dados das imagens lidas por meio da função cv2.imread( ).

Em seguida criamos uma variável de nome subtração que por sua vez chama a função cv2.subtract( ), também parametrizando a mesma com as variáveis em questão.

```
fig = plt.figure(figsize=(10,8))
plt.imshow(subtracao)

<matplotlib.image.AxesImage at 0x7fdd5451e828>
```

Plotando a imagem gerada a partir da subtração de imagem_1 e imagem_2,o resultado, como esperado, é uma imagem com muito mais contrastes se comparado com a imagem de soma, pois aqui, subtraindo certas partes elas podem se anular, deixando espaços “em branco”, que serão preenchidos com a cor preta.

```
fig = plt.figure(figsize=(10,8))
plt.imshow(subtracao, cmap=plt.cm.gray)
```

```
<matplotlib.image.AxesImage at 0x7f5b84e18470>
```

Podemos salientar ainda mais o contraste da imagem plotando a mesma em escala de cinza, por meio do parâmetro cmap=plt.cm.gray na função cv2.imshow( ).

Sobreposição de Imagens

Visando contornar esse problema de sobreposições que descaracterizam as imagens, foi criada a função addWeighted( ) da biblioteca opencv. Função esta que permite o fácil ajuste de quanto uma imagem irá sobrepor a outra, para que tenhamos resultados mais interessantes.

```
1  imagem_1 = cv2.imread('lamborghini_1.jpg', 1)
2  imagem_2 = cv2.imread('lamborghini_2.jpg', 1)
3
4  mescla = cv2.addWeighted(imagem_1, 0.8, imagem_2, 0.8, 0)
5
```

Exatamente da mesma forma como fizemos nos exemplos anteriores, temos duas variáveis com nossas respectivas imagens.

A partir disto criamos uma nova variável de nome mescla, que chama a função cv2.addWeighted( ), parametrizando a mesma com nossas variáveis imagem_1 e imagem_2, também passando como parâmetro na sequência das mesmas os valores 0.8, que em outras palavras nos significa o percentual de quanto uma imagem irá sobrepor a outra.

```
fig = plt.figure(figsize=(10,8))
plt.imshow(mescla)

<matplotlib.image.AxesImage at 0x7f5b85576898>
```

Apenas como exemplo inicial, definindo o percentual de sobreposição em 80%, o resultado como esperado é muito semelhante ao já feito anteriormente na soma normal das imagens.

```
mescla = cv2.addWeighted(imagem_1, 0.2, imagem_2, 0.8, 0)
fig = plt.figure(figsize=(10,8))
plt.imshow(mescla)

```

```
<matplotlib.image.AxesImage at 0x7f5b85040e48>
```

Alterando as proporções em 0.2 e 0.8, respectivamente para as variáveis imagem_1 e imagem_2, podemos ver que agora uma se sobrepõe a outra, gerando um efeito mais interessante.

Neste caso, as devidas sobreposições foram realizadas a nível dos pixels das matrizes, em uma proporção onde a imagem_1 tem seu conteúdo como 20% da composição da imagem final, assim como imagem_2 representa os 80% restantes da mesma.

```
mescla = cv2.addWeighted(imagem_1, 0.7, imagem_2, 0.3, 0)
fig = plt.figure(figsize=(10,8))
plt.imshow(mescla)
```

```
<matplotlib.image.AxesImage at 0x7f5b84f62a58>
```

Numa proporção de 70% / 30%, trazemos o conteúdo de imagem_1 em evidência enquanto o conteúdo de imagem_2 fica em segundo plano.

OPERAÇÕES MORFOLÓGICAS

Uma vez entendido que podemos fazer uma exploração a partir de uma ou mais imagens, realizando um pré-tratamento das mesmas, podemos avançar mais um passo entendendo que por meio das ferramentas certas podemos alterar características visuais de nossas imagens, tanto para realce puramente estético quanto para de fato realçar características que as tornem melhor manipuláveis por meio de suas matrizes.

Elementos Estruturantes

Para início de entendimento, raciocine que quando estamos falando dos elementos que compõe visualmente uma imagem, estamos falando de elementos estruturantes como bordas, contornos, diferenças de densidade, características únicas se comparado com outras regiões da mesma imagem, etc...

No intuito de tornar uma imagem melhor para processamento sobre a mesma podemos reconhecer, extrair e realçar características que a farão única, mapeada de modo que podemos a partir da mesma executar uma série de operações lógicas e/ou aritméticas.

A nível de matriz, o que faremos basicamente é criar uma matriz inicial que será sobreposta sobre o mapa de características de uma imagem, facilitando o reconhecimento de padrões e características de nossa imagem.

```
1  import cv2
2  import numpy as np
3  import matplotlib.pyplot as plt
4
```

Como de costume, todo processo se inicia com as devidas importações de bibliotecas, módulos e pacotes.

```
1  imagem8 = cv2.imread('placa.jpg', 1)
2
```

Em seguida, apenas como exemplo, criamos uma variável de nome imagem8 que recebe os dados de nossa imagem placa.jpg por meio da função cv2.imread( ).

```
[ ]  1  plt.imshow(imagem8)
     2
```

Plotando a imagem carregada podemos ver suas características visuais.

```
1  elem_rect = cv2.getStructuringElement(cv2.MORPH_RECT, (8,8))
2
3  print(elem_rect)
4
```

Na sequência partimos para a criação do elemento estruturante, nesse caso, criamos uma variável de nome elem_rect que por sua vez chama a função cv2.getStructuringElement( ).

Como parâmetro desta função basicamente temos de passar no mínimo duas referências, o formato da matriz-elemento estruturante e seu tamanho. Aqui, como queremos um quadrado para leitura de característica em quadrantes de nossa matriz de imagem, usamos do parâmetro cv2.MORPH_RECT, de tamanho 8x8.

```
3  print(elem_rect)
4

[[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 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1]]
```

Por meio de nossa função print( ) podemos exibir o conteúdo de elem_rect, e como esperado, temos uma matriz de 8 linhas por 8 colunas, com valores padrão pré-definidos.

Estes valores por sua vez serão substituídos pelas características dos quadrantes lidos, gerando novos mapas de características.

Aplicando Erosão

Uma das aplicações mais comumente usadas sobre imagens é a chamada erosão das mesmas. A técnica de erosão visa realçar características como bordas e contornos, que em muitos casos podem ser tão sutis a ponto de serem mal detectadas. Sendo assim, quando erodimos uma imagem basicamente estamos detectando e aumentando as linhas de desenho de uma imagem.

```
imgprocessada = cv2.erode(imagem8, elem_rect, iterations = 1)

```

Partindo para a prática, inicialmente criamos uma variável de nome imagemprocessada, que chama a função cv2.erode( ). Essa função por sua vez deve receber três parâmetros, a imagem fonte a ser erodida, o elemento estruturante e o número de vezes a função de erosão será aplicada sobre a imagem.

```
fig = plt.figure(figsize=(10,8))
ax1 = fig.add_subplot(121)
plt.imshow(imagem8)
ax2 = fig.add_subplot(122)
plt.imshow(imgprocessada)
plt.show()

```

Plotando a imagem original e a imagem erodida podemos ver o realce das linhas de desenho.

```
imgprocessada = cv2.erode(imagem8, elem_rect, iterations = 5)
fig = plt.figure(figsize=(10,8))
ax1 = fig.add_subplot(121)
plt.imshow(imagem8)
ax2 = fig.add_subplot(122)
plt.imshow(imgprocessada)
plt.show()
```

Apenas para fins de exemplo, alterando o número de vezes que a função de erosão será aplicada, podemos ver que, como sempre, devemos realizar diversos testes com os valores de parâmetros a fim de encontrar um resultado satisfatório, pois em muitos casos uma parametrização incorreta pode gerar resultados indesejáveis.

```
from skimage import morphology
from skimage import io

img = io.imread('placa.jpg')
eroded_img = morphology.binary_erosion(img)

io.imshow(eroded_img)
io.show()
```

Também apenas a nível de curiosidade, realizando o mesmo processo de erosão por meio da função morphology.binary_erosion( ) da biblioteca SKImagem o resultado obtido é praticamente o mesmo que com as funções da biblioteca OpenCV.

Aplicando Dilatação

De certa forma a chamada dilatação de características de uma imagem pode ser considerada o inverso da erosão, pois seu princípio é realizar a leitura de uma imagem descartando falsos positivos de linhas que podem estar atrapalhando o processamento.

Em nosso exemplo atual, o grande foco são os dígitos identificadores do veículo, sendo as demais linhas de desenho e traços de bordas e contornos não relevantes para o processo.

Sendo assim, em casos específicos como esse, vale a pena realizar também alguns testes aplicando dilatação sobre a imagem e comparando seus resultados com a técnica de erosão.

```
1  elem_elip = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (7,7))
2
3  imgprocessada2 = cv2.dilate(imagem8, elem_elip, iterations = 1)
4
```

Aproveitando o exemplo, vamos inclusive usar de outro elemento estruturante, agora em forma de elipse. Para isso criamos a variável elem_elip que chama a função cv2.getStructuringElement( ), dessa vez passando como parâmetro para a mesma cv2.MORPH_ELLIPSE, no tamanho de 7x7.

Na sequência criamos a variável imagemprocessada2 que chama a função cv2.dilate( ), que da mesma forma como a função cv2.erode( ) deve receber como parâmetros a imagem original, o elemento estruturante e o número de vezes ao qual a função será aplicada sobre a imagem.

```
fig = plt.figure(figsize=(10,8))
ax1 = fig.add_subplot(121)
plt.imshow(imagem8)
ax2 = fig.add_subplot(122)
plt.imshow(imgprocessada2)
plt.show()
```

Plotando lado a lado a imagem original e a imagem dilatada podemos ver que de fato na imagem dilatada todas as linhas sem muita relevância foram suavizadas ao ponto de desaparecerem, mantendo somente em destaque os dígitos identificadores.

```
elem_cruz = cv2.getStructuringElement(cv2.MORPH_CROSS, (7,7))
imgprocessada2 = cv2.dilate(imagem8, elem_cruz, iterations = 1)
```

```
fig = plt.figure(figsize=(10,8))
ax1 = fig.add_subplot(121)
plt.imshow(imagem8)
ax2 = fig.add_subplot(122)
plt.imshow(imgprocessada2)
plt.show()
```

Testando como elemento estruturante o elemento cv2.MORPH_CROSS, ou seja, em forma de cruz, acabamos por ter

um resultado levemente melhor se comparado ao anterior.

Cada tipo de elemento estruturante apresentará resultados peculiares de acordo com as formas da imagem, uns apresentando resultados melhores que outros, sendo assim, uma boa prática é realizar testes com ao menos dois tipos de elementos estruturantes diferentes analisando e comparando seus resultados.

```
imgprocessada3 = cv2.dilate(imagem8, elem_cruz, iterations = 3)
fig = plt.figure(figsize=(10,8))
ax1 = fig.add_subplot(121)
plt.imshow(imagem8)
ax2 = fig.add_subplot(122)
plt.imshow(imgprocessada3)
plt.show()
```

Apenas alterando o número de interações de 1 para 3 acabamos extrapolando, descaracterizando totalmente a imagem.

Podemos deduzir então que, a combinação de elemento estruturante somado ao número de vezes ao qual o processo será aplicado sobre a imagem deve ser alvo de testes para acharmos um resultado bom o suficiente para nosso propósito.

```python
from skimage import morphology
from skimage import io

from google.colab import files
arquivos = files.upload()

img = io.imread('placa.jpg')

dilated_img = morphology.binary_dilation(img)

io.imshow(dilated_img)
io.show()

```

Apenas como exemplo, mesmo processo de dilatação de linhas da imagem, porém realizado por meio da biblioteca SKImage.

```python
fig = plt.figure(figsize=(5,15))
ax1 = fig.add_subplot(311)
plt.imshow(imagem8)
plt.title('Imagem Original')
ax2 = fig.add_subplot(312)
plt.imshow(imgprocessada)
plt.title('Imagem Erodida')
ax2 = fig.add_subplot(313)
plt.imshow(imgprocessada2)
plt.title('Imagem Dilatada')
plt.show()

```

Plotando os resultados, acima temos a imagem original, ao meio temos a imagem erodida e abaixo a imagem dilatada. Apenas como exemplo.

Gerando um Gradiente Morfológico

Gradualmente, a medida que vamos conhecendo certas técnicas de processamento de imagens, podemos também realizar certas combinações entre as mesmas.

O chamado gradiente morfológico, tratando-se da função de tal nome, nada mais é do que a diferença entre as operações de erosão e dilatação de uma certa imagem, de modo a gerar uma nova matriz de características que pode ser aplicada a novas imagens do mesmo contexto.

```
imagem9 = cv2.imread('mao.jpg', 0)

```

```
fig = plt.figure(figsize=(10,8))
ax1 = fig.add_subplot(121)
plt.imshow(imagem9)
plt.show()
```

Visando sair da monotonia, vamos mudar um pouco nosso exemplo, realizando o processo em uma nova imagem.

Dessa vez, uma radiografia de uma mão, imagem gerada por raios-x onde é aplicada uma certa dose de radiação sobre os tecidos, ficando parte dela absorvida em estruturas densas como os ossos, gerando assim uma imagem fidedigna da anatomia para estudos.

A imagem em questão servirá como um bom exemplo pois imagens de radiografias possuem uma excelente margem de contraste entre os tecidos da anatomia, seus contornos e sua projeção quanto ao fundo da imagem.

```
fig = plt.figure(figsize=(10,8))
ax1 = fig.add_subplot(121)
plt.imshow(imagem9)
plt.show()

```

Lendo a imagem sem especificar o parâmetro do número de canais, sejam eles RGB ou escala de cinza, temos a seguinte projeção.

```
elem_elip2 = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (21,21))
elem_elip3 = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (5,5))

print(elem_elip2)
print(elem_elip3)
```

```
[[0 0 0 0 0 0 0 0 0 0 1 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 0 0 0 0 0 0]
 [0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0]
 [0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0]
 [0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0]
 [0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0]
 [0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 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 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 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]
 [0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0]
 [0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0]
 [0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0]
 [0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0]
 [0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0]
 [0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0]]
[[0 0 1 0 0]
 [1 1 1 1 1]
 [1 1 1 1 1]
 [1 1 1 1 1]
 [0 0 1 0 0]]
```

Em seguida criamos dois elementos estruturantes em forma de elipse, um de tamanho 21x21 e outro de tamanho 5x5, atribuídos as respectivas variáveis elem_elip2 e elem_elip3.

```
imgprocessada4 = cv2.morphologyEx(imagem9, cv2.MORPH_GRADIENT, elem_elip3)
imgprocessada5 = cv2.morphologyEx(imagem9, cv2.MORPH_TOPHAT, elem_elip2)
imgprocessada6 = cv2.morphologyEx(imagem9, cv2.MORPH_BLACKHAT, elem_elip2)
```

Na sequência criamos três variáveis, imagemprocessada4, imagemprocessada5 e imagem processada6. Para estas, atribuímos a função cv2.morphologyEx( ), parametrizado com nossa imagem, a técnica de extração de característica morfológica e o elemento estruturante.

Como dito anteriormente, aqui estamos tratando de uma combinação de técnicas, de modo a extrair características tanto de erosão quanto de dilatação para combinar e realçar as mesmas na imagem, visando tornar suas características visíveis melhor representadas.

```
fig = plt.figure(figsize=(8,20))
ax1 = fig.add_subplot(221)
plt.imshow(imagem9)
plt.title('Imagem Original')
ax2 = fig.add_subplot(222)
plt.imshow(imgprocessada4)
plt.title('Gradiente Morfológico')
ax3 = fig.add_subplot(223)
plt.imshow(imgprocessada5)
plt.title('TopHat')
ax4 = fig.add_subplot(224)
plt.imshow(imgprocessada6)
plt.title('BlackHat')
plt.show()

```

Da mesma forma como feito nos exemplos anteriores, vamos plotar as imagens para realizar as devidas comparações visuais.

E assim temos a representação das quatro imagens, no quadrante superior esquerdo temos a imagem original, no quadrante inferior esquerdo a imagem TopHat, que ressalta melhor os contornos dos ossos, no quadrante inferior direito a imagem BlackHat, que ressalta melhor as imagens dos contornos de partes moles (como a pele do paciente).

Por fim, no quadrante superior direito temos a imagem gerada como gradiente morfológico, repare que nesta temos uma bela combinação das imagens TopHat e BlackHat, gerando uma imagem sem perda de características e com nível de detalhe muito superior a imagem original.

```
elem_cruz2 = cv2.getStructuringElement(cv2.MORPH_CROSS, (21,21))
elem_cruz3 = cv2.getStructuringElement(cv2.MORPH_CROSS, (5,5))

imgprocessada4 = cv2.morphologyEx(imagem9, cv2.MORPH_GRADIENT, elem_cruz3)
imgprocessada5 = cv2.morphologyEx(imagem9, cv2.MORPH_TOPHAT, elem_cruz2)
imgprocessada6 = cv2.morphologyEx(imagem9, cv2.MORPH_BLACKHAT, elem_cruz2)

fig = plt.figure(figsize=(20,50))
ax1 = fig.add_subplot(141)
plt.imshow(imagem9)
plt.title('Imagem Original')
ax2 = fig.add_subplot(142)
plt.imshow(imgprocessada4)
plt.title('Gradiente Morfológico')
ax3 = fig.add_subplot(143)
plt.imshow(imgprocessada5)
plt.title('TopHat')
ax4 = fig.add_subplot(144)
plt.imshow(imgprocessada6)
plt.title('BlackHat')
plt.show()
```

Da mesma forma como nos exemplos anteriores, podemos realizar testas alterando nossos elementos estruturantes, para fins de comparação.

Plotando as imagens, nos mesmos moldes do exemplo anterior, na imagem superior esquerda temos a imagem original, inferior esquerda a TopHat, inferior direita a BlackHat e quadrante superior direito a imagem de gradiente morfológico.

Realizando uma breve comparação visual podemos reparar que para esse caso, o uso de elemento estruturante em forma de cruz mostrou resultados inferiores ao modelo anterior, perdendo qualidade principalmente a nível da rugosidade dos tecidos ósseos.

TRATAMENTO DE IMAGENS

Partindo para uma abordagem um pouco diferente, vamos ver alguns possíveis tratamentos de imagens por meio da biblioteca OpenCV e outras.

Lembrando que aqui no escopo de bibliotecas de processamento de imagens teremos à disposição apenas pequenos métodos e funções que podem ser integradas a certos processamentos de visão computacional. Para um real tratamento de imagens as mesmas devem ser processadas por softwares específicos para esses fins.

Filtro de Suavização por Borramento

Para darmos início a prática deste breve capítulo, vamos entender como se dá o processo de aplicação de filtros de suavização de imagem.

Basicamente sempre que estivermos falando em filtros de suavização, estaremos falando em filtros que tentarão reduzir artefatos na imagem como granulações indesejadas.

Se tratando de imagens, é muito comum o aparecimento de ruídos na imagem quando a mesma é obtida em ambientes com pouca luminosidade ou tempo de exposição inadequado.

```
1  import cv2
2  import numpy as np
3  import matplotlib.pyplot as plt
4
```

Como de costume, o processo se inicia com as devidas importações das nossas já conhecidas bibliotecas opencv, numpy e matplotlib.

```
1  imagem6 = cv2.imread('pato.jpg', 1)
2
```

Em seguida criamos a nossa variável imagem6 que recebe como atributo a imagem a qual faremos nossos testes, por meio da função cv2.imread( ).

```
imagemfiltrada = cv2.blur(imagem6, (3,3))

```

Na sequência vamos usar de nosso primeiro filtro de suavização, o mais comum de todos, o de borramento.

Então criamos nossa variável imagemfiltrada que chama a função cv2.blur( ), por sua vez parametrizada com nossa imagem6 e uma matriz 3x3 como ksize, este responsável por gerar um borramento baseado em um pixel em relação a todos seus vizinhos.

```
fig = plt.figure(figsize=(10,5))
ax1 = fig.add_subplot(121)
plt.imshow(imagem6)
plt.title('Imagem Original')
ax2 = fig.add_subplot(122)
plt.imshow(imagemfiltrada)
plt.title('Imagem Filtrada')
plt.show()

```

Plotando as imagens para comparação, a esquerda temos a imagem original, da mesma forma a direita temos a imagem referente a imagemfiltrada.

Visualmente podemos perceber que a imagem original possui um certo nível de ruído que foi de fato suavizado com a aplicação do filtro blur, sem perda de nitidez e detalhe da imagem do pato em si, o foco da imagem.

```
imagemfiltrada = cv2.blur(imagem6, (7,7))
fig = plt.figure(figsize=(10,5))
ax1 = fig.add_subplot(121)
plt.imshow(imagem6)
ax2 = fig.add_subplot(122)
plt.imshow(imagemfiltrada)
plt.show()
```

Alterando o parâmetro ksize da função cv2.blur( ) de 3x3 para 7x7 obtivemos um resultado não satisfatório, uma vez que essa matriz de borramento acabou por borrar todos elementos da imagem, perdendo totalmente a nitidez da mesma.

Como sempre, cada vez que estamos realizando tais processamentos em imagens cabe testar diferentes parâmetros em busca de um resultado ótimo. Os exemplos desse livro se comportam de uma determinada maneira, ao você carregar suas imagens, talvez os mesmos parâmetros gerem efeitos diferentes.

Filtro Gaussiano

Um filtro alternativo ao blur, muito comumente usado como filtro de suavização de uma imagem é o chamado filtro gaussiano. Este, por sua vez, também usa de cálculos sobre matrizes levando em consideração um pixel alvo e suas adjacências, porém, para alguns propósitos se mostra mais eficiente que o filtro blur.

```
imagemfiltrada = cv2.GaussianBlur(imagem6, (5,5), 0)

```

Criamos uma nova variável de nome imagemfiltrada, dessa vez, tendo como atributo a função cv2.GaussianBlur( ).

De forma parecida com o que vimos anteriormente, aqui passamos como parâmetro nossa imagem6, uma matriz e um terceiro parâmetro, aqui definido como 0, definindo que o número de vizinhos a ser considerado deve ser igual nos eixos x e y da imagem.

Este parâmetro também pode, em casos muito específicos, ter valores manuais atribuídos, para considerar a relação dos pixels alvo com seus vizinhos verticais ou horizontais.

```
fig = plt.figure(figsize=(10,5))
ax1 = fig.add_subplot(121)
plt.imshow(imagem6)
plt.title('Imagem Original')
ax2 = fig.add_subplot(122)
plt.imshow(imagemfiltrada)
plt.title('Imagem Filtrada')
plt.show()
```

Como de praxe, plotando as imagens para comparação podemos notar grande diferença entre a imagem original à esquerda e a processada à direita.

Repare que neste caso, a imagem processada, se comparado a imagem original sofreu um borramento muito acentuado, descaracterizando a mesma em sua nitidez, gerando um resultado inferior ao obtido pelo filtro blur.

```
imagemfiltrada = cv2.GaussianBlur(imagem6, (5,5), 3)
fig = plt.figure(figsize=(10,5))
ax1 = fig.add_subplot(121)
plt.imshow(imagem6)
plt.title('Imagem Original')
ax2 = fig.add_subplot(122)
plt.imshow(imagemfiltrada)
plt.title('Imagem Filtrada')
plt.show()
```

Mantendo o mesmo tamanho de matriz de borramento, mas alterando o terceiro parâmetro de 0 para 3, obtemos uma imagem com borramento gerado a partir da relação dos pixels alvo com seus vizinhos verticais, também não gerando uma imagem boa o suficiente para ser levada em consideração.

```
imagemfiltrada = cv2.GaussianBlur(imagem6, (5,5), 2)
fig = plt.figure(figsize=(10,5))
ax1 = fig.add_subplot(121)
plt.imshow(imagem6)
plt.title('Imagem Original')
ax2 = fig.add_subplot(122)
plt.imshow(imagemfiltrada)
plt.title('Imagem Filtrada')
plt.show()
```

Alterando os parâmetros de modo a aplicar a matriz de borramento considerando os vizinhos horizontais aos pixels alvo acabamos por obter um resultado ótimo, onde ao mesmo tempo temos uma considerável redução do nível de ruído da imagem, principalmente em seu fundo), e uma boa margem de nitidez no objeto foco da imagem, neste caso, um pato.

A relação que deve ser aplicada nestes casos se dará de acordo com as características da imagem, você deve levar em consideração se o objeto alvo da imagem é algo mais vertical ou mais horizontal na forma como o mesmo está apresentado na imagem, definindo assim o tipo e forma de matriz de borramento a ser aplicada.

```python
from PIL import Image
from PIL import ImageFilter

imagem6 = Image.open('pato.jpg')

blur_img = imagem6.filter(ImageFilter.GaussianBlur(2))
blur_img.show()

```

Resultados muito parecidos podem ser obtidos por meio da função filter( ), combinada com ImageFilter.GaussianBlur( ) da biblioteca Pillow.

```python
from skimage import io
from skimage import filters

imagem6 = io.imread('pato.jpg')
out = filters.gaussian(imagem6, sigma=1)

io.imshow(out)
io.show()

```

Assim como por meio da função filters.gaussian( ) da biblioteca SKImage.

Filtro Mediano Não-linear

Outro filtro nativo da biblioteca OpenCV muito usado é o chamado filtro mediano não-linear, da função cv2.medianBlur( ), que usa de outras métricas para tentar aplicar uma suavização, em certos casos gerando resultados melhores que o filtro cv2.blur( ) comum.

```
imagemfiltrada = cv2.medianBlur(imagem6, 3)

```

Partindo diretamente para a função criamos uma variável de nome imagemfiltrada que chama a função cv2.medianBlur( ). Para a mesma é repassado como parâmetros o conteúdo de nossa imagem6 e o grau de intensidade do filtro.

```
fig = plt.figure(figsize=(10,5))
ax1 = fig.add_subplot(121)
plt.imshow(imagem6)
plt.title('Imagem Original')
ax2 = fig.add_subplot(122)
plt.imshow(imagemfiltrada)
plt.title('Imagem Filtrada')
plt.show()

```

Plotando as imagens para comparação, à esquerda temos a imagem original, enquanto à direita temos a imagem filtrada.

Realizando uma breve comparação visual podemos notar que passa esse caso o filtro acabou por suavizar os ruídos mas gerou algumas distorções ao fundo da imagem.

```
from PIL import Image
from PIL import ImageFilter

img = Image.open('pato.jpg')

blur_img = img.filter(ImageFilter.MedianFilter(7))
blur_img.show()

```

Um resultado muito próximo pode ser obtido por meio da função ImageFilter.MedianFilter( ) do módulo ImageFilter da biblioteca Pillow.

```python
from skimage import io
from skimage import filters

img = io.imread('pato.jpg')

out = filters.median(img)
io.imshow(out)
io.show()
```

Assim como por meio da função filters.median( ) da biblioteca SKImage obtemos um resultado muito parecido.

Filtro Manual de Suavização

Uma vez explorados os filtros nativos das bibliotecas, podemos abordar uma última alternativa ao propósito de suavização que é a criação de um filtro personalizado.

Como dito em outros momentos, no que tange ao processamento de imagens o ideal é ter condições de se realizar diferentes testes a fim de descobrir um resultado satisfatório.

Dependendo muito da imagem em si, alguns filtros poderão apresentar diferentes respostas, e uma última alternativa que temos é criar um filtro manual de suavização.

Essa nomenclatura manual se dará pelo fato de que simplesmente estaremos testando diferentes máscaras diretamente sobre os dados da matriz, combinando com funções já existentes, de modo a forçar a mesma a suavizar certas características do objeto da imagem.

```
filtro = np.vstack([np.hstack([imagem6, cv2.blur(imagem6, (3, 3))]),
                    np.hstack([cv2.blur(imagem6, (5,5)), cv2.blur(imagem6, (7,7))]),
                    np.hstack([cv2.blur(imagem6, (9,9)), cv2.blur(imagem6, (11,11))]),
                    ])

```

Inicialmente criamos nossa variável de nome filtro, que chama a função np.vstack( ). Essa função tem a característica de “empilhar” matrizes sobrepondo as mesmas mantendo apenas a média dos valores em cada elemento de cada linha e coluna. O que em outras palavras nada mais é do que um filtro manual matricial.

Como parâmetros da função np.vstack( ) passamos novamente a função np.vstack( ), parametrizada com nossa imagem6 e com a função cv2.blur( ) por sua vez parametrizada também com nossa image6 e com uma matriz de referência.

Esse processo se repete com diferentes tamanhos de matrizes sobrepostas, de modo a aplicar diferentes médias entre as matrizes empilhadas.

```
fig = plt.figure(figsize=(8,5))
plt.imshow(imagem6)

<matplotlib.image.AxesImage at 0x7fb0ab532e80>
```

Plotando a imagem temos ideia do tipo e nível de suavização aplicado na imagem. Novamente, o ideal é testar alguns valores diferentes de matrizes verificando e comparando os resultados.

Filtro de Realce Sobel

Retomando a categoria de filtros com propósito de realçar determinadas características em uma imagem, vamos falar do chamado filtro Sobel.

O filtro Sobel é um filtro não linear, muito eficiente quando temos de realçar linhas horizontais ou verticais em objetos de uma imagem. Por essa característica o mesmo é muito usado para realçar linhas de construções em primeiro plano de uma imagem.

```
1  import cv2
2  import numpy as np
3  import matplotlib.pyplot as plt
4
```

A esta altura você já deve estar acostumado que todo processo sempre se inicia com as importações das devidas bibliotecas, módulos e pacotes que serão usados em nosso exemplo.

```
1  imagem7 = cv2.imread('predio1.jpg', 1)
2
```

Na sequência criamos uma variável de nome imagem7 que por meio da função cv2.imread( ) carrega nossa imagem. Dessa vez, para melhor demonstrar o filtro Sobel, foi escolhida uma imagem de um prédio histórico.

```
 1  imagemfiltradaX = cv2.Sobel(imagem7,
 2                              cv2.CV_8U,
 3                              0, 1,
 4                              ksize = 5)
 5
 6  imagemfiltradaY = cv2.Sobel(imagem7,
 7                              cv2.CV_8U,
 8                              1, 0,
 9                              ksize = 5)
10
```

Em seguida vamos ao processo de criar imagens com as linhas horizontais e verticais destacadas.

Então, inicialmente criamos uma variável de nome imagemfiltradaX que chama a função cv2.Sobel( ), parametrizando a mesma com nossa imagem 7, o segundo parâmetro diz respeito ao espectro de canais a serem interpretados, definindo como cv2.CV_8U estamos abrindo compatibilidade de leitura de qualquer imagem a partir de 8bits.

Como terceiro e quarto parâmetro combinados devemos passar a referência para a forma de linha a ser realçada, 0 e 1 respectivamente irá gerar realce nas linhas horizontais detectadas na imagem.

Por fim, ksize é o parâmetro que define de acordo com os pixels alvo, o número de vizinhos a serem considerados para que seja detectado e considerado parte de uma linha. Raciocine que a detecção de linhas e contornos em uma imagem é um processo meticuloso pois independente da imagem, sempre existe uma margem grande de falsos positivos detectados que devem ser trabalhados.

```
fig = plt.figure(figsize=(15,5))
ax1 = fig.add_subplot(131)
plt.imshow(imagem7)
plt.title('Imagem Original')
ax2 = fig.add_subplot(132)
plt.imshow(imagemfiltradaX)
plt.title('Linhas Horizontais')
ax2 = fig.add_subplot(133)
plt.imshow(imagemfiltradaY)
plt.title('Linhas Verticais')
plt.show()
```

Plotando as imagens...

À esquerda temos a imagem original, ao centro a imagem com as linhas horizontais realçadas e à direita temos a imagem com as linhas verticais realçadas.

Novamente, dependendo da imagem é imprescindível que se façam testes com diferentes valores dos parâmetros, afim de encontrar um resultado satisfatório para seu propósito.

```
from skimage import io
from skimage import filters
from skimage import color
imagem = io.imread("predio1.jpg")
imagem = color.rgb2gray(imagem)
bordas = filters.sobel(imagem)
io.imshow(bordas)
io.show()
```

Realizando o mesmo procedimento, aplicando o filtro Sobel, porém sua versão nativa da biblioteca SKImage, acabamos por ter um resultado muito inferior ao obtido via OpenCV.

Isto se dá pois por meio da biblioteca SKImagem temos uma versão do filtro Sobel bastante básica e que não separa linhas horizontais de verticais, ao contrário, gera uma sobreposição de linhas que para nosso exemplo acabou sendo inútil no sentido da palavra.

Filtro Espacial Laplaciano

Outro filtro de realce de linhas/contornos muito usado tanto associado ao filtro Sobel quanto como alternativa ao mesmo é o chamado filtro Laplaciano.

Este filtro por sua vez visa detectar e destacar linhas de contorno tanto horizontais quanto verticais, porém suas funções internas usam de métodos que detectam os pixels alvo e seus vizinhos, mas elevando seus valores ao quadrado, de modo a tentar diferenciar estes do restante da imagem, servindo como detector de linhas, bordas e contornos, principalmente em casos onde tais linhas não se resumem apenas a formas horizontais e/ou verticais.

```
imagemfiltradaLap = cv2.Laplacian(imagem7,
                                  cv2.CV_8U,
                                  ksize = 5)

```

Partindo diretamente para prática criamos uma variável de nome imagemfiltradaLap que por sua vez chama a função cv2.Laplacian( ), apenas parametrizando a mesma com nossa imagem7, com o parâmetro de compatibilidade de canais de cores e com o ksize do mesmo modo como no exemplo anterior.

```
fig = plt.figure(figsize=(20,8))
ax1 = fig.add_subplot(121)
plt.imshow(imagem7)
plt.title('Imagem Original')
ax2 = fig.add_subplot(122)
plt.imshow(imagemfiltradaLap)
plt.title('Filtro Laplacian')
plt.show()

```

Realizando o plot das imagens para comparação.

À esquerda temos a imagem original, à direita a imagem após ser submetida ao filtro Laplaciano.

De fato, aqui podemos notar um padrão excelente de detecção de todo tipo de contorno, não somente horizontal e vertical, mas em todas curvaturas características da imagem.

Filtro Canny

Apresentando um último filtro de realce, vamos fazer o uso do chamado filtro Canny. Este por sua vez conta com uma série de métricas próprias, levando em consideração valores de limites para com eles calcular o mínimo e máximo de características a serem detectados para serem considerados como bordas.

```
imagemfiltradaCanny = cv2.Canny(imagem7, 90, 90)

```

Direto ao ponto, criamos uma variável de nome imagemfiltradaCanny que chama a função cv2.Canny( ), parametrizando a mesma com nossa imagem7 e com os valores de limites 90 e 90.

```
fig = plt.figure(figsize=(15,8))
ax1 = fig.add_subplot(121)
plt.imshow(imagem7)
plt.title('Imagem Original')
ax2 = fig.add_subplot(122)
plt.imshow(imagemfiltradaCanny)
plt.title('Filtro Canny')
plt.show()

```

Imagens plotadas, na esquerda a imagem original e na direita a imagem pós-processada pelo filtro Canny.

De imediato podemos ver que a imagem Canny tem boa parte de suas características simplesmente removidas, deixando apenas o que foi detectado e considerado linhas de contorno.

Mais uma vez ressalto que a cada exemplo destes cabe a realização de testes com diferentes valores em seus parâmetros.

Apenas como exemplo, alterando os valores de limites para 60 e 80 acabamos por ter uma imagem mais poluída pois o filtro acabou considerando mais características como parte de linhas e contornos em nossa imagem.

```python
from skimage import io
from skimage import feature
from skimage import color
imagem = io.imread("predio1.jpg")
imagem = color.rgb2gray(imagem)
linhas = feature.canny(imagem)
io.imshow(linhas)
io.show()
```

Realizando o mesmo procedimento, na mesma imagem, por meio do filtro Canny nativo da biblioteca SKImage, obtivemos um resultado inútil.

Encerrando este tópico e ao mesmo tempo fazendo um pequeno adendo, no que tange realizar o tratamento de imagens por meio da biblioteca OpenCV, é interessante a medida com que você se familiariza com o uso da mesma, realizar testes comparativos com diferentes parâmetros nas funções ou até mesmo explorar outras funções nativas da mesma.

```
cv2.Canny(,)
cv2.Canny

Canny(image, threshold1, threshold2[, edges[, apertureSize[, L2gradient]]]) -> edges
.   @brief Finds edges in an image using the Canny algorithm @cite Canny86 .
.
.   The function finds edges in the input image and marks them in the output map edges using the
.   Canny algorithm. The smallest value between threshold1 and threshold2 is used for edge linking. The
.   largest value is used to find initial segments of strong edges. See
.   <http://en.wikipedia.org/wiki/Canny_edge_detector>
.
.   @param image 8-bit input image
```

Ao declarar qualquer uma das funções exemplificadas anteriormente, é possível ver um breve resumo de quais parâmetros podem ser definidos assim como uma descrição das funcionalidades dos mesmos.

Para se aprofundar ainda mais e extrair melhor potencial tanto desta, quanto de outras bibliotecas, é sempre recomendável consultar a documentação oficial da biblioteca.

DETECÇÃO DE OBJETOS

Segmentação Por Binarização

Uma das técnicas mais comuns de detecção e classificação de objetos, no âmbito de visão computacional, é a chamada segmentação por binarização.

Usando da função cv2.THRESH_BINARY_INV( ), nativa da biblioteca OpenCV, podemos mapear certas características de um ou mais objetos de uma imagem para que se destaque qualquer outro objeto que não tenha as mesmas características.

Raciocine que tal técnica é muito aplicada, por exemplo, em classificação de grãos, células, lotes de objetos agrupados, etc... De modo a identificar quais são os objetos que mais se repetem e, destacar qualquer outro artefato na imagem que não siga as mesmas características.

Partindo para a prática, vamos aplicar tal técnica em uma lâmina de estudo por microscopia de um aleatório aglomerado de células sanguíneas.

```
1  import cv2
2  import numpy as np
3  from matplotlib import pyplot as plt
4
```

Como sempre, todo processo se inicia com as importações das bibliotecas, módulos e pacotes que nos oferecerão ferramentas para

nosso código.

```
1  imagem = cv2.imread('celulas.jpg')
2
```

Na sequência criamos uma variável de nome imagem que por meio da função cv2.imread( ) carrega nossa imagem de células sanguíneas.

```
1  metodo = cv2.THRESH_BINARY_INV
2
3  ret, imgBin = cv2.threshold(imagem, 200, 255, metodo)
4
```

Em seguida criamos uma variável de nome metodo que possui atribuída para si o método cv2.THRESH_BINARY_INV.

Também criamos duas variáveis de nomes ret e imgBin, respectivamente, que chamam a função cv2.threshold( ), parametrizando a mesma com nossa imagem, o intervalo de canais de cores a serem considerados (200 a 255) e nosso método atribuído a variável homônima.

```
fig = plt.figure(figsize=(15,5))

ax1 = fig.add_subplot(121)
plt.imshow(imagem)

ax2 = fig.add_subplot(122)
plt.imshow(imgBin)

plt.show()
```

Plotando nossas imagens temos, à esquerda nossa imagem normal e à direita a imagem pós processamento pelo método cv2.THRESH_BINARY_INV.

De imediato podemos notar que temos a detecção de todos objetos em cena, mesmo os mesmos tendo variações de tamanho e formato.

```
ret, imgBin = cv2.threshold(imagem, 110, 125, metodo)

```

A partir disto podemos fazer um simples janelamento alterando os valores de nossos parâmetros para ressaltar intervalos específicos de acordo com a finalidade que desejamos.

```
fig = plt.figure(figsize=(15,5))

ax1 = fig.add_subplot(121)
plt.imshow(imagem)

ax2 = fig.add_subplot(122)
plt.imshow(imgBin)

plt.show()
```

Apenas como exemplo, aplicando o método cv2.THRESH_BINARY_INV em um intervalo entre 110 e 125, conseguimos descartar todas células normais, destacando apenas os corpos estranhos da amostra.

Binarização Adaptativa

Mantendo o mesmo objeto de exemplo, porém combinando certas técnicas já entendidas anteriormente, podemos gerar certas combinações que podem ser interessantes.

A técnica conhecida como binarização adaptativa nada mais é do que a aplicação de segmentação em imagens com diferentes padrões de exposição à luz e contraste, sendo uma técnica complementar para detecção de artefatos e corpos estranhos em imagens.

```
th2 = cv2.adaptiveThreshold(imgGauss,
                            255,
                            cv2.ADAPTIVE_THRESH_MEAN_C,
                            cv2.THRESH_BINARY,
                            11,
                            3)

th3 = cv2.adaptiveThreshold(imgGauss,
                            255,
                            cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
                            cv2.THRESH_BINARY,
                            11,
                            3)

```

Tal técnica pode ser feita de forma nativa por meio da função cv2.adaptiveThreshold( ).

Em nosso exemplo, realizaremos dois testes, sendo o primeiro deles em nossa variável th2, que por sua vez chama a função cv2.adaptiveThreshold( ) parametrizando-a com nossa imgGauss, em todo intervalo de cores, aplicando o método cv2.ADAPTIVE_THRESH_MEAN_C e cv2.THRESH_BINARY.

De forma parecida, em nossa variável th3 por meio da função cv2.adaptiveThreshold( ) aplicamos os métodos cv2.ADAPTIVE_THRESH_GAUSSIAN_C e cv2.THRESH_BINARY.

```
fig = plt.figure(figsize=(15,5))
ax1 = fig.add_subplot(131)
plt.imshow(imagem)
plt.title('Imagem Original')
ax2 = fig.add_subplot(132)
plt.imshow(th2)
plt.title('Median')
ax2 = fig.add_subplot(133)
plt.imshow(th3)
plt.title('Gaussian')
plt.show()
```

Plotando as imagens para realizar as devidas comparações...

Como esperado, temos à nossa esquerda a imagem “original”, ao centro a imagem gerada e atribuída a nossa variável th2, por fim à direita a imagem gerada para th3.

Binarização Por OTSU

Técnica com resultados semelhantes aos métodos anteriores, pois aqui a chamada binarização por OTSU nada mais é do que a combinação da aplicação de certos métodos.

```
imagem = cv2.imread('celulas.jpg', 0)

```

```
metodo = cv2.THRESH_BINARY_INV + cv2.THRESH_OTSU

ret, imgBin = cv2.threshold(imagem, 0, 255, metodo)

```

Diretamente ao resultado final, criamos nossa variável metodo que chama as funções cv2.THRESH_BINARY_INV e cv2.THRESH_OTSU, somando as mesmas. Internamente ambas funções serão executadas, somando os valores de seus resultados.

Também é criada nossa variável ret que guarda os valores de limiares para posteriores consultas, e nossa imgBin. Note que aqui ambas variáveis chamam a função cv2.threshold( ), parametrizando a mesma com nossa imagem, no intervalo de canais entre 0 a 255 e aplicando as funções de metodo.

Toda essa sobreposição de funções sendo aplicada sobre nossa imagem tem por objetivo tentar uma diferente fórmula de extração de características.

```
fig = plt.figure(figsize=(12,4))

ax1 = fig.add_subplot(121)
plt.imshow(imagem)

ax2 = fig.add_subplot(122)
plt.imshow(imgBin)

plt.show()
```

Plotando as imagens temos à esquerda a imagem “original” e à direita o resultado de OTSU.

```
fig = plt.figure(figsize=(10,5))
plt.hist(imagem.ravel(), 256, [0,256])

plt.show()
```

```
print(f'O melhor limiar é: {ret}')

O melhor limiar é: 209.0
```

Apenas retomando certos conceitos, é possível gerar um histograma a partir de qualquer imagem por meio da função ravel( ). Da mesma forma, a partir de nossa variável ret podemos descobrir o melhor limiar, valor este que pode ser usado como referência para o janelamento da imagem.

Transformada de Hough

Se você veio estudando cada tópico sequencialmente, deve ter reparado que no que diz respeito a extração de características de linhas, contornos, bordas, etc... diversas técnicas podem ser aplicadas, sendo necessário o uso de certas técnicas de acordo com as particularidades da imagem em si, no que diz respeito a maneira como as linhas dos objetos estão representadas.

Nessa mesma linha de raciocínio, uma das grandes aplicações de visão computacional é o rastreamento de linhas de uma rua ou estrada para carros autônomos. Técnica também aplicada, com algumas adaptações, para monitoramento de tráfego onde o espaço monitorado se trata de um trecho de estrada.

Tenha em mente que diferentemente das outras técnicas vistas até agora, a chamada transformada de Hough em suas principais características tem métodos de detecção de linhas retas, curvas e até mesmo convergentes, simulando a perspectiva de distância onde as linhas tendem a estar separadas quando próximas do observador e juntas quando longe do mesmo, obtendo ótimos resultados nestes contextos se comparado com outras técnicas.

A transformada de Hough na verdade serve para detectar toda e qualquer forma, porém se mostra bastante eficiente em detecção

de linhas onde é possível mapear angulações nos encontros e/ou intersecções das mesmas, nos sendo muito útil no propósito de detecção de tracejados de ruas e/ou estradas.

Este tipo de característica também pode ser aproveitada a partir de uma imagem estática ou em tempo real, uma vez que um vídeo com este propósito nada mais é do que uma sequência de imagens.

```
1  import cv2
2  import numpy as np
3  import matplotlib.pyplot as plt
4
```

Importadas as bibliotecas, módulos e pacotes, uma vez não havendo erros no processo de importação podemos ir direto ao ponto.

```
1  imagem = cv2.imread('estrada.jpg', 1)
2  imagemcinza = cv2.cvtColor(imagem, cv2.COLOR_BGR2GRAY)
3
```

Inicialmente criamos uma variável de nome imagem que por meio da função cv2.imread( ) atribui para si o conteúdo de estrada.jpg.

Na sequência criamos uma segunda variável de nome imagemcinza, que como o nome sugere, converte o conteúdo de imagem para uma matriz de canais em escala de cinza. Isso é feito por meio da função cv2.cvtColor( ) parametrizada com nossa imagem e o método cv2.COLOR_BGR2GRAY.

```
1  detectalinhas = cv2.Canny(imagemcinza, 100, 150)
2  linhas = cv2.HoughLinesP(detectalinhas, 1, np.pi/180, 200, 255)
3
```

Em seguida entramos no processo de detectar propriamente as linhas, sendo aqui necessário combinar duas funções distintas, porém com mesmo propósito.

Sendo assim criamos uma variável de nome detectalinhas que chama a função cv2.Canny( ) já vista anteriormente, onde parametrizamos a mesma com nossa imagemcinza e definimos o

intervalo em que os canais serão varridos em busca das características de linhas.

Da mesma forma criamos uma variável de nome linhas que por sua vez chama a função cv2.HoughLinesP( ), repassando como parâmetros para a mesma o conteúdo de detectalinhas, a primeira coordenada, a segunda coordenada como uma relação de pi em um raio de 180 graus e por fim o intervalo de canais a serem lidos.

Lembre-se que a transformada de Hough conceitualmente depende de um ponto inicial e de um ponto de convergência entre linhas que gere uma angulação, para que seja gerada a perspectiva de encontro destas linhas.

```
1  for linha in linhas:
2    x1, y1, x2, y2 = linha[0]
3    cv2.line(imagem, (x1,y1), (x2, y2), (0,255,0), 2)
4
```

Uma vez detectadas as linhas, temos de desenhar as mesmas em cima de nossa imagem original.

Isto pode ser feito de forma simples por meio do laço for. Inicialmente declaramos nosso laço for onde será feita uma varredura no conteúdo de linhas, atribuindo seus dados/valores a cada repetição para a variável temporária linha.

Dentro do corpo do laço for declaramos quatro variáveis de nomes x1, y1, x2 e y2, respectivamente, que receberão os dados de linha em sua posição 0.

Em seguida, por meio da função cv2.line( ) desenhamos as linhas de destaque sobre nossa imagem original, unindo os pontos entre x1 e y1, assim como entre x2 e y2, desenhando tais linhas na cor verde e com uma espessura definida como 2.

```
fig = plt.figure(figsize=(15,5))
plt.imshow(imagem)

```

```
<matplotlib.image.AxesImage at 0x7f2d0a02b4a8>
```

Plotando nossa imagem podemos ver que de fato as linhas de tracejado da estrada foram perfeitamente detectadas, porém ainda existem muitos falsos positivos na vegetação ao redor, sendo necessário a realização de mais testes com diferentes parâmetros até encontrar um resultado satisfatório.

```
detectalinhas = cv2.Canny(imagemcinza, 200, 255)
linhas = cv2.HoughLinesP(detectalinhas, 1, np.pi/180, 10, 200)

```

```
fig = plt.figure(figsize=(15,5))
plt.imshow(imagem)

```

```
<matplotlib.image.AxesImage at 0x7f2d07bffc50>
```

Após realizados alguns testes com diferentes combinações de parâmetros eis que chegamos no resultado desejado, onde apenas o tracejado central e o lateral da pista ficaram em destaque, pintados em verde.

```
from skimage import io
from skimage import filters
from skimage.transform import (hough_line, probabilistic_hough_line)
from skimage.feature import canny

imagem = io.imread('estrada.jpg')
bordas = canny(imagem, 2, 1, 25)
linhas = hough_line(imagem)
detector = probabilistic_hough_line(bordas,
                                    threshold=10,
                                    line_length=5,
                                    line_gap=3)

```

```
fig = plt.figure(figsize=(15,5))
plt.imshow(imagem)

<matplotlib.image.AxesImage at 0x7f2d07bffc50>
```

Realizando o mesmo processo via biblioteca SKImage obtemos um resultado muito próximo, apenas com pequena diferença em performance de processamento.

EXTRAÇÃO DE CARACTERÍSTICAS

Um pouco diferente do que vimos até o momento, vamos entender alguns processos simples de extração de características de um determinado objeto, visando extrair detalhes que tornem possível a identificação e classificação desse objeto.

Basicamente conseguimos por meio de funções nativas da biblioteca OpenCV extrair características como cor, aspectos dimensionais, inerciais ou até mesmo topográficos.

Detecção de Cantos

Uma das aplicações mais comumente usadas é a detecção de cantos a partir de uma imagem. Raciocine que esta prática é muito importante quando queremos, por exemplo, mapear um certo ambiente, já que uma das características mais básicas de um ambiente fechado é que o mesmo tenha uma série de limites demarcados por cantos.

Apenas ressaltando que, como dito em outros momentos, à medida que avançamos de forma gradual com estes tópicos, cada vez menos nos ateremos a conceitos básicos para não os tornar tão repetitivos.

```
1  import cv2
2  import numpy as np
3  from matplotlib import pyplot as plt
4
```

Então, importadas as dependências necessárias para nosso código, vamos direto ao ponto.

```
1  imagem = cv2.imread('hospital2.jpg', 1)
2  imagemcinza = cv2.cvtColor(imagem, cv2.COLOR_BGR2GRAY)
3  imagemcinza = np.float32(imagemcinza)
4
```

Inicialmente criamos nossa variável de nome imagem que carrega os dados de nosso arquivo hospital2.jpg por meio da função cv2.imread( ).

Na sequência criamos uma variável de nome imagemcinza, que por meio da função cv2.cvtColor( ) converte nossa imagem originalmente colorida para uma em tons de cinza.

Em seguida, por uma questão de compatibilidade com as funções que utilizaremos nos próximos passos, convertemos os dados de imagemcinza para uma array numpy do tipo float32 por meio da função np.float32( ) parametrizada com imagemcinza.

```
cantos = cv2.cornerHarris(imagemcinza, 2, 3, 0.01)
el_estruturante = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (5,5))
dilatacao = cv2.dilate(cantos, el_estruturante)

```

Dando sequência criamos uma variável de nome cantos, que por sua vez chama a função cv2.cornerHarris( ) onde passamos como parâmetro para a mesma nossa imagem cinza, também definindo o tamanho de cada bloco a ser considerado, ksize e o tamanho de k.

Como em outros momentos, basicamente o que estamos fazendo é definir um pixel alvo a ser rastreado e uma série de métricas a serem aplicadas a seus vizinhos para que se faça a detecção e mapeamento dos cantos.

Também criamos um elemento estruturante em forma de elipse por meio do método cv2.MORPH_ELLIPSE, de tamanho 5x5, por meio da função cv2.getStructuringElement( ), tudo isso atribuído a nossa variável el_estruturante.

Da mesma forma, aplicamos uma dilatação a nossa imagemcinza por meio da função cv2.dilate( ), guardando estes dados em nossa variável dilatacao.

```
imagem[dilatacao > 0.05 * cantos.max()] = [0,0,255]

```

Em seguida definimos que em nossa imagem, todo valor de dilatação quando maior que 0.05 multiplicado pelo valor máximo encontrado em cantos deve ser pintado em azul, demarcando estes pontos de interesse.

```
fig = plt.figure(figsize = (10,8))
plt.imshow(imagem)

```

```
<matplotlib.image.AxesImage at 0x7fa36c7dda20>
```

Plotando nossa imagem podemos ver que de fato, boa parte dos cantos em sua forma geométrica foram detectados e marcados.

```
cantos2 = cv2.goodFeaturesToTrack(imagemcinza, 100, 0.05, 0.25)

for canto in cantos2:
    x,y = canto[0]
    cv2.circle(imagem, (x,y), 4, (0,0,255), -1)

```

Explorando uma outra possibilidade, fazendo o uso da função cv2.goodFeaturesToTrack( ) podemos obter outra projeção, baseada

em métricas que referem bons resultados de modo geral.

```
fig = plt.figure(figsize = (10,8))
plt.imshow(imagem)

<matplotlib.image.AxesImage at 0x7fa36c64f748>
```

Plotando nossa imagem podemos ver uma significativa melhora no que diz respeito a quantidade de cantos detectados na mesma imagem, tendo apenas três cantos visíveis não sendo detectados, porém, cantos estes pouco relevantes para o contexto da imagem.

```python
from matplotlib import pyplot as plt
from skimage.io import imread
from skimage.color import rgb2gray
from skimage.feature import corner_harris, corner_subpix, corner_peaks

image = imread('hospital2.jpg')
image = rgb2gray(image)

corners = corner_harris(image)
coords = corner_peaks(corners, min_distance=5)
coords_subpix = corner_subpix(image, coords, window_size=13)
fig, ax = plt.subplots()
ax.imshow(image, interpolation='nearest', cmap=plt.cm.gray)
ax.plot(coords[:, 1], coords[:, 0], '.b', markersize=3)
ax.plot(coords_subpix[:, 1], coords_subpix[:, 0], '+r', markersize=15)
ax.axis((0, 350, 350, 0))
plt.show()
```

Executando o mesmo processo via biblioteca SKImage obtivemos um resultado inferior aos anteriores, inferior inclusive ao apresentado pela função cv2.cornerHarris( ).

Template Matching

Uma aplicação muito interessante quando tratamos de reconhecimento de certas características é a chamada Template Matching.

Raciocine que tal técnica visa comparar duas imagens, processando as mesmas por igual, com as mesmas ferramentas, em busca de características que sejam compatíveis para as duas imagens.

Em outras palavras, imagine uma foto de uma pessoa de corpo inteiro e de meio corpo, assim como uma segunda imagem apenas com o rosto daquela pessoa. O algoritmo executado percorrerá todas as características das mesmas, reconhecendo e marcando o rosto da pessoa em questão.

Importante salientar que tal técnica não busca uma combinação estritamente perfeita de todos os pixels para as duas imagens, diferente disso, o algoritmo buscará tais características mesmo em imagens de diferentes proporções ou estruturas de composição.

```
1  import cv2
2  import numpy as np
3  from matplotlib import pyplot as plt
4
```

Na prática, como sempre, todo processo se inicia com as importações das bibliotecas e suas dependências.

```
1  imagem = cv2.imread('mona_lisa.jpg', 1)
2  imagem2 = cv2.imread('mona_lisa_face.jpg', 1)
3
```

Em seguida criamos duas variáveis de nomes imagem e imagem2, respectivamente, atribuindo para as mesmas o conteúdo de duas imagens por meio da função cv2.imread( ).

```
1  plt.imshow(imagem)
```

```
<matplotlib.image.AxesImage at 0x7fb77ebd1438>
```

Plotando o conteúdo de imagem podemos ver que para esse exemplo temos uma representação do famoso quadro Mona Lisa de Leonardo da Vinci.

```
1  plt.imshow(imagem2)
```

```
<matplotlib.image.AxesImage at 0x7fb77ce9b320>
```

Plotando a segunda imagem temos um simples recorte realizado a partir da primeira imagem.

```
altura, largura, canais = imagem2.shape

print(altura)
print(largura)
print(canais)

```

```
254
181
3
```

Pegando com base nossa imagem2 podemos descobrir algumas características da mesma por meio do método imagem2.shape, distribuindo esses dados para as respectivas variáveis de altura, largura e número de cais.

Sendo assim, temos uma imagem composta de 254 pixels de altura por 181 pixels de largura, totalmente colorida pois sua composição está baseada em três canais de cor.

```
metodos = ['cv2.TM_CCOEFF',
           'cv2.TM_CCOEFF_NORMED',
           'cv2.TM_CCORR',
           'cv2.TM_CCORR_NORMED',
           'cv2.TM_SQDIFF',
           'cv2.TM_SQDIFF_NORMED']

```

Preparando o ambiente para o template matching em si, criamos uma variável de nome métodos que armazena em forma de lista algumas métricas pré-programadas para a biblioteca opencv.

Essas métricas por sua vez nada mais são do que uma série de funções internas para comparação de dados das matrizes das imagens, inclusive outras métricas podem ser encontradas na documentação da biblioteca opencv. Por hora vamos nos ater apenas as 6 principaos.

```
for metodo in metodos:
  copiaimagem = imagem.copy()
  metodo2 = eval(metodo)
  templates = cv2.matchTemplate(copiaimagem, imagem2, metodo2)
  plt.imshow(templates)
  plt.title(metodo)
  plt.show()

```

Na sequência finalmente chegamos ao processo de template matching. Para isso inicialmente criamos um laço for que percorre cada uma daquelas métricas salvas em nossa variável métodos, atribuindo seu conteúdo a variável temporária metodo.

Dentro do corpo desse laço for, criamos a variável copiaimagem que como o nome sugere, realiza uma cópia da imagem original por meio da função copy( ).

Também criamos uma variável de nome metodo2 que realizará o estudo comparativo de cada métrica salva em metodo a cada laço de repetição.

Em seguida criamos a variável templates, que por sua vez chama a função cv2.matchTemplate( ), parametrizando a mesma com copiaimagem, imagem2 e método2. Em outras palavras, aqui a função realizará a comparação entre as imagens, usando de um método por vez a cada laço de repetição.

Por fim, plotaremos os resultados obtidos nesse processo.

Eis que temos os resultados no que parece ser um mapa de calor, onde para algumas métricas as cores mais quentes parecem representar regiões de melhor combinação entre as imagens.

Em suma, aqui estamos visualizando simplesmente a projeção resultante das aplicações de cada métrica como comparativo entre as imagens cedidas.

```
for metodo in metodos:
  copiaimagem = imagem.copy()
  metodo2 = eval(metodo)
  templates = cv2.matchTemplate(copiaimagem, imagem2, metodo2)
  min_val, max_val, min_loc, max_loc = cv2.minMaxLoc(templates)

  if metodo in (cv2.TM_SQDIFF, cv2.TM_SQDIFF_NORMED):
    top_left = min_loc
  else:
    top_left = max_loc
  bottom_right = (top_left[0] + altura, top_left[1] + largura)
  cv2.rectangle(imagem, top_left, bottom_right, 255, 10)

```

Partindo para a real correlação entre estes mapas de calor, o que faremos é uma adição de algumas linhas de código em nosso laço for.

Dessa vez, criamos uma estrutura condicional onde se o conteúdo de metodo consta nas funções de template matching, que este forneça os parâmetros de correlação para nossa imagem.

Dessa maneira, teremos dados para gerar uma real correlação entre as imagens, com uma representação visual de onde as mesmas conferem as mesmas características.

Por fim é simplesmente criada uma estrutura de bounding box, uma caixa que sinalizará onde estão esses pontos. Esse processo é feito por meio da função cv2.rectangle( ) parametrizado com nossa imagem, os dados de top_left e bottom_right, assim como uma cor e uma espessura de borda definidas manualmente.

```
plt.subplot(121)
  plt.imshow(templates)
  plt.title('Resultado do TM')
  plt.subplot(122)
  plt.imshow(copiaimagem)
  plt.title('Pontos Correlacionados')
  plt.suptitle(metodo)
  plt.show()

```

Plotando as imagens...

Por fim, obtivemos os respectivos resultados esperados, onde algumas métricas se mostraram eficientes em correlacionar as imagens, como por exemplo a imagem do quadrante inferior direito de nosso exemplo, com bounding box desenhada justamente no rosto da Mona Lisa, indicando que segundo aquela métrica e aquele mapa de calor aquela região é a que confere para ambas as imagens as mesmas características.

Resultado do TM
Pontos Correlacionados
cv2.TM_CCORR_NORMED
Resultado do TM
Pontos Correlacionados

Como também era de se esperar, para algumas das métricas obtivemos resultados relativamente ruins, onde é possível notar falsos positivos apontados na imagem.

DETECÇÃO DE OBJETOS

Haar Cascades Para Objetos

No processo de extração de características de objetos, ao longo das décadas diferentes engenheiros e cientistas da computação foram desenvolvendo meios e técnicas tentando resolver este tipo de problema computacional de forma lógica e eficiente.

Em seu artigo científico *Computer Cool Vision and Pattern Recognition*, Paul Viola e Michael Jones descreveram alguns algoritmos de reconhecimento de características de um objeto por meio de seus padrões em pixels, lidos sequencialmente no processo descrito como “em cascata”, onde um mapa de características poderia ser treinado para detectar todo e qualquer tipo de objeto a um baixo custo de poder computacional.

No artigo em si é explicado em detalhes que basicamente todo e qualquer objeto pode ser descrito quando o mapeamos por macro e micro regiões, destacando quais características importantes se encontram naquela região e sua apresentação básica em pixels, levando em consideração um grupo de pixels alvo e todos seus vizinhos.

Desse modo, é possível aplicar uma série de equações matemáticas sobre as matrizes de mapeamento do objeto em questão, detectando o mesmo a partir de qualquer outra imagem.

Raciocine que, por exemplo, a nível de um rosto de uma pessoa sempre teremos os olhos em um determinado quadrante, assim como a posição das linhas e contornos de nariz, boca e do formato do rosto em si. Mesmo existindo literalmente milhões de variações, todo ser humano médio padrão compartilha das mesmas características dessas estruturas, tendo variações das mesmas que o tornam único.

A nível de imagem, objetos mais simples tendem a ser mais facilmente mapeados e reconhecidos, mas independente da complexidade, todo e qualquer objeto pode ter suas características mapeadas e transliteradas para uma máscara que chamamos haar cascade.

haarcascade_cars.xml

```
1  <?xml version="1.0"?>
2  <!-- Add more negative to training. -->
3  <opencv_storage>
4  <cars3 type_id="opencv-haar-classifier">
5    <size>
6      20 20</size>
7    <stages>
8      <_>
9        <!-- stage 0 -->
10       <trees>
11         <_>
12           <!-- tree 0 -->
13           <_>
14             <!-- root node -->
15             <feature>
16               <rects>
17                 <_>
18                   6 12 8 8 -1.</_>
19                 <_>
20                   6 16 8 4 2.</_></rects>
21               <tilted>0</tilted></feature>
22             <threshold>0.0452074706554413</threshold>
23             <left_val>-0.7191650867462158</left_val>
24             <right_val>0.7359663248062134</right_val></_></_>
25         <_>
```

Apenas como exemplo, uma haar cascade configurada para detecção de carros possui mapas de todas as características

comuns a diferentes carros em suas particularidades.

Detecção de Carros

Entendido o necessário sobre uma haar cascade, podemos partir para prática.

Inicialmente, vamos realizar a detecção de carros a partir de uma imagem, usando de uma haar cascade previamente treinada para esse fim.

Toda e qualquer haar cascade terá uma série de características definidas para que tal objeto seja reconhecido, essas características obviamente variam muito de acordo com o objeto, tendo também como agravante o fato de que o mesmo tipo de objeto pode possuir muitas formas e particularidades.

Contextualizando para a detecção de carros, raciocine que uma haar cascade dedicada ao reconhecimento deste tipo de objeto deve carregar as características comuns a todo e qualquer carro.

Ainda assim é normal que nos primeiros testes você obtenha muitos falsos positivos, uma vez que tais características podem ser facilmente confundidas no processo de detecção.

Realize o exercício mental de, em poucas palavras, descrever as características básicas de um carro visto de frente. Todo carro terá uma forma total retangular, além de regiões onde estarão situados seu para-choques e para-brisas. Pois bem, agora raciocine a infinidade de veículos diferentes que certamente possuem todas essas características...

Em função disso o processo de detecção de carros, embora simples, é um tanto quanto ineficiente. Em aplicações reais é comum combinar diferentes técnicas para, por exemplo, detectar o veículo como um todo e separadamente as características de sua placa, otimizando o processo de reconhecimento.

Por hora, vamos começar pelo básico, entendendo na prática como se dá o uso de uma haar cascade.

```
1  import cv2
2
```

Como de praxe, todo processo sempre se inicia com as importações necessárias, neste caso, inicialmente usaremos apenas da biblioteca OpenCV.

```
1  classificador_carros = cv2.CascadeClassifier('haarcascade_cars.xml')
2
```

Em seguida é preciso carregar nossa haar cascade para poder fazer o uso da mesma. Para isto, criamos uma variável de nome classificador_carros que atribuída para si possui a função cv2.CascadeClassifier( ), parametrizada com o nome de nosso arquivo haar cascade. Nesse caso, haarcascade_cars.xml.

```
1  imagem = cv2.imread('carros1.jpg')
2  imagem_cinza = cv2.cvtColor(imagem, cv2.COLOR_BGR2GRAY)
3
```

Na sequência carregamos nossa imagem que servirá de fonte e a convertemos para escala de cinza.

Sendo assim, criamos uma variável de nome imagem que por meio da função cv2.imread( ) lê e carrega o conteúdo de nosso arquivo carros1.jpg.

De forma parecida, criamos uma variável de nome imagem_cinza que por sua vez chama a função cv2.cvtColor( ) parametrizando a mesma com o conteúdo de nossa variável imagem e com o método cv2.COLOR_BGR2GRAY.

```
carros = classificador_carros.detectMultiScale(imagem_cinza,
                                               scaleFactor=1.01,
                                               minNeighbors=15)

```

Dando sequência criamos uma variável de nome carros, que instancia nosso clarificador_carros, chamando a função detectMultiScale( ) passando como parâmetros para a mesma nossa imagem_cinza e definindo outros dois parâmetros; scaleFactor que diz respeito ao fator de escalonamento, uma vez que estima-se que de acordo com a perspectiva da imagem tenhamos diferentes tamanhos de carros e minNeighbors que estipula o número de possíveis objetos de mesmo tipo que possam estar próximos, parâmetro muito útil para diminuir falsos positivos que nesse contexto detectam carros dentro de carros.

```
for [x, y, l, a] in carros:
  imagem = cv2.rectangle(imagem, (x,y), (x + l, y + a), (0,0,255), 2)

```

Na sequência por meio de um laço for percorreremos cada objeto detectado, desenhando sobre o mesmo uma caixa de marcação ligando seus pontos de largura e altura.

```
cv2.imshow('Carros', imagem)
cv2.waitKey(0)

```

Por fim, por meio da função cv2.imshow( ) exibimos em tela o resultado final de nosso processo de detecção de carros.

Executando nosso código, caso não haja nenhum erro de sintaxe, é gerada uma janela nos mostrando a imagem inicialmente carregada com suas referentes bounding boxes sobre os pontos de interesse.

Repare que um veículo branco ao fundo não foi detectado, o que é esperado pois sua forma e contornos se misturam ao fundo da imagem, não sendo detectado pela haar cascade.

Dependendo o propósito e aplicação de seu código, serão necessários ajustes em parâmetros para tentar se chegar a um resultado satisfatório.

Apenas como exemplo, o simples reajuste dos valores de scaleFactor para 1.05 e minNeighbors para 1 já resultou em um processo de detecção totalmente falho.

Detecção de Objetos Personalizados

Para detecção de objetos personalizados específicos, se faz necessária a criação de uma haar cascade personalizada.

Basicamente o processo é feito treinando uma base de dados com muitas imagens do objeto em questão, no processo chamado de geração de imagens positivas, para que se extraiam todas as possíveis características do mesmo. Da mesma forma, deve ser criada uma segunda base de dados com objetos completamente aleatórios que não possuam nenhuma das características do objeto escolhido, na chamada geração de imagens negativas.

Por exemplo, para treinar o reconhecimento de uma faca, a base de dados de imagens positivas conterá muitas imagens de facas, enquanto a base de dados de imagens negativas terá cenas com diferentes objetos, mas nenhuma faca.

O processamento dessas bases de dados para a geração da haar cascade é feito por uma ferramenta chamada AdaBoost, que trabalha sobre os arquivos fonte da biblioteca OpenCV.

O processo em si, em sua sequência lógica, basicamente é definido inicialmente pela criação das bases de imagens positivas e negativas, referenciadas em seus respectivos diretórios, onde por meio das funções opencv_createsamples( ) é gerado um arquivo de extensão .vec, processado pelas funções opencv_haartraining( ) e/ou opencv_traincascade( ), gerando ao final do processo um arquivo haar cascade .xml.

Todo o processo é descrito em detalhes nos artigos:

https://coding-robin.de/2013/07/22/train-your-own-opencv-haar-classifier.html

https://pythonprogramming.net/haar-cascade-object-detection-python-opencv-tutorial/

DETECÇÃO DE FACES

Haar Cascades Para Faces

Muito semelhante ao que vimos em haar cascades para detecção de objetos, com uma rápida busca no Google podemos encontrar uma série de haar cascades com suas métricas próprias para detecção de faces.

Nesse contexto encontramos desde a Haar Cascade Frontal Faces desenvolvida pela Intel até e distribuída no formato opensource até haar cascades proprietárias, algumas simples adaptações do modelo da intel, outras totalmente criadas a partir do zero.

A questão é que se tratando da aplicação da técnica de Haar Cascades, boa parte das vezes temos um arquivo com uma série de possíveis pontos de interesse previamente mapeados e programados a serem confrontados com uma imagem, em busca de pontos em comum.

haarcascade_frontalface_default.xml

```
43   the use of this software, even if advised of the possibility of s
44 -->
45 <opencv_storage>
46 <haarcascade_frontalface_default type_id="opencv-haar-classifier">
47   <size>24 24</size>
48   <stages>
49     <_>
50       <!-- stage 0 -->
51       <trees>
52         <_>
53           <!-- tree 0 -->
54           <_>
55             <!-- root node -->
56             <feature>
57               <rects>
58                 <_>6 4 12 9 -1.</_>
59                 <_>6 7 12 3 3.</_></rects>
60               <tilted>0</tilted></feature>
61             <threshold>-0.0315119996666908</threshold>
62             <left_val>2.0875380039215088</left_val>
63             <right_val>-2.2172100543975830</right_val></_></_>
64         <_>
65           <!-- tree 1 -->
66           <_>
67             <!-- root node -->
68             <feature>
69               <rects>
70                 <_>6 4 12 7 -1.</_>
71                 <_>10 4 4 7 3.</_></rects>
72               <tilted>0</tilted></feature>
73             <threshold>0.0123960003256798</threshold>
74             <left_val>-1.8633940219879150</left_val>
75             <right_val>1.3272049427032471</right_val></_></_>
76         <_>
```

Na imagem acima um exemplo das primeiras 76 linhas (das 35713 linhas do arquivo completo) da haar cascade da Intel. Aqui, por uma questão de compatibilidade, temos um arquivo tipo .xml com literalmente milhares de combinações de mapeamentos para detecção de rostos apenas em uma projeção frontal.

Embora complexo, é possível criar suas próprias haar cascades para detecção de todo e qualquer objeto, porém, em minha simples opinião, a complexidade de tal processo de criação

não justifica seu uso, uma vez que temos diversos meios alternativos para realização do mesmo processo.

A Haar Cascade Frontal Faces da Intel pode ser livremente baixada por meio do link: https://github.com/opencv/opencv/blob/master/data/haarcascades/haarcascade_frontalface_default.xml assim como diferentes versões podem ser encontradas facilmente buscando no Google.

Detecção a Partir de Imagens

Usando da aplicação da técnica de Haar Cascades, vamos realizar o processo de detecção de rostos a partir de uma imagem.

Essa aplicação por mais básica que possa parecer, se estende por diversas outras aplicações como por exemplo detecção a partir de vídeos (uma vez que cada frame de um vídeo é uma imagem) e detecção em tempo real em monitoramento de vídeo (mesmo princípio da detecção frame a frame em vídeo).

```
import cv2
import matplotlib.pyplot as plt

!pip install opencv-contrib-python

```

Como sempre, todo processo tem seu início com as devidas importações das bibliotecas, módulos e pacotes que serão úteis

para nosso exemplo.

Apenas uma ressalva, dependendo o momento em que você estiver pondo em prática esse exemplo, como também já dito anteriormente, pode ser que algumas funcionalidades da biblioteca opencv possam ter sofrido alterações.

Sendo assim, uma prática interessante de ser realizada neste ponto é a instalação da biblioteca opencv em sua versão contrib.

A versão contrib da biblioteca é uma versão modificada e mantida pela comunidade, justamente preservando algumas funcionalidades anteriormente já descontinuadas para a versão padrão da biblioteca opencv.

Nessa linha de raciocínio, caso você não tenha essa versão instalada em conjunto com a versão base da biblioteca opencv, pode acontecer de você tentar executar determinadas ações que já não existem mais ou que sofreram alterações em sua forma de uso.

```
1  classificador = cv2.CascadeClassifier('haarcascade_frontalface_default.xml')
2
```

Importadas todas as dependências, inicialmente criamos nossa variável de nome classificador que chama a função cv2.CascadeClassifier( ), parametrizando a mesma com o nome do arquivo haar cascade a ser utilizado. Nesse caso, em forma de string, ‘haarcascade_frontalface_default.xml’.

Lembrando que dependendo da IDE ao qual você está usando pode ser necessário especificar o caminho absoluto do arquivo para que o mesmo seja lido corretamente.

```
1  imagem = cv2.imread('pessoas.jpg', 1)
2  imagem_cinza = cv2.cvtColor(imagem, cv2.COLOR_BGR2GRAY)
3
```

Em seguida, como já feito inúmeras vezes antes, realizamos a leitura de uma imagem e a conversão da mesma para escala de cinza, por meio das funções cv2.imread( ) e cv2.cvtColor( ),

respectivamente, atribuindo seus dados para as respectivas variáveis imagem e imagem_cinza.

```
1  faces = classificador.detectMultiScale(imagem_cinza)
2
```

Na sequência criamos uma variável de nome faces, que executa em nosso classificador o método detectMultiScale( ) parametrizado com nossa imagem_cinza.

Aqui, estamos aplicando a função cv2.CascadeClassifier( ) em nossa imagem_cinza, realizando todo o processo interno de pareamento da mesma com a haar cascade.

```
1  print(len(faces))
2
```

```
4
```

Realizando uma rápida consulta via função len( ) obtivemos um resultado inicial onde em nossa imagem exemplo foram detectados 4 rostos.

```
1  print(faces)
2
```

```
[[559 101 167 167]
 [239  93 223 223]
 [785 153 203 203]
 [518 607  52  52]]
```

Lembrando que como estamos trabalhando sobre matrizes de imagens, ao tentar exibir em tela o conteúdo de faces por meio de nossa função básica print( ) nos é retornado os valores em pixels para cada face detectada.

Esses valores podem ser referência para outras aplicações, uma vez que aqui temos em cada linha os valores do pixel destaque no eixo x, no eixo y, assim como valores para largura e altura em pixels.

Em nossa imagem, pegando como exemplo apenas a primeira linha, temos um rosto nas posições 559 e 101 da matriz, sendo este um rosto de 167x167 pixels de tamanho.

```
for [x, y, l, a] in faces:
    print(x,y,l,a)

559 101 167 167
239 93 223 223
785 153 203 203
518 607 52 52
```

Na sequência por meio de um laço for podemos salvar tais dados em variáveis para que com as mesmas possamos desenhar nossas bounding boxes.

```
for [x, y, l, a] in faces:
    cv2.rectangle(imagem, (x,y), (x + l, y + a), (0,0,255), 2)
```

Para o desenho da bounding box em si, realizamos um novo laço for onde os valores instanciados anteriormente para x, y, l e a agora percorrerão nossa variável faces.

Por meio da função cv2.rectangle( ) repassamos os dados de imagem, x e y, x ligado a l e y ligado a a, um parâmetro de cor da linha e um referente a espessura da linha a ser desenhada.

```
fig = plt.figure(figsize=(10,8))
plt.imshow(imagem)

<matplotlib.image.AxesImage at 0x7f8e5e54a080>
```

Plotando nossa imagem temos de fato 4 bounding boxes, referente a 4 rostos, porém repare que uma dessas bounding boxes simplesmente está apontando para um local totalmente aleatório onde foi supostamente detectado um rosto. Nesses casos, precisamos realizar alguns ajustes manuais em parâmetros para contornar este tipo de erro.

Muitos falsos positivos podem aparecer em detecção de restos e isso é perfeitamente normal. Muitos desses erros acabam ocorrendo por má interpretação da escala dos rostos em relação ao restante da imagem, assim como características ocultas ou sobrepostas que dificultam o reconhecimento uma vez que as máscaras de rosto de haar cascades esperam que todos rostos tenham características relevantes parecidas.

```
faces = classificador.detectMultiScale(imagem_cinza,
                                       scaleFactor=1.01,
                                       minNeighbors=9)

for [x, y, l, a] in faces:
  cv2.rectangle(imagem, (x,y), (x + l, y + a), (0,0,255), 2)

fig = plt.figure(figsize=(10,8))
plt.imshow(imagem)

```

Definindo manualmente parâmetros como fator de escalonamento e número mínimo de objetos vizinhos semelhantes podemos obter novos resultados.

Nesse caso, note que agora os 4 rostos foram reconhecidos, porém, alguns deles mais de uma vez e ainda temos falsos positivos na imagem, demarcando regiões onde não existem rostos.

```
faces = classificador.detectMultiScale(imagem_cinza,
                                       scaleFactor=1.01,
                                       minNeighbors=9,
                                       minSize=(50,50))

for [x, y, l, a] in faces:
    cv2.rectangle(imagem, (x,y), (x + l, y + a), (0,0,255), 2)

fig = plt.figure(figsize=(10,8))
plt.imshow(imagem)

```

Definindo um último parâmetro manualmente, responsável por definir um tamanho mínimo para os rostos detectados, é esperado que se elimine os falsos positivos encontrados anteriormente.

O retorno, como esperado, é uma nova imagem com os 5 rostos devidamente detectados, sem falsos positivos.

Detecção de Olhos

Avançando um pouco em nossos estudos, podemos entender e por em prática o processo de detecção de olhos em uma imagem.

Esse processo, complementar ao de detecção de faces, pode ser útil tanto para sistemas de reconhecimento que tenham olhos como um dos critérios, quanto em processos simples de detecção de faces onde os olhos como parte da mesma podem servir como um viés de confirmação, quando usado sobrepondo a máscara haar cascade de face, se torna mais um elemento da mesma, diminuindo o número de falsos positivos detectados como face.

O processo que seguirá nos parágrafos abaixo é, inclusive, praticamente igual ao de detecção de faces, apenas incluindo nesse processo uma nova haar cascade dedicada a detectar formas de olhos.

```
1  import cv2
2  import matplotlib.pyplot as plt
3
```

Sendo assim, inicialmente realizamos as importações.

```
1  classificador_faces = cv2.CascadeClassifier('haarcascade_frontalface_default.xml')
2  classificador_olhos = cv2.CascadeClassifier('haarcascade_eye.xml')
3
```

Na sequência criamos as variáveis dedicadas a ler e guardar o conteúdo de nossas haar cascades. Nesse caso, classificador_faces recebe chama a função cv2.CascadeClassifier( ), parametrizado com haarcascade_frontalface_default.xml, enquando classificador_olhos, por meio da mesma função, instancia haarcascade_eye.xml.

```
1  imagem = cv2.imread('criancas.jpg')
2  imagem_cinza = cv2.cvtColor(imagem, cv2.COLOR_BGR2GRAY)
3
```

Em seguida carregamos nossa imagem por meio da função cv2.imread( ) atribuindo seu conteúdo para a variável imagem.

De forma parecida realizamos a conversão da mesma para tons de cinza por meio da função cv2.cvtColor( ) parametrizada com imagem e com o método cv2.COLOR_BGR2GRAY.

```
1  faces = classificador_faces.detectMultiScale(imagem_cinza)
2
```

Feito todo o processo anterior, partimos para a detecção em si, criando nossa variável faces que instancia classificador_faces chamando a função detectMultiScale( ) parametrizado com nossa imagem_cinza.

```
1  for [x, y, l, a] in faces:
2      imagem = cv2.rectangle(imagem, (x,y), (x + l, y + a), (0,0,255), 2)
3      regiao = imagem[y:y+a, x:x+l]
4      regiao_cinza = cv2.cvtColor(regiao, cv2.COLOR_BGR2GRAY)
5      olhos_detectados = classificador_olhos.detectMultiScale(regiao_cinza)
6      print(olhos_detectados)
7
8      for [ox, oy, ol, oa] in olhos_detectados:
9          cv2.rectangle(regiao, (ox, oy), (ox+ol, oy+oa), (255,0,0), 2)
10
```

Se não houve nenhum erro de sintaxe até o momento, estima-se que o processo de detecção tenha sido realizado.

Em seguida criamos as bounding boxes, as caixas de marcação que serão desenhadas em nossos pontos de interesse detectados como rosto e olhos.

Para isso realizamos um laço for onde as variáveis temporárias x, y, l e a percorrerão o conteúdo de faces.

Dentro do corpo desse laço for, inicialmente criamos uma variável de nome imagem que chama a função cv2.rectangle( ) passando os parâmetros para o desenho da bounding box igual fizemos na detecção de faces em um exemplo anterior.

Também criamos uma variável de nome regiao que recebe de imagem os pontos demarcados para y e para x, ligando seus tracejados de y até a e de x até l.

Na sequência criamos uma outra variável de nome regiao_cinza, convertendo regiao para cinza por meio da função cv2.cvtColor( ).

Em seguida criamos uma nova variável de nome olhos_detectados que instancia classificador_olhos, chamando a função detectMultiScale( ) parametrizado com o conteúdo de regiao_cinza.

Note que nesse processo até então simplesmente realizamos o processo de detecção de faces e do desenho de suas bounding boxes, com o diferencial de realizar um recorte específico nas faces detectadas para que a partir das mesmas sejam detectados e desenhados os olhos destas faces por meio de sua haar cascade específica.

Por fim, realizamos um outro laço for dedicado a desenhar o tracejado sobre os olhos detectados no processo anterior.

```
1  print(olhos_detectados)
2
```

```
()
[[28 41 38 38]
 [82 40 40 40]]
[[29 40 37 37]
 [80 41 36 36]
 [41 95 23 23]]
[[25 26 40 40]
 [78 40 45 45]]
[[85 35 34 34]
 [25 45 40 40]]
```

Da mesma forma como feito no exemplo da detecção de faces simples, a partir deste ponto podemos ver os dados oriundos da detecção em sua forma matricial assim como plotar os resultados na imagem.

Por meio de nossa função print( ), parametrizada com o conteúdo de olhos_detectados temos os pontos de referência dos pixels onde começa e termina cada olho detectado individualmente a partir da imagem.

```
fig = plt.figure(figsize=(10,8))
plt.imshow(imagem)

<matplotlib.image.AxesImage at 0x7fce031c9048>
```

Plotando nossa imagem podemos ver em destaque via bounding boxes os rostos e os olhos detectados. Note que nessa fase inicial, como esperado, existem alguns falsos positivos uma vez que foram detectados 5 rostos e 9 olhos, logo, devemos realizar alguns ajustes nos parâmetros e realizar novos restes até encontrar um resultado satisfatório.

```python
import cv2

classificador_gatos = cv2.CascadeClassifier('haarcascade_frontalcatface.xml')

imagem = cv2.imread('gatos.jpg')
imagem_cinza = cv2.cvtColor(imagem, cv2.COLOR_BGR2GRAY)

gatos = classificador_gatos.detectMultiScale(imagem_cinza, scaleFactor=1.01)

for [x, y, l, a] in gatos:
    imagem = cv2.rectangle(imagem, (x,y), (x + l, y + a), (0,0,255), 2)

fig = plt.figure(figsize=(10,8))
plt.imshow(imagem)

```

Apenas realizando um adendo, com uma rápida pesquisa no Google é possível encontrar haar cascades previamente codificados para os mais diversos fins.

Apenas como exemplo, aplicando um haar cascade treinado para reconhecimento de rostos em gatos, podemos realizar tal classificação da mesma forma como nos exemplos anteriores para rostos humanos.

Detecção a Partir de Webcam

Das utilidades mais importantes da biblioteca OpenCV, uma das mais importantes é certamente a detecção de rostos a partir de um dispositivo de entrada, neste caso, uma webcam.

O mesmo princípio pode ser aplicado em toda interface com propósito de detecção de rostos em vídeo, como circuitos internos/externos de monitoramento de segurança por exemplo.

Você irá reparar que o processo em si é muito parecido com os demais já realizados nos exemplos anteriores, bastando obter os dados de entrada e os processar por meio de uma função dedicada a tal fim, inclusive usando de haar cascades em sua forma básica.

```
1  import cv2
2
```

Como de costume, importamos a biblioteca OpenCV por meio do comando import cv2.

```
1  detector_face = cv2.CascadeClassifier('haarcascade_frontalface_default.xml')
2
3
```

Na sequência importamos nossa haar cascade configurada para detecção de faces do mesmo modo como realizado nos exemplos anteriores.

Sendo assim, criamos nossa variável detector_face que chama a função cv2.CascadeClassifier( ) parametrizando o mesmo com nossa haarcascade_frontalface_default.xml.

```
1  captura = cv2.VideoCapture(0)
2
3  captura.release()
4  cv2.destroyAllWindows()
5
```

Seguindo com a mesma sequência lógica que viemos realizando, o próximo passo seria o processo de importação da mídia a qual iremos realizar certos processamentos. Nesse caso em particular, estaremos realizando a captura de imagens por meio de uma webcam.

O processo em si é muito simples e intuitivo, apenas contendo algumas linhas de código adicionais para que tudo funcione corretamente.

Inicialmente criamos nossa variável captura, que chama a função cv2.VideoCapture( ). Cuidado ao escrever tal linha de código pois como você bem sabe, Python é uma linguagem case sensitive e o interpretador espera que nossos códigos respeitem a codificação de letras maiúsculas e minúsculas.

Como parâmetro para a função cv2.VideoCapture( ) temos algumas opções básicas, ao não declarar nenhum parâmetro ou ao definir o mesmo manualmente como 0 estaremos definindo que se faça o uso da webcam padrão do sistema. Caso você possua mais de uma câmera, definir tal parâmetro entre 0 e 1 fará com que se alterne entre uma câmera e outra.

Por fim, sempre que estamos realizando a captura a partir de uma webcam devemos fazer uma codificação adicional onde, após feito o uso da mesma, devemos liberá-la para outros usos e também a descarregar da memória.

Por meio do método release( ) interrompemos o acesso a webcam, do mesmo modo, executando a função cv2.destroyAllWindows( ) descarregamos todo e qualquer objeto da memória relacionado a nossos dispositivos de entrada.

```
1   while True:
2     ret, img = captura.read()
3     print(ret)
4     print(img)
5
6     img_cinza = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
7     face_detectada = detector_face.detectMultiScale(img_cinza)
8
9     for (x, y, l, a) in face_detectada:
10      cv2.rectangle(img, (x, y), (x+l, y+a), (0,0,255), 2)
11
```

Na sequência realizaremos o processo de detecção das faces e desenho do tracejado sobre as mesmas.

A primeira particularidade envolvida nesse processo é que neste exemplo, lendo arquivos a partir de uma webcam, não estamos trabalhando sobre apenas uma imagem estática mas uma sequência delas. Dependendo a webcam em si sendo usada como dispositivo de entrada, essa captura pode envolver de 15 a 60 frames capturados por segundo, sendo que o processamento realizado será aplicado sobre cada uma destas imagens.

Da mesma forma onde um vídeo é uma sequência de imagens sincronizadas de modo a transmitir a impressão de movimento, a grosso modo temos uma sequência de imagens projetadas e sincronizadas a um áudio. Anteriormente, realizamos a detecção de faces a partir de uma imagem, aqui o processo é exatamente o mesmo, com um certo impacto na performance pois estaremos aplicando tal processo o número de vezes que temos em frames da captura.

Em função disso, diferentemente de um vídeo onde temos um começo, meio e fim demarcados, aqui teremos uma captura constante até que alguma outra ação a interrompa.

Para isso, o processo de detecção e desenho das bounding boxes irá rodar todo dentro de uma estrutura de repetição while.

Na programação básica em Python, a estrutura de repetição while serve para programarmos uma determinada ação que deve

ser repetida até que se atinja um determinado objetivo. Em nosso exemplo, o processo de captura, detecção e demarcação das caixas de interesse ocorrerão até que a gente mande esse processo parar pressionando uma tecla.

Sendo assim, por meio de uma estrutura while True, teremos um laço de repetição indeterminado e constante, apenas interrompido por um comando dado a partir do usuário.

```
1   while True:
2     ret, img = captura.read()
3     print(ret)
4     print(img)
5
6     img_cinza = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
7     face_detectada = detector_face.detectMultiScale(img_cinza)
8
9     for (x, y, l, a) in face_detectada:
10      cv2.rectangle(img, (x, y), (x+l, y+a), (0,0,255), 2)
11
```

Então, inicialmente declaramos nossa estrutura while True, indentado para a mesma codificaremos algumas coisas.

Primeiramente declaramos duas variáveis de nome ret e img, que chamam a função captura.read( ). A variável ret é uma simples variável de controle que retorna se o dispositivo de captura foi iniciado corretamente.

```
True
[[[126 139 165]
  [124 136 162]
  [125 137 164]
  ...
  [255 253 255]
  [255 253 255]
  [255 253 255]]

 [[125 140 155]
  [127 142 157]
  [127 139 156]
  ...
  [255 254 255]
  [255 254 255]
  [255 254 255]]
```

Inclusive se você optar por exibir em console o dado atribuído a ret por meio da função print( ) verá a cada captura de cada frame um retorno True.

Para nossa variável img será de fato atribuído os dados oriundos da captura, frame a frame, em sua forma matricial.

Na sequência criamos nossa variável img_cinza que converte o conteúdo de img para uma escala de tons de cinza por meio da função cvr.cvtColor( ).

Em seguida criamos uma variável de nome face_detectada que chama a função detectMultiScale( ) em nossa variável detector_face, que por sua vez chama a função cv2.CascadeClassifier( ).

Desse modo, estamos realizando a captura e detecção de uma ou mais faces por meio de haar cascade.

Seguindo com o código, podemos realizar a codificação do desenho das bounding boxes. Da mesma forma como feito anteriormente em outros exemplos, por meio da função cv2.rectangle( ) desenhamos sobre as áreas de interesse detectadas um tracejado e plotamos o mesmo sobre nossa imagem original situada em img.

```
14    cv2.imshow('Face Detectada', img)
15
16    if cv2.waitKey(1) == ord('x'):
17        break
18
```

Por fim, ainda indentado dentro de nossa estrutura de repetição while, realizamos o processo de exibição da captura por meio da função cv2.imshow( ), definindo um nome para a janela em forma de string e instanciando qual a origem da captura.

Lembre-se que toda essa estrutura foi criada dentro de uma estrutura de repetição while, que por sua vez, repetirá tal processo até que algo interrompa o processo.

Para não cair em um loop infinito, devemos codificar o que de fato interromperá a captura. De forma simples, podemos atribuir que ao usuário pressionar uma determinada tecla, a captura é encerrada.

Para isso criamos uma estrutura condicional dentro de nosso while, onde especificamos que se o usuário pressionar a tecla “x” de seu teclado, a captura será interrompida. Isso é feito por meio da função cv2.waitKey( ) parametrizada em 1 e definindo por meio do método ord( ) ‘x’ que, ao pressionar tal tecla, será acionado o gatilho para parar a execução deste bloco de código.

Caso a função cv2.waitKey( ) seja parametrizada com 0, qualquer tecla interromperá o processo.

Como esperado, ao rodar nosso código a webcam é ligada e é iniciado o processo de captura, exibindo em tela uma janela com uma projeção em tempo real da captura com sua bounding box.

Devemos nos lembrar sempre que, certos fatores externos podem prejudicar o processo de detecção de rostos. Em situações reais devemos ver se não existem artefatos que podem atrapalhar este processo, assim como aqui é perfeitamente aplicável todos ajustes de parâmetros realizados nos exemplos anteriores.

Landmarks de Rostos

No processo de detecção de faces, uma técnica alternativa a de haar cascades e igualmente eficiente é a de landmarks.

Basicamente quando estamos falando em landmarks de rostos também estamos falando de uma máscara a ser aplicada sobre uma imagem para detecção de rostos, porém, a técnica de landmarks envolve o reconhecimento de nada mais nada menos que 68 pontos que são característicos para todo e qualquer rosto.

Em outras palavras, um pouco diferente da técnica de haar cascades que busca padrões de diferença de pixels de olhos, por exemplo, a técnica de landmarks para essa região busca encontrar o desenho de um olho em todos seus contornos.

Os 68 pontos chave que são levados em consideração aqui, basicamente terão de ser reconhecidos em conjunto, identificando a forma de um rosto, a forma e posição dos olhos, nariz e boca, de modo a identificar este conjunto como um rosto válido.

Assim como temos uma série de haar cascades de domínio público, livre para uso, com uma rápida pesquisa no Google é possível encontrar diversos tipos de landmarks, alguns com mais ou

menos pontos, mas que de forma geral buscam identificar as mesmas características.

```
1  import cv2
2  import dlib
3
```

Como sempre, o processo se inicia com as devidas importações das bibliotecas, módulos e pacotes que serão usados ao longo do código. Dessa vez, um pouco diferente das vezes anteriores, importamos junto a biblioteca OpenCV a biblioteca DLib, pois será por meio de ferramentas desta que iremos aplicar landmarks sobre nossa captura.

```
1  captura = cv2.VideoCapture(0)
2  captura.set(cv2.CAP_PROP_FPS, 30)
3  captura.set(cv2.CAP_PROP_FRAME_WIDTH, 320)
4  captura.set(cv2.CAP_PROP_FRAME_WIDTH, 130)
5
```

Em seguida, além de inicializar nosso dispositivo de captura, devemos realizar algumas pequenas configurações fixando sua escala e número de quadros por segundo.

Para isso, inicialmente criamos nossa variável captura que recebe como atributo a função cv2.VideoCapture( ), parametrizada em 0, o que em outras palavras significa que estamos usando o dispositivo de entrada padrão de nosso sistema operacional.

Na sequência, por meio da função captura.set( ) inicialmente definimos um limite padrão de 30 quadros por segundo por meio do método cv2.CAP_PROP_FPS.

Da mesma forma, por meio da mesma função, definimos uma altura e largura de quadros padrão por meio dos métodos cv2.CAP_PROP_FRAME WIDTH. Nesta etapa, é de suma importância realizar este escalonamento pois a técnica de landmarks possui máscaras que esperam encontrar as características de um rosto em tamanho e formato.

Apenas contextualizando o parágrafo anterior, tenha em mente que as máscaras aplicadas via dlib tem certas posições predefinidas de modo a encontrar tais características de acordo com sua posição em um mapa de pixels.

Como exemplo, raciocine que existem posições como a de uma boca onde se espera que a mesma esteja entre os pixels 48 e 68 da máscara; olho direito entre 36 e 42, sobrancelha direita entre 17 e 22; nariz entre 27 e 35 e curvatura do queixo entre 0 e 17.

Para que tais posições sejam respeitadas e coincidam com as características no rosto, o mesmo deve estar escalonado de forma proporcional a estes valores.

```
1  detector = dlib.get_frontal_face_detector()
2  previsor = dlib.shape_predictor('shape_predictor_68_face_landmarks.dat')
3
```

Dando sequência, criamos uma variável de nome detector que chama a função dlib.get_frontal_face_detector( ), sem parâmetros mesmo.

Em seguida criamos uma variável de nome previsor que chama a função dlib.shape_predictor( ) parametrizando a mesma com nosso arquivo de landmarks, shape_predictor_68_face_landmarks.dat.

```
1  while True:
2      ret, img = captura.read()
3      img = cv2.flip(img, 180)
4      img_cinza = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
5      var = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8, 8))
6      mascara = var.apply(img_cinza)
7
8      deteccoes = detector(mascara, 1)
9
```

Na sequência, da mesma forma como feito no exemplo anterior, precisamos criar um sistema que leia nossos dados de captura constantemente, indefinidamente, até que seja acionado algum gatilho pelo usuário para que esta ação encerre.

Para isso criamos uma estrutura de repetição while, onde enquanto a mesma estiver com status True, suas ações serão executadas em loop.

Indentado a nossa estrutura de repetição inicialmente criamos duas variáveis ret e img, que da mesma forma como no exemplo anterior, guardam o status da captura e a imagem em si, respectivamente. Ambas chamam a função captura.read( ) inicializando nossa webcam.

Em seguida sobre nossa imagem atribuída a img, é realizado um processo de rotação em 180 graus na mesma por meio da função cv2.flip( ).

Também é realizada a conversão da mesma para tons de cinza por meio da função cv2.cvtColor( ) e pelo método cv2.COLOR_BGR2GRAY.

Em seguida criamos uma variável de nome var que inicializa a função cv2.createCLAHE( ) parametrizando a mesma com valores para clipLimie e tileGridSize. Nesta linha de código estamos definindo o escalonamento a ser gerado para que as proporções da imagem e da máscara landmark coincidam corretamente.

Uma nova variável de nome mascara é criada e atribuído para a mesma está a função var.apply( ) parametrizada com nossa img_cinza.

Por fim, declaramos uma variável de nome deteccoes que por sua vez instancia nosso detector, parametrizazndo o mesmo com o conteúdo de nossa variável mascara.

```
10    for k,d in enumerate(deteccoes):
11        shape = previsor(mascara, d)
12        for i in range(1,68):
13            cv2.circle(img,
14                       (shape.part(i).x, shape.part(i).y),
15                       1, (0, 255, 0),
16                       thickness=2)
17
```

Ainda indentado para nossa estrutura de repetição while, criamos uma estrutura responsável por ler cada frame, obter as coordenadas do mesmo e aplicar as devidas máscaras aos rostos detectados.

Para isso, inicialmente criamos um laço for onde as variáveis temporárias k e d percorrerão cada face detectada em deteccoes.

Dentro de seu corpo é criada uma variável de nome shape que instancia nosso previsor repassando para o mesmo os dados de mascara e de d.

Em seguida é criado outro laço de repetição dentro do anterior, agora percorrendo cada um dos 68 pontos definidos em nosso arquivo de landmarks.

Por meio da função cv2.circle( ) é desenhado um pequeno ponto em cada um dos pontos de landmark detectados e aplicados sobre os possíveis rostos reconhecidos. Como parâmetro de cv2.circle( ) podemos definir, como já realizado outras vezes em outros exemplos, parâmetros para que se desenhe em cima da imagem nossos pontos de interesse.

```
18        cv2.imshow("image", img)
19
20        if cv2.waitKey(1) & 0xFF == ord('q'):
21            break
22
```

Ainda na indentação de nossa estrutura de repetição while, rodamos a função cv2.imshow( ) parametrizada com um nome de janela em forma de string e com o conteúdo de img.

Da mesma forma, criamos uma simples estrutura que será o gatilho para interromper o processo de captura ao pressionar a tecla “q”. Como já comentado anteriormente, esse processo é imprescindível para que nossa estrutura de repetição não caia em um loop infinito.

```
1   cv2.destroyAllWindows()
2
```

Por fim, executamos a função cv2.destroyAllWindows( ) para que o processo de captura se encerre e todo objeto relacionado com nosso dispositivo de captura seja descarregado da memória.

Se não houver nenhum erro de sintaxe, ao executar o código a webcam é inicializada e em tempo real é possível ver uma janela com a imagem e suas devidas landmarks projetada sobre a mesma, coincidindo os pontos da máscara com as características do rosto.

Apenas por curiosidade, mesmo que algumas características não sejam reconhecidas, a máscara será aplicada da mesma forma, tentando usar como parâmetro as características presentes, embora poucas.

Detecção via Dlib + CNN

Apenas como um subtópico complementar ao processo de detecção de faces. Outra modalidade de fácil implementação, com excelente margem de acertos em sua aplicação, é a realização de detecção de faces por meio de uma rede neural convolucional.

Nos capítulos subsequentes vamos abordar em detalhes a implementação de arquiteturas de redes neurais artificiais convolucionais (modelos de rede neural artificial dedicadas ao processamento de imagens).

Por hora, vamos usar de uma pequena rede neural artificial treinada e disponível como função da biblioteca Dlib.

Para este feito iremos fazer o uso de uma rede neural simples especificamente treinada para reconhecimento de rostos.

```
1  import cv2
2  import dlib
3
```

Partindo para o exemplo, como de costume, iniciamos o processo importando as bibliotecas necessárias para nosso código. Em especial agora além da importação da biblioteca OpenCV, importaremos a biblioteca Dlib.

```
1  imagem = cv2.imread('silviogugu.jpg')
2
3  detector = dlib.cnn_face_detection_model_v1('mmod_human_face_detector.dat')
4  faces = detector(imagem)
5
```

Inicialmente criamos nossa variável de nome imagem que por meio da função cv2.imread( ) realiza a leitura de nosso arquivo de imagem silviogugu.jpg.

Na sequência criamos uma variável chamada detector, que chama a função dlib.cnn_face_detection_model_v1( ), por sua vez parametrizada com os dados do arquivo mmod_human_face_detector.dat.

Apenas contextualizando, quando tratamos de redes neurais artificiais, basicamente treinamos a mesma para que se detectem certos padrões e aprenda com os mesmos, de modo que todo aprendizado de máquina pode ser guardado em um arquivo para reutilização, sem a necessidade de treinar a rede neural novamente a cada execução.

Dessa forma, independentemente do propósito ao qual será aplicada a rede neural artificial, existirá um arquivo gerado para transferência de aprendizagem. Em nosso exemplo, usaremos de um modelo de rede neural pré-treinada para reconhecimento exclusivo de faces humanas.

Por fim, é declarada uma variável de nome faces que instancia nosso detector passando para sua função interna o conteúdo de imagem.

```
1  for face in faces:
2      e, t, d, b, c = (int(face.rect.left()),
3                       int(face.rect.top()),
4                       int(face.rect.right()),
5                       int(face.rect.bottom()),
6                       face.confidence)
7      cv2.rectangle(imagem, (e,t), (d, b), (255,255,0), 2)
8      print(f'Margem de precisão: {c}')
9
```

Em seguida, por meio de um laço for que percorrerá cada face identificada em faces, são criadas as caixas de marcação, com algumas pequenas diferenças das criadas até então em exemplos passados.

Para esse processo via biblioteca Dlib, temos de criar 5 variáveis, e, t, d, b e c, respectivamente atreladas aos dados de esquerda, topo, direita, baixo e margem de confiabilidade da detecção.

No exemplo acima criamos estas variáveis de forma reduzida, porém é totalmente possível criar cada uma independentemente em sua linha de código chamando a função face.rect.coordenada( ). Com exceção de c que recebe dados de face.confidence.

Por fim, por meio da função cv2.rectangle( ), passamos os dados de imagem, as coordenadas, cor e espessura da borda da caixa de marcação a ser desenhada.

```
1  print(faces)
2  print(f'Número de faces: {len(faces)}')
3
4  cv2.imshow('Faces', imagem)
5  cv2.waitKey(0)
6  cv2.destroyAllWindows()
7
```

Encerrando, são realizadas as codificações para exibição do número de faces detectadas, assim como para geração da janela

que exibirá o arquivo de faces identificadas e os referidos gatilhos para encerramento deste processo.

```
C:\Users\Fernando\anaconda3\envs\ReconhecimentoDlib\python.exe
Margem de precisão: 1.0957918167114258
Margem de precisão: 1.0755246877670288
<_dlib_pybind11.mmod_rectangles object at 0x000002670FCFA378>
Número de faces: 2
```

Não havendo nenhum erro de sintaxe, o retorno que temos é, como esperado, a janela com os rostos destacados na imagem, assim como em nosso terminal os dados para margem de precisão na detecção e o número de faces detectadas.

RASTREAMENTO DE OBJETOS

A esta altura, muito provavelmente você já deve estar com uma boa base da biblioteca OpenCV, entendendo seus meios e métodos de processamento de imagens assim como a sequência lógica usada para a maioria das aplicações.

Sendo assim, partiremos para alguns conceitos finais um pouco mais avançados, mas igualmente simples por parte de implementação via OpenCV.

Uma das aplicações mais úteis ao meu ver é o uso de visão computacional para rastreamento de objetos.

Um pouco diferente do simples processo de detecção e reconhecimento, o chamado rastreamento é uma ferramenta excelente para monitorar um determinado ambiente em seu contexto de ações.

Raciocine que desde o acompanhamento de uma bola em um jogo de futebol até sistemas de radar em autoestradas ou leitura do ambiente a partir de um carro autônomo, tudo isso envolve em alguma camada de processamento o que chamamos de rastreamento de objetos.

Por parte lógica em algoritmo, lembre-se que um vídeo nada mais é do que uma sequência de imagens apresentada de forma sincronizada, de modo a oferecer uma sensação de movimento.

Nesse contexto, algoritmos de rastreamento normalmente tem seu foco em identificar a posição de determinados objetos comparando o frame anterior com o atual, fazendo uma leitura total de cada frame entendendo o deslocamento de um objeto entre os mesmos acusando movimento em uma determinada direção e com uma certa velocidade.

Por exemplo, em um jogo de futebol um determinado jogador correndo da linha média do campo em direção a área do time adversário, frame a frame estará se deslocando naquela direção, enquanto as características do fundo, que nesse caso é o campo, permanece imóvel.

A ideia de rastreamento de objetos é muito aplicada implicitamente também na indústria de cinema, onde cenas gravadas em chroma key tem um fundo falso com alguns marcadores para que seja inserido um fundo via CGI e que o ator tenha sua posição e movimento projetada sobre esse fundo de forma natural e realista. Nesses casos, o interessante é que a técnica de rastreamento em si é aplicada sobre o fundo, e não sobre o objeto (ator/atriz).

Outro ponto a salientar antes de partirmos para a prática é que, diferentemente dos processos anteriores onde era necessário o uso de haar cascades específicas para cada tipo de objeto, aqui o processo não envolve nenhuma máscara de reconhecimento, pois o algoritmo quando implementado sobre uma imagem, captura ou vídeo, identificará todo e qualquer objeto em função de seu movimento e não sua forma, contornos ou qualquer outra característica.

Rastreamento de Um Objeto Personalizado

Partindo para a prática, vamos entender o processo de rastreamento de um objeto personalizado.

O exemplo que estaremos codificando é baseado em rastreamento de um carro a partir das imagens de uma câmera de segurança, porém, como explicado no tópico anterior, este processo não fica limitado somente a este tipo de objeto, podendo ser adaptado para qualquer finalidade de rastreamento.

```
!pip install opencv-python==3.4.4.19
!pip install opencv-contrib-python==3.4.4.19

```

Como dito no início deste livro, quando necessário, seria colocado uma nota sobre a versão específica da biblioteca OpenCV a ser utilizada, evitando erros de compatibilidade.

Para esse exemplo em especial é necessário fazer o uso das versões 3.4.4.19 da biblioteca OpenCV oficial e de sua versão contrib, que possui algumas ferramentas implementadas pela comunidade, inclusive ferramentas de rastreamento. Este código não funcionará em versões anteriores a 3.4.4.19.

```
1  import cv2
2  import sys
3  from random import randint
4
```

Como sempre, o processo se inicia com as devidas importações das bibliotecas módulos e pacotes... Neste caso, usaremos além da biblioteca OpenCV as bibliotecas sys e random, pois posteriormente criaremos um sistema de validação onde será necessária a geração de números aleatórios.

```
1  (vmax, vmin, vsub) = (cv2.__version__).split('.')
2
```

Ao longo desse exemplo usaremos de algumas estruturas de validação, pois o processo é um tanto quanto mais complexo do que os exemplos abordados anteriormente.

Sendo assim, em algumas partes do código serão executadas estruturas condicionais baseadas na versão da biblioteca OpenCV para escolha correta do mecanismo rastreador.

Para isso, criamos três variáveis reservadas com nomes vmax, vmin e vsub, que terão seus valores atribuídos com base na justaposição dos dados contidos em __version__, ou seja, para vmax será atribuído o valor 3, para vmin será atribuído o valor 4 e para vsub será atribuído o valor 4.

```
1   rastreadores = ['BOOSTING',
2                   'MIL',
3                   'KCF',
4                   'TLD',
5                   'MEDIANFLOW',
6                   'MOSSE',
7                   'CSRT']
8   rastreador = rastreadores[6]
9   print(rastreador)
10
```

Após essa etapa, podemos de fato começar a criar os blocos de código dedicados ao processo de rastreamento.

Dessa forma, inicialmente criamos uma variável de nome rastreadores que em forma de lista recebe 7 nomes de rastreadores nativos do OpenCV. Pela documentação do mesmo é até possível encontrar mais métodos rastreadores, aqui vamos nos ater aos 7 mais comumente usados com resultados satisfatórios.

Também criamos uma variável de nome rastreador que por sua vez receberá, de acordo com o número de índice especificado, um dos rastreadores contidos em rastreadores. Neste caso, índice 6 referente ao rastreador CSRT, e como de costume, ao rodar nosso código será interessante realizar testes com os demais.

```
1   if int(vmin) < 3:
2       rx = rastreador
3   else:
4       if rastreador == 'BOOSTING':
5           rx = cv2.TrackerBoosting_create()
6       if rastreador == 'MIL':
7           rx = cv2.TrackerMIL_create()
8       if rastreador == 'KCF':
9           rx = cv2.TrackerKCF_create()
10      if rastreador == 'TLD':
11          rx = cv2.TrackerTLD_create()
12      if rastreador == 'MEDIANFLOW':
13          rx = cv2.TrackerMedianFlow_create()
14      if rastreador == 'MOSSE':
15          rx = cv2.TrackerMOSSE_create()
16      if rastreador == 'CSRT':
17          rx = cv2.TrackerCSRT_create()
18
```

Na sequência, embora pareça um pouco confuso, criaremos uma estrutura condicional que atuará como uma estrutura validadora, pois caso o valor atribuído a vmin for inferior a 3, bastará usar do rastreador definido em nossa variável rastreador por meio de seu índice, caso contrário, é necessário chamar manualmente o método que cria e inicializa um rastreador.

Então, simplesmente criamos uma estrutura condicional onde quando não possível inicializar o rastreador por meio de nossa variável rastreador, usaremos a string atribuída para a mesma para chamar a função referente a tal nome de rastreador.

Por exemplo, caso o valor de rastreador seja equivalente a TLD, a variável rx chamará manualmente a função cv2.TrackerTLD_create( ), que inicializa tal rastreador a ser aplicado em nosso objeto de entrada.

```
1   video = cv2.VideoCapture('estrada.mp4')
2   if not video.isOpened():
3       print('ERRO: Não foi possível carregar o vídeo!!!')
4       sys.exit()
5   ok, frame = video.read()
6   if not ok:
7       print('ERRO: Não foi possível ler o arquivo de vídeo!!!')
8       sys.exit()
9   bbox = cv2.selectROI(frame, False)
10  ok = rx.init(frame, bbox)
11  colors = (randint(0, 255), randint(0, 255), randint(0, 255))
12
```

Em seguida, criamos nossa variável vídeo que por sua vez chama a função cv2.VideoCapture( ), parametrizando a mesma com nosso arquivo de vídeo estrada.mp4.

Na sequência criamos outra estrutura de validação, responsável por verificar se o carregamento e leitura do vídeo ocorreram corretamente.

Isso pode ser facilmente realizado criando uma estrutura condicional para tal fim, a primeira delas, caso não ocorra com sucesso a execução da função vídeo.isOpened( ), é exibido para o usuário uma mensagem de erro e encerrado o processo por meio da função sys.exit( ).

Exatamente da mesma forma é criada outra estrutura condicional onde caso o status atribuído para ok (que recebe True ou False como atributo) ou não houverem dados atribuídos para frame, por meio da função vídeo.read( ), também é exibida ao usuário uma mensagem de erro e encerrado o processo.

Na sequência vamos fazer algo diferente do que já vimos até então no processo de aplicação de bounding boxes, apenas no intuito de demonstrar outras possibilidades.

Criamos uma variável de nome bbox que recebe como atributo a função cv2.selectROI( ) repassando como parâmetro para a mesma cada frame. Neste processo, ao iniciar nosso código será carregado o vídeo porém pausado em seu primeiro frame, nos

projetando um cursor que nos permitirá desenhar a bounding box sobre o objeto ao qual queremos rastrear.

Na sequência, por meio da função rx.init( ) de nossa variável ok, parametrizada com frame e bbox, iniciamos a execução do vídeo que inicialmente estava pausado.

Por fim, realizamos uma pequena codificação apenas por uma questão de compatibilidade, permitindo que todos canais de cor sejam processados em simultâneo no vídeo.

```
1  while True:
2      ok, frame = video.read()
3      if not ok:
4          break
5      timer = cv2.getTickCount()
6      ok, bbox = rx.update(frame)
7      fps = cv2.getTickFrequency() / (cv2.getTickCount() - timer)
8      if ok:
9          (x, y, w, h) = [int(v) for v in bbox]
10         cv2.rectangle(frame, (x, y), (x + w, y + h), colors, 2, 1)
11     else:
12         cv2.putText(frame, 'Falha no rastreamento!!!', (100, 80),
13                     cv2.FONT_HERSHEY_SIMPLEX, .75, (0, 0, 255), 2)
14     cv2.putText(frame, rastreador + 'Rastreador', (100, 20),
15                 cv2.FONT_HERSHEY_SIMPLEX, .75, (50, 170, 50), 2)
16     cv2.putText(frame, 'FPS: ' + str(int(fps)), (100, 50),
17                 cv2.FONT_HERSHEY_SIMPLEX, .75, (50, 170, 50), 2)
18     cv2.imshow('Rastreamento', frame)
19     if cv2.waitKey(1) & 0XFF == 27:
20         break
21
```

Partindo para parte final, de forma parecida como fizemos ao capturar vídeo a partir de uma webcam, precisamos criar algumas estruturas de código dentro de um laço de repetição, uma vez que estamos processando um vídeo frame a frame e reproduzindo tais frames em forma de vídeo.

Dando continuidade, criamos nossa estrutura while True, dentro do corpo da mesma instanciamos nossas variáveis ok e frame, inicializando o processo de ler nosso arquivo de vídeo por meio da

função vídeo.read( ). Caso haja algum erro nesse processo, o loop já é pausado e encerrado o processo.

Em seguida criamos uma variável de nome timer que inicia a função cv2.getTickCount( ), sem parâmetros mesmo. Basicamente o dado retornado para timer será usado no processo de sincronização dos frames do vídeo em sua reprodução.

Na sequência atualizamos os dados/valores de nossas variáveis ok e bbox por meio da função rx.updade( ) parametrizado com frame. Este processo serve para que a cada atualização seja lido um novo frame subsequente, e não o mesmo já processado.

Também criamos uma variável de nome fps que por sua vez recebe o valor obtido pela diferença entre os dados obtidos pelas funções cv2.getTickFrequency( ) e cv2.getTickCount( ) – timer. Esses dados irão gerar o número de quadros por segundo sendo processados e projetados.

Ainda indentado em nossa estrutura de repetição criamos uma estrutura condicional para o desenho das bounding boxes nos frames subsequentes ao primeiro. Lembrando que no primeiro frame desenharemos manualmente a bounding box sobre o objeto a ser rastreado, nos próximos frames estas bounding boxes serão desenhadas acompanhando o objeto em seu movimento.

Também é criada uma estrutura básica para colocar em tela as informações de qual rastreador está sendo utilizado no processo e o número de quadros por segundo que estão sendo projetados. Este bloco de código é totalmente opcional.

Por fim por meio da função cv2.imshow( ) criamos a janela em si ao qual será projetado cada frame, assim como uma codificação com a função cv2.waitKey( ) para que possamos encerrar o processo a qualquer momento pressionando a tecla Esc.

```
C:\Users\Fernando\anaconda3\envs\RastreamentoOpenCV\python.exe
CSRT
Select a ROI and then press SPACE or ENTER button!
Cancel the selection process by pressing c button!

Process finished with exit code 0
```

Se não houver nenhum erro de sintaxe, nem de carregamento de nosso vídeo em questão, ao executar nosso código é possível ver um retorno gerado pela função cv2.video.read( ) orientando o usuário a selecionar a área de interesse e pressionar a tecla espaço ou enter para confirmar.

Neste momento você já tem a janela gerada com o primeiro frame, assim como o cursor do mouse, podendo desenhar a bounding box sobre o objeto o qual você deseja rastrear.

Neste caso, apenas como exemplo, desenhei uma caixa de marcação sobre a van amarela situada na região superior da janela. Ao pressionar espaço, o vídeo é executado e é possível acompanhar o rastreamento de tal veículo.

Rastreamento de Múltiplos Objetos

Com base na técnica aplicada ao exemplo anterior, podemos realizar o rastreamento manual de mais de um objeto (quantos forem necessário) simplesmente fazendo o uso da função cv2.MultiTrackerCreate( ) combinada ao processo já visto anteriormente de criação de rastreador manual.

Apenas como exemplo, também faremos uma codificação um pouco diferente da do exemplo anterior, apenas no intuito de demonstrar algumas outras possibilidades, haja visto que temos sérias limitações ao tentar reproduzir tais exemplos aqui em forma de texto.

```
1  import cv2
2  import sys
3  from random import randint
4
```

Mais uma vez iniciamos todo o processo com as devidas importações.

```
1  rastreadores = ['BOOSTING',
2                  'MIL',
3                  'KCF',
4                  'TLD',
5                  'MEDIANFLOW',
6                  'MOSSE',
7                  'CSRT']
8
```

Criamos nossa variável rastreadores que recebe em forma de lista strings com os nomes dos rastreadores disponíveis pelas funções internas da biblioteca OpenCV.

```
1  def escolhe_rastreador(rastreador):
2      if rastreador == rastreadores[0]:
3          rx = cv2.TrackerBoosting_create()
4      elif rastreador == rastreadores[1]:
5          rx = cv2.TrackerMIL_create()
6      elif rastreador == rastreadores[2]:
7          rx = cv2.TrackerKCF_create()
8      elif rastreador == rastreadores[3]:
9          rx = cv2.TrackerTLD_create()
10     elif rastreador == rastreadores[4]:
11         rx = cv2.TrackerMedianFlow_create()
12     elif rastreador == rastreadores[5]:
13         rx = cv2.TrackerMOSSE_create()
14     elif rastreador == rastreadores[6]:
15         rx = cv2.TrackerCSRT_create()
16     else:
17         rx = None
18         print('ERRO: Rastreador inválido!!!')
19     return rx
20
```

Na sequência criamos manualmente uma função de nome escolhe_rastreador( ) parametrizada com o conteúdo de nossa variável rastreador.

Dentro do corpo dessa função temos uma codificação quase idêntica a do exemplo anterior, onde em função do número do índice instanciamos um determinado rastreador criando a função que o chama manualmente.

Caso haja algum erro, é retornada uma mensagem ao usuário, caso não, o rastreador selecionado é atribuído para a variável rx.

```
1  captura = cv2.VideoCapture("estrada.mp4")
2
3  ok, frame = captura.read()
4  if not ok:
5      print('ERRO: Não foi possível ler o arquivo de vídeo.')
6      sys.exit(1)
7
```

Em seguida criamos o sistema que inicializa nosso dispositivo de entrada, nesse caso, realizando a leitura do arquivo de vídeo estrada.mp4 atribuindo seus dados para variável captura.

De forma parecida, para as variáveis ok e frame são atribuídos os dados da inicialização e do conteúdo do vídeo, respectivamente, por meio da função captura.read( ).

Também é criada uma estrutura condicional para caso o processo de leitura do vídeo não aconteça como esperado, uma mensagem é exibida ao usuário e o processo é encerrado descarregando qualquer objeto relacionado da memória.

```
1   while True:
2       bbox = cv2.selectROI('MultiTracker', frame)
3       bboxes.append(bbox)
4       colors.append((randint(0, 255), randint(0,255), randint(0,255)))
5       print('Selecione o objeto a ser rastreado e pressione ENTER.')
6       print('Selecione o próximo objeto a ser rastreado.')
7       print('Pressione Q para sair da seleção e começar a rastrear.')
8       print('Pressione C para cancelar a seleção.')
9       print('Pressione ESC para sair.')
10      k = cv2.waitKey(0) & 0XFF
11      if (k == 113):
12          break
13
```

Também é criado, de forma muito parecida com o exemplo anterior, um laço de repetição para que seja criadas as caixas de marcação das áreas de interesse. Lembrando que para esse exemplo é possível demarcar mais de um objeto de acordo com o propósito da aplicação.

Indentado ao corpo dessa estrutura de repetição, temos a inicialização ds funções de seleção da área de interesse, criação e mapeamento das bounding boxes, inicialização dos 3 canais de cor em toda sua extensão, além de algumas instruções que serão exibidas para o usuário via terminal. Além é claro, do gatilho para que possamos encerrar o processo manualmente ao pressionar uma tecla, nesse caso, tecla ESC.

```
rastreador_escolhido = 'CSRT'
multi_rastreador = cv2.MultiTracker_create()

for bbox in bboxes:
    multi_rastreador.add(escolhe_rastreador(rastreador_escolhido),
                         frame,
                         bbox)

```

Na sequência é criada uma variável de nome rastreador_escolhido, que recebe em forma de string o nome de um rastreador. Lembrando que este dado será instanciado em nossa função rastreador_escolhido( ), ao invés de escolher um rastreador via índice como no exemplo anterior.

Em seguida é criada a variável multi_rastreador que chama a função cv2.MultiTracker_create( ), sem parâmetros mesmo, apenas inicializando a mesma. Esta função que nos permitirá rastrear quantos objetos forem preciso em uma cena.

Próximo passo, por meio de um laço for que percorre nossa codificação para criação de bounding boxes, integramos a mesma em nosso multi_rastreador, por meio da função add( ) parametrizando o mesmo com nossa função escolhe_rstreador( ) por sua vez parametrizado com o conteúdo de rastreador_escolhido, frame e bbox.

Esta é outra forma de gerar bounding boxes em nossos objetos rastreados em vídeo.

```
while captura.isOpened():
    ok, frame = captura.read()
    if not ok:
        break
    ok, boxes = multi_rastreador.update(frame)
    for i, newbox in enumerate(boxes):
        (x, y, w, h) = [int(v) for v in newbox]
        cv2.rectangle(frame, (x, y), (x + w, y + h), colors[i], 2, 1)
    cv2.imshow('Rastreamento Múltiplo', frame)
    if cv2.waitKey(1) & 0XFF == 27:
        break

```

Finalizando nosso rastreador de múltiplos objetos personalizados, hora de criar a codificação responsável por nos gerar um retorno que nesse caso é o vídeo sendo executado, com as respectivas bounding boxes demarcadas sobre os objetos rastreados.

Para isso, criamos uma estrutura de repetição que, enquanto o vídeo estiver sendo executado, algumas ações serão realizadas.

Primeira delas é a atualização das variáveis ok e frame, uma vez que todo o processo se repetirá frame a frame do vídeo.

Da mesma forma, para ok e boxes é realizada a atualização por meio da função .update( ) em nosso multi_rastreador.

Para que a cada frame seja desenhada uma nova bounding box para cada objeto em sua nova posição, é criado um laço for que percorre a variável boxes a cada repetição, desenhando sobre a mesma uma nova demarcação.

Por fim, por meio da função cv2.imshow( ) acionamos a projeção do vídeo em si, assim como codificamos um gatilho para que o processo se encerre a qualquer momento quando pressionada a tecla ESC.

Apenas retomando alguns pontos, lembre-se que inicialmente teremos um vídeo pausado em seu primeiro frama, onde desenharemos manualmente a bounding boxes sobre os objetos a serem rastreados, após feito esse processo, ao apertar a tecla Q o vídeo começará a ser executado já nos exibindo o rastreamento de cada um dos objetos.

Vale lembrar que, cada objeto selecionado terá seu impacto na performance do modelo, um número grande de objetos sendo rastreados em tempo real tende a exigir muito processamento e consequentemente pode haver queda no número de quadros por segundo sendo exibidos como retorno.

Apenas como exemplo, supondo que para um determinado fim temos de rastrear estes três veículos, desenhamos manualmente sobre os mesmos as referentes caixas de marcação.

```
Selecione o objeto a ser rastreado e pressione ENTER.
Selecione o próximo objeto a ser rastreado.
Pressione Q para sair da seleção e começar a rastrear.
Pressione C para cancelar a seleção.
Pressione ESC para sair.
```

Via terminal temos as instruções que programamos manualmente.

Após selecionar os veículos e pressionar a tecla Q, temos como esperado, o vídeo sendo executado com os objetos em questão sendo rastreados.

Detecção + Rastreamento de um Objeto

Entendidos os princípios básicos dos processos de detecção e de rastreamento de objetos personalizados, podemos fazer um exercício simulando uma aplicação que envolva ambos processos.

Como mencionado no início do livro, a ideia fundamental era aprendermos sobre os tópicos de forma gradual, sendo que a medida em que fossemos progredindo, não nos ateríamos a reexplicar repetidas vezes os mesmos conceitos básicos.

Porém, este é um bom momento para por meio de um exemplo prático, realizar uma abordagem que combine a lógica dos mecanismos de detecção e de rastreamento de objetos.

Sendo assim, vamos combinar os meios vistos anteriormente explorando algumas outras possibilidades ao realizar a detecção automatizada e o rastreamento de uma pessoa a partir de um vídeo.

Diferentemente dos exemplos anteriores onde desenhávamos manualmente as caixas de marcação sobre as áreas de interesse,

vamos trazer algo mais próximo da aplicação real que é onde todo esse processo é feito de forma automatizada.

```
1  import cv2
2  import sys
3  from random import randint
4
```

Como sempre, o processo é iniciado com as importações das bibliotecas, módulos e pacotes necessários para o funcionamento do código.

```
1  video = cv2.VideoCapture('pessoa.mp4')
2  detector = cv2.CascadeClassifier('fullbody.xml')
3  rastreador = cv2.TrackerCSRT_create()
4
```

De início criamos uma variável de nome vídeo que por meio da função cv2.VideoCapture( ) realiza a leitura de um arquivo de vídeo.

Na sequência criamos uma variável de nome detector, que por sua vez chama a função cv2.CascadeClassifier( ) parametrizado com um arquivo haar cascade pré-programado para detecção de pessoas em corpo inteiro.

Lembrando que diferentemente dos exemplos anteriores onde demarcávamos um objeto a ser rastreado, aqui no processo de automatização de detecção de uma pessoa para o rastreamento da mesma voltamos a usar do método via haar cascades.

Também criamos uma variável de nome rastreador que inicializa a função cv2.TrackerCSRT_create( ), sem parâmetros mesmo. Aqui, no lugar de uma série de rastreadores usaremos apenas o CSRT, pois é um dos mais, senão o mais, eficiente para detecção de pessoas via haar cascades.

```
1  if not video.isOpened():
2      print('ERRO: Não foi possível abrir o vídeo.')
3      sys.exit()
4  ok, frame = video.read()
5  if not ok:
6      print('ERRO: Não foi possível ler o arquivo de vídeo.')
7      sys.exit()
8
```

Dando sequência criamos a codificação necessária pelo processo de validação onde caso não seja possível abrir e ler o vídeo, uma mensagem de erro é apresentada ao usuário.

Isso se dá apenas verificando se a função vídeo.isOpened( ), assim como a função vídeo.read( ) foram inicializadas com sucesso, encerrando o processo por meio da função sys.exit( ) descarregando tais objetos da memória.

```
1  def detector_pessoas():
2      while True:
3          ok, frame = video.read()
4          frame_cinza = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
5          detector_frame = detector.detectMultiScale(frame_cinza,
6                                                     scaleFactor=1.09)
7          for (x, y, l, a) in detector_frame:
8              cv2.rectangle(frame, (x,y), (x+l, y+a), (0,255,0), 2)
9              cv2.imshow('Detecções', frame)
10             cv2.waitKey(0)
11             cv2.destroyAllWindows()
12             return x, y, l ,a
13
```

Na sequência criamos a função detector_pessoas( ), onde integraremos o processo de leitura do vídeo em seu primeiro frame, detectando o objeto em questão e desenhando a bounding box em torno do mesmo.

Para isso, dentro do corpo de nossa função detector_pessoas( ) criamos uma estrutura de repetição while True, onde indentado para a mesma são criadas as variáveis ok e frame, recebendo dados de vídeo.read( ).

Em seguida é criada a variável frame_cinza, que realiza a conversão de cores do frame de colorido para escala de cinza por meio da função cv2.cvtColor( ) parametrizada com frame e com o método cv2.COLOR_BGR2GRAY.

Lembrando que esta etapa é uma codificação adicional, onde por uma questão de performance, realizamos os devidos processamentos sobre uma imagem em preto e branco.

Também criamos uma variável detector_frame que instancia nosso detector chamando a função detectMultiScale( ), passando como parâmetros para o mesmo nosso frame_cinza e definindo manualmente o fator de escalonamento do objeto em relação ao restante da imagem para 1.09 no parâmetro scaleFactor.

Na sequência é realizada a codificação do bloco de código responsável por desenhar a caixa de marcação sobre as coordenadas encontradas no objeto, assim como a geração da janela que exibirá este primeiro frame.

Por fim, é retornado os valores encontrados para as coordenadas x, y, l, a, pois estes valores serão repassados em outras funções para que o desenho das bounding boxes funcione corretamente aqui neste modelo automatizado.

```
1  bbox = detector_pessoas()
2  ok = rastreador.init(frame, bbox)
3  canais = (randint(0, 255), randint(0, 255), randint(0, 255))
4
```

Fora da indentação de nossa estrutura de repetição, de volta ao escopo global de nosso código, criamos a variável bbox que apenas inicializa nossa função detector_pessoas( ), sem parâmetros mesmo.

A variável ok é atualizada com nosso rastreador, chamando a função init( ), parametrizando a mesma com frame e bbox, dessa forma nosso detector será retroalimentado automatizando o processo de detecção e desenho da caixa de marcação frame a frame.

Por fim criamos uma variável de nome canais que gera aleatoriamente uma cor para a bounding box, por meio das funções randint( ) em cada canal de cor, gerando valores entre 0 a 255.

```
1  while True:
2      ok, frame = video.read()
3      if not ok:
4          break
5      ok, bbox = rastreador.update(frame)
6      if ok:
7          (x,y,w,h) = [int(v) for v in bbox]
8          cv2.rectangle(frame,
9                        (x,y),
10                       (x + w, y + h),
11                       canais,
12                       2, 1)
13     else:
14         bbox = detector_pessoas()
15         rastreador = cv2.TrackerMOSSE_create()
16         rastreador.init(frame, bbox)
17     cv2.imshow('Rastreamento', frame)
18     esc = cv2.waitKey(1) & 0XFF
19     if esc == 27:
20         break
21
```

Partindo para a parte final do processo, criamos nossa segunda estrutura de repetição, esta voltada para a reprodução do vídeo em si.

Dessa forma, após declarado um novo while True, indentado para o mesmo nossas variáveis ok e frame são atualizadas com os novos dados oriundos da função vídeo.read( ).

É criada uma simples estrutura de validação onde caso a reprodução do vídeo não ocorra o processo já é encerrado.

Em seguida ok e bbox são atualizados agora instanciando nosso rastreador, chamando a função update( ) para o mesmo, parametrizado com os dados de frame.

Logo na sequência é criada uma outra estrutura de validação, agora em forma de condicional, onde caso o vídeo seja reproduzido com sucesso, são lidas as coordenadas guardadas em bbox, que anteriormente ao executar nossa função detector_pessoas( ) sempre ao final do processo recebia o retorno dos valores de x, y, l ,a. Agora esses dados são lidos por nosso laço for em v, repassando esses dados para x, y, w, h via list comprehension.

Dando prosseguimento é criado o bloco de código responsável por desenhar a bounding box em si.

Também definimos um else, para caso as condições anteriores não sejam atingidas, nossa função rastreador_pessoas( ) é novamente iniciada porém agora usando de um rastreador diferente, nesse caso MOSSE, com sua função inicializadora instanciada em nosso rastreador.

Por fim, apenas realizamos a codificação necessária para gerar a janela de reprodução do vídeo e o gatilho para encerramento do mesmo por meio da tecla Esc.

Caso não tenha ocorrido nenhum erro de sintaxe, ao executar nosso código é exibida a primeira tela, com o vídeo pausado, onde é

possível ver que a detecção da pessoa foi realizada corretamente.

Ao pressionar qualquer tecla é possível assistir ao vídeo onde, nesse caso, por todo percurso dessa pessoa caminhando, a mesma possui uma caixa de marcação que a acompanha.

Neste processo, de acordo com a integridade do vídeo, caso todos frames sejam processados corretamente, não haverá perda de quadros. Lembre-se que configuramos o código de tal forma que caso nosso rastreador inicial não consiga realizar o rastreamento, um rastreador alternativo é inicializado.

Apenas em casos extremos como se o vídeo estiver corrompido em alguns pontos, seus quadros serão perdidos, mas de forma geral, de acordo com todas estruturas de validação criadas anteriormente, se o arquivo de vídeo foi carregado e reproduzido corretamente, a tendência é que tenhamos todos seus frames processados e reproduzidos corretamente.

Rastreamento de Objetos via Webcam

Partindo para os tópicos finais deste capítulo, não podemos deixar de entender como se dá o processo de rastreamento de um objeto a partir de uma webcam.

Visando exemplificar outras possibilidades existentes para nossos dispositivos de entrada, faremos integração de outras funcionalidades do Python para tal fim.

Uma particularidade que deverá ser levada em consideração para este exemplo é que, diferente da detecção via webcam, onde cada frame é processado apenas para detectar e demarcar o objeto, no âmbito de rastreamento sobre tal tipo de dado de entrada, para obtenção de melhores resultados até mesmo o padrão de cores utilizado é diferente do padrão RGB.

Raciocine que diferentemente de um vídeo, onde cada frame subsequente é compatível com o anterior em quase todos aspectos, enquanto que quando estamos realizando a captura usando como fonte uma webcam, devemos contornar problemas nativos deste tipo de dispositivo como perda e reajuste de foco, ruídos gerados por má iluminação, espaços de cores distorcidos, saturação, etc...

Tenha em mente que, ao contrário dos exemplos anteriores onde nossa fonte era lida em três canais de cor, sendo convertida para escala de cinza para alguns processamentos, agora o algoritmo utilizado fará o rastreamento de um determinado objeto com base em sua densidade de pixels próximos a uma cor do objeto.

Apenas como exemplo, imagine o rastreamento de um rosto onde a pessoa em frente a webcam se move para lugares onde haverá diferenças em como a iluminação interage com sua pele, por meio da densidade de pixels o rastreador consegue seguir identificando cada região do objeto compensando tais problemas.

É perfeitamente possível usar dos métodos anteriores para esse mesmo fim, porém, iremos fazer uma abordagem diferente onde para este propósito teremos resultados melhores quando comparado aos métodos anteriormente aplicados para outros exemplos.

```
1  import cv2
2  import time
3  from imutils.video import VideoStream
4  import matplotlib.pyplot as plt
5
```

E eis que mais uma vez o processo se inicia com as importações das bibliotecas módulos e pacotes que serão utilizados em nosso exemplo.

Dessa vez temos duas particularidades, primeira delas a importação da biblioteca time que oferece alguns recursos de sincronização com o relógio do sistema.

Também realizamos a importação do pacote VideoStream do módulo vídeo da biblioteca ImUtils, pois faremos uma abordagem diferente no processo de reprodução do vídeo em tempo real da webcam.

```
1  captura = VideoStream(src = 0).start()
2  time.sleep(1.0)
3  captura = cv2.VideoCapture(0)
4  ok, frame = captura.read()
5
```

Dando sequência criamos uma variável de nome captura, que carrega as funções de VideoStream( ), usando do dispositivo de captura padrão do sistema, inicializando o mesmo por meio da função start( ).

Em seguida executamos a função time.sleep( ) parametrizada com 1.0. Este procedimento é feito pois quando lemos dados a partir de uma webcam, nos primeiros momentos a qual inicia, normalmente leva um certo tempo até realizar seus ajustes de captura quanto ao foco e iluminação do ambiente.

Sendo assim, por meio da função sleep( ) que realiza uma pausa com tempo predeterminado, programamos a mesma para quando inicializada a webcam, que se espere um segundo até poder dar sequência na execução das demais linhas de código.

Na sequência nossa variável captura chama a função cv2.VideoCapture( ), onde para as variáveis ok e frame são repassados os dados de se a webcam foi inicializada corretamente e seu conteúdo propriamente dito do primeiro frame lido, respectivamente.

```
1  bbox = cv2.selectROI(frame, False)
2  x, y, l ,a = bbox
3  identificador = (x, y, l, a)
4  print(identificador)
5
6  roi = frame[y: y + a, x: x + l]
7  cor_roi = cv2.cvtColor(roi, cv2.COLOR_BGR2HSV)
8  cv2.imshow('Regiao de Interesse em HSV', cor_roi)
9
```

Dando continuidade, criamos as estruturas responsáveis pela criação das bounding boxes e sua aplicação sobre o objeto em questão no frame.

Para isso, inicialmente criamos nossa variável bbox que atribuída para si tem a função cv2.selectROI( ), parametrizada com frame. Lembre-se que por meio dessa função teremos a projeção de um frame onde desenharemos manualmente a bounding box sobre o que quisermos rastrear.

Também criamos as variáveis x, y, l e a, responsáveis por guardar os dados de localização dos pontos de interesse.

Complementar a essas variáveis, criamos uma variável de nome identificador que por sua vez recebe os dados das coordenadas de x, y, l e a. Por meio da função print( ), parametrizando a mesma com os dados de identificador, é possível ver via terminal as posições dos pixels que demarcam a primeira bounding box.

Na sequência criamos a variável roi, que recebe de frame apenas o conteúdo da região de interesse, como um recorte da imagem para que esta seja usada como máscara para as funções de rastreamento.

Também criamos uma variável de nome cor_roi que converte o conteúdo de roi do padrão RGB para HSV por meio da função cv2.cvtColor( ) parametrizada com os dados de roi e com o método cv2.COLOR_BGR2HSV. Lembre-se que como mencionado no início deste capítulo, por uma questão de eficiência, não estaremos rastreando um objeto por meio de cores ou padrões de cinza, mas sim por sua densidade de pixels via escala HSV.

Por meio da função cv2.imshow( ) é possível gerar a projeção de cor_roi para entender de fato o que é a chamada leitura por densidade de pixels.

```
1   histograma_roi = cv2.calcHist([cor_roi],
2                                 [0],
3                                 None,
4                                 [180],
5                                 [0,180])
6   histograma_roi = cv2.normalize(histograma_roi,
7                                  histograma_roi,
8                                  0,
9                                  255,
10                                 cv2.NORM_MINMAX)
11  crit_termino = (cv2.TermCriteria_EPS | cv2.TermCriteria_COUNT,
12                  10,
13                  1)
14
```

Em seguida são realizadas as gerações de histogramas a partir da imagem capturada em densidade de pixels para cor_roi. Nos tópicos iniciais do livro já vimos algumas meios de obtenção e plotagem de histogramas, agora, apenas para fins de exemplo, usaremos de outros meios alternativos para esse mesmo propósito.

Sendo assim, primeiro criamos uma variável de nome histograma_roi que por meio da função cv2.calcHist( ) irá gerar este tipo de dado referência para entendermos do objeto e seus pixels

projetados em densidade, quais as cores mais persistentes no objeto, sendo utilizadas para o rastreamento.

Como parâmetros desta função, passamos alguns dados como a fonte, nesse caso cor_roi, e fatores da cor e amplitude de canais lidos como referência.

Da mesma forma, realizamos a normalização destes dados matriciais para uma escala entre 0 e 1, apenas por uma questão de performance. Aqui, por meio da função cv2.normalize( ) realizamos tal feito atualizando os dados da própria variável histograma_roi, por meio do método cv2.NORM_MINMAX.

Por fim, criamos uma variável de nome crit_termino que define manualmente algumas condições que, quando atingidas, encerram o processo subentendendo que houveram recorrentes erros na captura.

Para isso, crit_termino chama os métodos cv2.TermCriteria_EPS e cv2.TermCriteria_COUNT, onde caso um ou outro retorne True, crit_termino será o gatilho que acionará o término do processo de captura.

```
15  plt.hist(roi.ravel(), 180, [0,180])
16  plt.show()
17
18  cv2.imshow('Regiao de Interesse', roi)
19  cv2.waitKey(0)
20
```

Apenas revisando, a partir do momento em que temos as variáveis dependentes todas declaradas de acordo com seu propósito, podemos plotar o conteúdo das mesmas para de fato visualizar seus dados.

Como realizado logo no capítulo inicial do livro, podemos a qualquer momento gerar um histograma para visualizar, neste caso, o agrupamento dos pixels quanto a sua densidade. Esse processo é feito por meio da função plt.hist( ) combinada a ravel( ).

Além do histograma gerado para os rados de roi, podemos ver o conteúdo de roi simplesmente o passando como parâmetro para função cv2.imshow( ).

```
1   while True:
2       ok, frame = captura.read()
3       if ok == True:
4           hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
5           densidade = cv2.calcBackProject([hsv],
6                                           [0],
7                                           histograma_roi,
8                                           [0, 180],
9                                           1)
10          ok, rastreio = cv2.meanShift(densidade,
11                                       (x, y, l, a),
12                                       crit_termino)
13          x, y, l, a = rastreio
14          cv2.rectangle(frame, (x, y), (x+l, y+a), (0,255,0), 2)
15          cv2.imshow('Rastreamento', frame)
16          cv2.imshow('Densidade', densidade)
17          cv2.imshow('Regiao de Interesse', roi)
18
19          if cv2.waitKey(1) == 27:
20              break
21      else:
22          break
23
```

Partindo para etapa final, do mesmo modo como feito nos outros exemplos, vamos criar a estrutura de repetição que executará as funções encadeadas até que o processo seja encerrado pelo usuário.

Nesse contexto, inicialmente criamos nossa estrutura condicional while True onde indentado para o mesmo, temos de início as variáveis ok e frame que inicializam nosso dispositivo de entrada por meio da função captura.read( )

Na sequência é criada uma estrutura condicional onde enquanto o atributo de ok for True, serão executados alguns blocos de código.

Primeiro deles, onde é criada uma variável de nome hsv, que realiza a conversão do padrão de cores original, normalmente RGB

para o formato HSV, referente a cores relacionadas a densidade de pixels por meio da função cv2.cvtColor( ) parametrizada com o conteúdo de frame, executando o método cvt.COLOR_BGR2HSV.

Logo após é declarada uma variável de nome densidade que possui atribuída para si a função cv2.calcBackProject( ) por sua vez parametrizada com o conteúdo de hsv, e de histograma_roi, aqui, internamente serão feitas algumas correlações entre os dados de hsv e do histograma, equiparando os mesmos dentro do intervalo estabelecido (0 a 180).

De forma parecida, são criadas as variáveis ok e rastreio, onde para rastreio temos a atribuição da função cv2.meanShift( ), parametrizado com os dados de densidade, coordenadas x, y, l e a da região de interesse e com os dados de crit_termino. Desse modo está criada toda cadeia de leitura e reprodução do dispositivo de captura, em tempo real, projetando seus dados da forma como foram interpretados.

Também são atualizados os valores para x, y, l e a com os dados oriundos de rastreio.

Por fim, por meio da função cv2.rectangle( ) geramos a bounding box que será aplicada frame a frame sobre a região de interesse.

Podemos realizar algumas codificações adicionais, apenas como exemplo, exibindo o conteúdo de frame, de densidade e de roi para que fique mais claro todo processo de conversões que foram realizados.

Encerrando definimos que o usuário pode encerrar o processo (que executará em loop inifinito) simplesmente pressionando a tecla Esc.

Finalizando, são chamadas as funções para encerrar todos os processos e liberar a webcam, por meio das funções cv2.destroyAllWindows( ) e captura.release( ), respectivamente.

Caso não haja nenhum erro de sintaxe, ao executar nosso código teremos os retornos de todas funções print( ) além, é claro,

do rastreamento sendo realizado em tempo real via webcam.

Primeiro frame, onde desenhamos manualmente qual a região de interesse a se rastreada.

Histograma com equivalência de pixels de mesma cor de acordo com a densidade, em uma escala de 0 a 180.

Região de interesse mapeada.

Projeção da densidade de pixels.

E por fim a janela com o rastreamento sendo exibido em tempo real. Aqui podemos ver, inclusive, que da região de interesse demarcada, em função da densidade de pixels, o ponto central usado como referência é o meio da máscara, deixando outros pontos dentro da caixa de marcação menos relevantes pois nos mesmos existe muita variação da densidade de pixels.

Independentemente do tipo de objeto essa métrica será seguida, de toda a caixa de marcação o algoritmo irá escolher a região onde há maior concentração de pixels de mesma cor para usar este pondo como referência, obtendo melhor desempenho no processo de rastreamento.

Uma das grandes limitações que você deve ter percebido sobre este exemplo é que o mesmo realiza o mapeamento do conteúdo interno a uma caixa de marcação, usando de densidade de pixels para melhor rastreamento, e em função disso fica limitado a essa máscara que tem tamanho fixo em todo processo de rastreamento.

O interessante quando falamos de rastreamento é que o objeto em questão seja facilmente rastreado por todo campo da captura, não somente em seus eixos de largura e altura, mas também em profundidade.

Assim como realizamos uma codificação específica para dar tempo da webcam ser inicializada e adaptada as condições de iluminação do ambiente, focando seu objeto, uma prática interessante é adaptarmos nosso algoritmo para que o objeto seja rastreado também em diferenças de profundidade em relação ao “cenário”.

Raciocine que no processo de um objeto se aproximar ou se afastar da câmera, a bounding box deve se ampliar e/ou reduzir compensando a posição do objeto, o que não era previsto no modelo anterior e que pode fazer grande diferença em uma implementação real.

Basicamente o que faremos é uma adaptação do exemplo anterior para que frame a frame seja levada em consideração o tamanho do objeto, prevendo seu comportamento de aproximação ou distanciamento, escalonando a região de interesse proporcional ao mesmo.

```python
import numpy as np
import cv2
from imutils.video import VideoStream
import time

captura = VideoStream(src=0).start()
time.sleep(1.0)

captura = cv2.VideoCapture(0)
ok, frame = captura.read()
bbox = cv2.selectROI(frame, False)
x, y, w, h = bbox
rastreio = (x, y, w, h)
roi = frame[y:y+h, x:x+w]
hsv_roi = cv2.cvtColor(roi, cv2.COLOR_BGR2HSV)
roi_hist = cv2.calcHist([hsv_roi], [0], None, [180], [0, 180])
roi_hist = cv2.normalize(roi_hist, roi_hist, 0, 255, cv2.NORM_MINMAX)
term_crit = (cv2.TERM_CRITERIA_EPS | cv2.TERM_CRITERIA_COUNT, 10, 1)

while True:
    ok, frame = captura.read()
    if ok == True:
        hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
        densidade = cv2.calcBackProject([hsv], [0], roi_hist, [0, 180], 1)

        ok, rastreio = cv2.CamShift(densidade, (x, y, w, h), term_crit)
        pts = cv2.boxPoints(ok)
        pts = np.int0(pts)
        img2 = cv2.polylines(frame, [pts], True, 255, 2)

        cv2.imshow('Camshift Rastreado', img2)
        if cv2.waitKey(1) == 13:
            break
    else:
        break
cv2.destroyAllWindows()
captura.release()
```

Reaproveitando praticamente todo código anterior, única grande diferença se dá pela substituição da função cv2.meanShift( ) pela cv2.CamShift( ), onde agora por meio deste algoritmo se fará o rastreamento.

Também é retrabalhado a imagem gerada como retorno, utilizada como frame. Para isso, criamos uma variável de nome pts

que chama a função cv2.boxPoints( ) parametrizada com o estado de ok.

Logo após convertemos os dados de pts para int, por meio da função np.int0( ). Esta conversão se faz necessária para que a imagem a nível de matriz possa sofrer escalonamento em tempo real.

Por fim, é criada uma variável de nome img2 que executa a função cv2.polylines( ) parametrizando a mesma com os dados de frame, pts, gerando uma nova bounding box no canal 255 com espessura de borda de 2 pixels.

Raciocine que este processo, por meio do algoritmo CamShift, realiza internamente a leitura da imagem tanto pela sua densidade de pixels quanto por sua forma em polígonos, uma vez que para escalonar a região de interesse e a caixa de marcação é necessário usar destes dois parâmetros para identificar e rastrear o objeto.

Executando nosso código, é realizada a projeção de uma janela para que manualmente realizemos a marcação do objeto a ser rastreado.

Após a marcação, pressionando a tecla Espaço é possível ver em tempo real a captura da webcam, assim como a projeção da bounding box em torno do objeto selecionado, independentemente de sua posição.

Rastreando o Deslocamento de Objetos

Finalizando este capítulo, vamos usar de um exemplo onde é possível não somente rastrear um objeto, mas obter o tracejado de todo seu deslocamento.

Essa é uma das aplicações mais comuns em circuitos de monitoramento, seja de passagens de pessoas, seja de garagens, seja onde for necessário entender o deslocamento de um objeto de um ponto A até um ponto B.

Este mesmo algoritmo também é muito útil para câmeras de segurança de vias públicas, onde é possível ver todo o caminho percorrido por veículos quando os mesmos são parte de um acidente, sendo possível reconstruir o comportamento dos mesmos e entender como se deu o acidente de trânsito.

O algoritmo em questão, comumente chamado de Fluxo Óptico, assim como todos os outros apresentados anteriormente, nada mais é do que uma série de estudos de cientistas da computação visando criar algoritmos de rastreamento.

Nesse caso, usaremos de algumas funções que acabaram por ser implementadas na biblioteca OpenCV para uso fácil e efetivo, porém, com uma rápida pesquisa é possível encontrar artigos científicos com outras propostas que tem como objetivo esta mesma finalidade.

Apenas realizando um adendo, raciocine que, uma vez que estamos trabalhando com todo e qualquer tipo de imagem, basicamente estamos trabalhando com funções que internamente processam os dados da imagem em forma de matriz. Cada elemento vetor que compõe uma matriz pode ser abstraído como um vetor na vida real, traduzindo seus dados/valores para informações de direção, movimento e deslocamento.

É exatamente isto que é feito a partir de um algoritmo de fluxo óptico, uma vez que por meio dos dados de um frame, em escala de densidade de pixels, podemos colocar uma máscara fixa sobre a matriz com pontos pré-determinados, onde monitorando tais pontos, podemos entender o movimento da cena acompanhando o deslocamento de cada pixel, sua direção, velocidade e intensidade de cor (no caso de rastreamento de objeto específico).

```
1  import cv2
2  import numpy as np
3
```

Como de costume, iniciamos nossa codificação com as devidas importações das bibliotecas, módulos e pacotes que se farão necessários ao longo de nosso exemplo.

```
1  criterios = cv2.TERM_CRITERIA_EPS | cv2.TERM_CRITERIA_COUNT
2
```

Na sequência criamos uma variável de nome critérios, que será usada logo em seguida, como atributo para esta variável colocamos dois métodos da biblioteca OpenCV que armazenam valores padrão de critérios de encerramento quando usamos de dispositivos de

entrada com captura contínua como uma webcam por exemplo. Esses critérios já foram usados algumas vezes nos exemplos anteriores, logo, não vamos explicá-los desnecessariamente.

```
1  captura = cv2.VideoCapture("pessoas.mp4")
2
```

Como de costume criamos nossa variável de nome captura que por meio da função cv2.VideoCapture( ) realiza a leitura de nosso arquivo pessoas.mp4.

```
1  parametros1 = dict(maxCorners = 50,
2                     qualityLevel = 0.3,
3                     minDistance = 7)
4  parametros2 = dict(winSize = (15, 15),
5                     maxLevel = 2,
6                     criteria = (criterios, 10, 0.03))
7
```

Em seguida são criadas duas variáveis que receberão em forma de dicionário alguns parâmetros a serem repassados posteriormente para as funções de rastreamento via fluxo óptico. Para parametros 2 basicamente definimos o número de cantos detectáveis, nível de qualidade (no sentido de quão relevantes serão os objetos detectados por sua densidade de pixels) e a distância média a ser considerada, pois em certos casos, a perspectiva de um objeto se aproximando ou se afastando deve ser mapeada.

Da mesma forma é criada a variável parametros2 que em forma de dicionário define o tamanho da janela ao qual o vetor se projetará, parâmetro relevante para rastreio de um objeto quando o mesmo avança diferentes posições na matriz de acordo com seu movimento, level máximo de detecções sobrepostas, por fim, chamando os métodos de critério de término atribuídos anteriormente para variável critérios.

```
canais = np.random.randint(0, 255, (100, 3))
ok, frame = captura.read()
frame_cinza = cv2.cvtColor(frame,
                           cv2.COLOR_BGR2GRAY)
cantos = cv2.goodFeaturesToTrack(frame_cinza,
                                 mask = None,
                                 **parametros1)
mascara = np.zeros_like(frame)

```

Dando continuidade criamos mais algumas variáveis que definem certas condições, chamam funções e inicializam alguns processos do rastreamento.

Primeira delas de nome canais que por meio da função np.random.randint( ) gera de acordo com o parâmetro, números aleatórios no intervalo de 0 a 255, cem vezes, distribuídos em 3 canais de cor. Lembrando que aqui não estamos usando cores propriamente ditas, mas canais de densidades de pixels.

Logo após criamos as variáveis ok e frame que inicializam a leitura do dispositivo de entrada por meio da função captura.read( ). Salientando que aqui, para este exemplo, estamos lendo um arquivo de vídeo.

Também é realizada a conversão de frame para frame_cinza, por meio da função cv2.cvtColor( ) e do método cv2.COLOR_BGR2GRAY.

Na sequência é criada a variável cantos, que por meio da função cv2.goodFeaturesToTrack( ) realizará a detecção de formas e ângulos possivelmente de cantos de objetos. Para esta função são repassados como parâmetros os dados de frame_cinza e os parâmetros definidos anteriormente no dicionário parametros1.

Por fim, é criada uma máscara matricial inicialmente composta apenas de zeros por meio da função np.zeros_like( ), parametrizada com frame. Dessa forma, será criada uma máscara com um dado/vetor para cada pixel, do mesmo tamanho e proporção da matriz original de frame.

```
while True:
    ok, frame = captura.read()
    img_cinza = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    cantos2, status, erros = cv2.calcOpticalFlowPyrLK(frame_cinza,
                                                      img_cinza,
                                                      cantos,
                                                      None,
                                                      **parametros2)
    novas_arestas = cantos2[status == 1]
    velhas_arestas = cantos[status == 1]

```

Partindo para a etapa final, é criado o bloco de código onde todo processo ocorrerá em forma de loop de repetição até que o usuário acione algum gatilho.

Sendo assim, como já feitas algumas vezes anteriormente, é criada a estrutura de repetição while True, onde indentado para a mesma inicialmente temos as variáveis ok e frame, agora inicializando a reprodução do vídeo por meio da função captura.read( ).

Em seguida é realizada a conversão de cor para cinza do conteúdo de frame.

Na sequência são criadas as variáveis cantos2, status e erros, que recebem dados da função cv2.calcOpticalFlowPyrLK( ), por sua vez parametrizada com os dados de frame_cinza, img_cinza, cantos e parametros2. Nesta etapa, por intermédio dessa função que ocorrerá o rastreamento sobre o vídeo em reprodução.

Dando continuidade é realizada uma codificação adicional onde são criadas as variáveis novas_arestas e velhas_arestas, respectivamente atualizando cantose e cantos na função cv2.calcOpticalFlowPyrLK( ) em seu status, dessa forma, enquanto houver status, o vídeo continua sendo reproduzido assim como executada a função sobre o mesmo. A partir do momento em que o vídeo é encerrado, toda cadeia de variáveis atreladas a esta função retornarão ao seu estado inicial.

```
12      for i, (novas, velhas) in enumerate(zip(novas_arestas,
13                                              velhas_arestas)):
14          a, b = novas.ravel()
15          c, d = velhas.ravel()
16          mascara = cv2.line(mascara,(a, b),(c, d),canais[i].tolist(),2)
17          frame = cv2.circle(frame, (a,b), 5, canais[i].tolist(), -1)
18
19      img = cv2.add(frame, mascara)
20      cv2.imshow('Fluxo Optico', img)
21      if cv2.waitKey(1) == 13:
22          break
23
```

Ainda indentado a nossa estrutura de repetição while True criamos um laço de repetição for que com a variável temporária i e (novas, velhas) percorre os valores de novas_arestas e velhas_arestas.

Dessa maneira são criadas as variáveis a e b que chamam a função novas.ravel( ), enquanto as variáveis c e d chamam a função velhas.ravel( ). Apenas relembrando que a função ravel( ) nos gera um histograma da imagem, onde tais dados sobre a densidade de pixels serão relevantes para o rastreamento.

Em seguida para a variável mascara é atribuída a função cv2.line( ) parametrizada com si própria como variável de retorno, dos dados de a, b, c e d, assim como canais em posição i (já que este valor é gerado aleatoriamente). Em outras palavras aqui estaremos desenhando o tracejado por onde os objetos identificados passaram, em forma de linha.

De forma parecida para frame é atribuída a função cv2.circle( ), parametrizada com basicamente os mesmos dados. A diferença se dará porque aqui serão demarcados com um pequeno círculo os objetos em questão que foram identificados quanto ao seu movimento.

Dessa forma, o algoritmo detectará alguns objetos e colocará sobre os mesmos um ponto de marcação sobre uma região de interesse facilmente rastreável de acordo com a densidade de pixels, conforme o objeto se desloca, este ponto acompanhará o

ponto de interesse junto ao objeto e no caminho percorrido será desenhada uma linha.

Finalizando essa parte simplesmente é gerada a janela para reprodução do vídeo assim como o é criado o gatilho para que a mesma possa ser encerrada manualmente pelo usuário.

```
24      frame_cinza = img_cinza.copy()
25      cantos = novas_arestas.reshape(-1, 1, 2)
26
27  cv2.destroyAllWindows()
28  captura.release()
29
```

Finalizando todo o processo é criada uma codificação adicional apenas para gerar uma cópia de segurança de img_cinza, atribuindo seu conteúdo para frame_cinza, assim como cantos tem seus dados atualizados com o conteúdo de novas_arestas, por meio da função np.reshape( ) apenas para que tais dados, quando retroalimentados, sejam repassados com escala de cor e formato compatíveis com o esperado.

Por fim são chamadas as funções para que se encerre todo e qualquer processo envolvido, inclusive liberando o dispositivo fonte de entrada para que o mesmo seja descarregado da memória.

Como esperado, caso não haja nenhum erro de sintaxe, a partir do momento em que o código é executado o vídeo é aberto, nele é possível ver pontos sobre os objetos detectados de acordo com seu movimento inicial. Note que temos pontos sobre algumas pessoas assim como sobre alguns galhos das árvores que estavam de fato se movendo.

À medida que os objetos se deslocam na cela é possível ver facilmente o ponto de marcação assim como o tracejado de seu percurso de deslocamento.

RECONHECIMENTO DE FACES VIA CNN

Eis que chegamos em um dos capítulos mais interessantes (se não o mais) deste livro, onde é possível combinar um pouco de tudo o que foi visto até então para que somando todo este conhecimento possamos criar uma robusta e eficiente ferramenta de reconhecimento de faces.

Em nosso exemplo iremos processar uma imagem, detectando um determinado rosto, extraindo características do mesmo para que tais dados alimentem uma rede neural artificial convolucional que irá aprender a reconhecer tal rosto, tendo plena capacidade de identificar e reconhecer o mesmo independentemente do contexto.

Diferentemente dos outros exemplos que usam de redes neurais artificiais, aqui faremos o uso de uma rede neural convolucional totalmente pré-programada, de modo que bastará instanciarmos a mesma ao longo de nosso código assim como uma função qualquer. Embora não seja o foco deste livro, nos próximos criaremos passo a passo uma rede neural convolucional entendendo de forma prática seu funcionamento.

Importante salientar que, para este exemplo em particular, criaremos três estruturas de código, embora parecidas, diferentes e separadas em arquivos diferentes não modularizados.

A primeira delas será responsável pelo processo de detecção do rosto e o mapeamento dos pontos característicos que o tornam único entre os demais rostos.

Já na segunda parte iremos treinar nossa rede neural com uma pequena base de dados de imagens com diferentes fotografias da mesma pessoa, para que a mesma detecte e aprenda todos os padrões de características únicas daquela pessoa.

Por fim, na terceira parte estaremos submetendo nossa rede neural a um teste, onde submeteremos para análise uma foto completamente diferente das usadas para treino, onde a rede irá identificar todos os rostos que compõe a cena, comparando cada um e suas características com sua base treinada, identificando corretamente apenas a pessoa em questão.

Inicialmente temos de dar uma atenção à organização de nossos arquivos. Quando estamos falando do uso de redes neurais artificiais convolucionais, precisamos criar uma base de dados com imagens divididas em imagens para treino e para teste.

Em nosso caso, no diretório raiz de nosso projeto criamos uma pasta chamada img_treino, dentro da mesma temos 11 fotos de enquadramento do rosto do ator Silvester Stallone.

Para nosso exemplo, apenas 11 imagens serão suficientes, porém raciocine que, dada uma maior complexidade ao projeto, como reconhecimento de várias pessoas, ou vários objetos, uma base de dados maior para treino da rede neural se faz necessária.

Aqui, como apenas reconheceremos as características do rosto deste ator por meio da extração de pontos, apenas 11 imagens serão suficientes.

A partir disto podemos partir para o código!

```
1  import cv2
2  import dlib
3  import numpy as np
4
```

Inicialmente realizamos as importações das bibliotecas OpenCV, Dlib e Numpy por meio dos comandos acima.

```
1  fonte = cv2.FONT_HERSHEY_COMPLEX_SMALL
2
```

Em seguida realizamos uma pequena codificação onde declaramos uma variável chamada fonte que por sua vez instancia uma fonte padrão da biblioteca OpenCV, pelo método cv2.FONT_HERSHEY_COMPLEX_SMALL.

```
1  detector_faces = dlib.get_frontal_face_detector()
2  detector_pontos = dlib.shape_predictor('shape_predictor_68_face_landmarks.dat')
3
```

Na sequência criamos uma variável de nome detector_faces que como atributo para si possui a função dlib.get_frontal_face_detector( ).

De forma parecida criamos a variável detector_pontos, que por sua vez chama a função dlib.shape_predictor( ) parametrizada com o arquivo shape_predictor_68_face_landmarks.dat.

Essa função realiza um processo muito semelhante a aplicação de máscara de 68 pontos da biblioteca OpenCV, onde temos pontos de marcação para as estruturas que compõe a face e a ideia é que tais pontos sejam identificados e aplicados sobre nossas imagens de treino.

```
1  imagem = cv2.imread('stallone.jpg')
2  faces = detector_faces(imagem, 2)
3
```

Dando sequência criamos nossa variável imagem que por meio da função cv2.imread( ) realiza a leitura de nosso arquivo stallone.jpg.

Também é criada a variável faces, que instancia nosso detector_faces (que possui a função dlib.get_frontal_face_detector( )), parametrizando a mesma com o conteúdo de imagem, onde o segundo parâmetro define a escala da imagem, sendo 2 o dobro do tamanho da imagem original, apenas por convenção, para que seja mais fácil a identificação dos pontos faciais.

```
1   def pontos_faciais(imagem, pontosFaciais):
2       for ponto in pontosFaciais.parts():
3           cv2.circle(imagem,
4                      (ponto.x, ponto.y),
5                      2, (0,255,0), 2)
6       for num, coord in enumerate(pontosFaciais.parts()):
7           cv2.putText(imagem,
8                       str(num),
9                       (coord.x, coord.y),
10                      fonte, .55,
11                      (0,0,255), 1)
12
```

Dando sequência criamos nossa função pontos_faciais( ) que receberá como parâmetros dados de uma imagem e dados de pontosFaciais.

Dentro do corpo da função, inicialmente criamos um laço for que percorrerá cada ponto em cada parte de pontosFaciais, desenhando um circulo para demarcação deste ponto por meio da função cv2.circle( ). Note que como parâmetros para esta função passamos o conteúdo de imagem, ponto em seu eixo x e ponto em seu eixo y, e isso se faz necessário pois como estamos lendo o conteúdo de pontosFaciais em partes para que os mesmos sejam repassados para as funções da biblioteca Dlib corretamente.

Nesse mesmo contexto, criamos um segundo laço for que percorre com as variáveis temporárias num e coord o número de pontos encontrados pela função enumerate( ) parametrizada com

pontosFaciais.parts( ). Aqui basicamente estamos fazendo um processo complementar ao laço for anterior, numerando sequencialmente cada ponto encontrado.

Para melhor legibilidade, por meio da função cv2.putText( ) inserimos um pequeno texto que irá sobrepor a imagem original.

Dessa forma, após aplicada a máscara sobre o rosto detectado na foto, cada ponto terá sua posição mapeada assim como cada ponto também terá um identificador numerado.

```
1   def mascara(imagem, pontosFaciais):
2       linhas = [[0, 16, False], # queixo
3                 [17, 21, False], # sobrancelha direita
4                 [22, 26, False], # sobrancelha esquerda
5                 [27, 30, False], # linha do nariz
6                 [30, 35, True], # base do nariz
7                 [36, 41, True], # olho esquerdo
8                 [42, 47, True], # olho direito
9                 [48, 59, True], # linha externa labio
10                [60, 67, True]] # linha interna labio
11      for lin in range(0, len(linhas)):
12          linha = []
13          for i in range(linhas[lin][0], linhas[lin][1] + 1):
14              pontoxy = [pontosFaciais.part(i).x,
15                         pontosFaciais.part(i).y]
16              linha.append(pontoxy)
17          linha = np.array(linha)
18          cv2.polylines(imagem, [linha],
19                        linhas[lin][2],
20                        (255,0,0), 2)
21
```

De modo complementar a função anterior criamos uma função de nome mascara( ) que também receberá como parâmetro os dados de imagem e pontosFaciais. Esta função por sua vez realizará um desenho em nossa máscara coincidindo certos pontos com certas características do rosto, desenhando por exemplo, olhos, nariz e boca...

Para isso dentro do corpo da função mascara( ) criamos uma variável de nome linhas que em forma de lista recebe algumas coordenadas referentes aos pontos mapeados e para os mesmos definindo um status True ou False. Basicamente os pontos instanciados como False serão apenas pontos, enquanto os pontos instanciados como True serão ligados e sobre os mesmos será desenhada uma linha.

Repare no comentário da imagem, ao lado de cada sequência de pontos está a descrição de que estrutura da face tais pontos demarcam.

Na sequência é criado um novo laço for, que faz a variável temporária lin percorrer cada ponto de toda a extensão de linhas.

Dentro do laço for inicialmente é criada uma variável de nome linha que inicialmente possui atribuída para si apenas uma lista vazia.

Então é criado um novo laço for dentro do anterior, onde a variável temporária i percorrerá todos os pontos identificados no laço for anterior e atribuídos para lin.

Tais dados, que nada mais são coordenadas de pontos topográficos de estruturas da face, ficarão atribuídos a pontosFaciais que compõe pontoxy. Esses dados são adicionados a nossa lista linha por meio da função append( ).

Em seguida é necessário converter estes dados para o formato de array tipo Numpy, e isto é feito simplesmente chamando a função np.array( ), parametrizando a mesma com o próprio conteúdo de linha. Uma array do tipo numpy é um formato de arquivo matricial indexado onde serão possíveis realizar algumas aplicações de funções da biblioteca Dlib.

Finalizando esta parte, desenhamos as linhas em si por meio da função cv2.polylines( ) parametrizada com nossa imagem, a array linha e linhas nas posições mapeadas anteriormente lin e 2ª coluna.

```
1  for face in faces:
2      pontos = detector_pontos(imagem, face)
3      pontos_faciais(imagem, pontos)
4      mascara(imagem, pontos)
5      print(pontos.parts())
6
```

Fora da indentação das funções anteriores, novamente ao escopo global de nosso arquivo, criamos um laço for que por meio da função temporária face percorrerá cada face encontrada e atribuída a faces.

Em seguida nossa variável pontos chama a função em detector_pontos( ) repassando os dados de imagem e face. Da mesma forma pontos_faciais( ) é instanciado com os dados de imagem e pontos. E do mesmo modo nossa função mascara( ) é executada com os dados de imagem e pontos.

Por fim, por meio da função print( ) exibimos os pontos em si por meio de pontos.parts( ).

```
1  cv2.imshow('Faces', imagem)
2  cv2.waitKey(0)
3  cv2.destroyAllWindows()
4
```

Encerrando esta primeira etapa são chamadas nossas já velhas conhecidas funções para gerar uma janela com resultado final, que espera o gatilho de uma tecla qualquer para encerrar seu processo, na sequência descarregando toda e qualquer informação referente a esse processo da memória.

Faces

Como esperado, temos em nossa janela a máscara aplicada sobre o rosto de Sylvester Stallone, com cada ponto mapeado corretamente, com as linhas desenhando as linhas dos pontos topográficos do rosto.

Uma vez realizada a etapa anterior, podemos partir para a etapa intermediária, onde usaremos destes tipos de dado para treinar uma rede neural artificial convolucional para que aprenda as características únicas do rosto do ator Sylvester Stallone, a ponto de poder reconhecer este rosto a partir de outras imagens.

```
1  import os
2  import glob
3  import _pickle as cPickle
4  import dlib
5  import cv2
6  import numpy as np
7
```

Em um novo arquivo, apenas por convenção nomeado de reconhecimento_treino.py, iniciamos nosso código com algumas importações de bibliotecas adicionais.

Primeira delas não usada até então a biblioteca glob, que nos permitirá a leitura e escrita de nossos arquivos de imagem para que possamos gerar uma base de dados para treino da rede neural.

Outra biblioteca que será de grande importância é a biblioteca Pickle, pois por meio dela iremos exportar nossos dados para reutilização na fase de testes.

Da mesma forma são importadas as bibliotecas já conhecidas de nossos exemplos.

```
1  detector_faces = dlib.get_frontal_face_detector()
2  detector_pontos = dlib.shape_predictor("shape_predictor_68_face_landmarks.dat")
3  reconhecimento_facial = dlib.face_recognition_model_v1("dlib_face_recognition_resnet_model_v1.dat")
4
```

Em seguida criamos nossas variáveis detector_faces e detector_pontos assim como no exemplo anterior. Também criamos uma variável de nome reconhecimento_facial que por sua vez

chama a função dlib.face_recognition_model_v1( ), parametrizada com o arquivo dlib_face_recognition_resnet_model_v1.dat.

Essa função, como o nome sugere, fará o devido reconhecimento da face por meio de funções de convolução processadas em uma rede neural interna pré-treinada para reconhecimento facial de humanos.

```
1  indice = {}
2  idx = 0
3  descritores = None
4
```

Continuando, criamos algumas variáveis que serão instanciadas posteriormente, porém, precisam estar declaradas no escopo global de nosso código para evitar erros.

Primeira delas índice que recebe como atributo um dicionário vazio, idx que recebe o valor 0 e descritores que inicialmente recebe como atributo None, haja visto que em Python não podemos deixar uma variável declarada sem nenhum dado atribuído pois podemos gerar erros.

```
1  for data in glob.glob(os.path.join('img_treino', '*.jpg')):
2      imagem = cv2.imread(data)
3      faces_detectadas = detector_faces(imagem, 1)
4      num_faces_detectadas = len(faces_detectadas)
5
6      if num_faces_detectadas > 1:
7          print(f'ERRO: Mais de uma face na imagem {data}')
8          exit(0)
9      elif num_faces_detectadas < 1:
10         print(f'ERRO: Nenhuma face encontrada em {data}')
11         exit(0)
12
```

Dando continuidade, para iniciarmos o processo de geração de nossa base de dados, usamos inicialmente de um laço for, que percorrerá cada arquivo dentro do diretório img_treino que possuir a extensão .jpg. Para isso, note que em nosso laço for usamos de uma variável temporária de nome data que percorrerá a função

glob.glob( ), parametrizada com os.path.join( ) por sua vez parametrizada com img_treino e *.jpg.

Indentado ao corpo deste laço for temos novamente a variável imagem, que por meio da função cv2.imread( ) lerá cada arquivo encontrado e atribuído para data.

Na sequência a variável faces_detectadas instancia nosso detector_faces, parametrizando o mesmo com o conteúdo de imagem, em sua escala normal.

Também é criada uma variável num_faces_detectadas que por meio da função len( ) contará quantas faces foram detectadas ao todo ao longo desse processo.

Em seguida é criada uma estrutura de validação, por meio da condicional if, onde caso o número de faces detectadas nesse processo seja maior ou menor que 1, é exibida uma mensagem de erro ao usuário. Lembre-se que nesta etapa estamos trabalhando sobre os dados de reconhecimento de face para treinamento, então, cada arquivo deve possuir apenas a face da pessoa ao qual queremos extrair características.

```
13      for face in faces_detectadas:
14          pontos_identificacao = detector_pontos(imagem, face)
15          caracteristicas = reconhecimento_facial.compute_face_descriptor(imagem,
16                                                                          pontos_identificacao)
17          lista_caracteristicas = [df for df in caracteristicas]
18          descritor_facial = np.asarray(lista_caracteristicas,
19                                        dtype=np.float64)
20          descritor_facial = descritor_facial[np.newaxis, :]
21
22          if descritores is None:
23              descritores = descritor_facial
24          else:
25              descritores = np.concatenate((descritores,
26                                            descritor_facial),
27                                           axis=0)
28          indice[idx] = data
29          idx += 1
30
```

Ainda indentado como o laço for anterior, criamos um novo laço for que agora percorre cada face em faces_detectadas.

Dentro desse laço for inicialmente criamos uma variável de nome pontos_identificados, que instancia nosso detector_pontos( ) parametrizando o mesmo com os dados de imagem e de face.

Também é declarada uma variável de nome caracteristicas que para reconhecimento_facial chama a função compute_face_descriptor( ) por sua vez parametrizado com os dados de imagem e de pontos_identificados. Nesta etapa serão realizados alguns mapeamentos para que os pontos identificados em cada imagem tenham o mesmo padrão.

Criamos uma variável de nome lista_características que recebe em forma de lista todos os dados atribuídos em características. Aqui, apenas por convenção, realizamos este laço for em formato de list comprehension para não tornar a indentação do código ainda mais confusa.

Declaramos uma variável de nome descritor_facial que em formato de array numpy, por meio da função np.asarray( ) recebe os dados de lista_caracteristicas. Por uma questão de compatibilidade adicionamos uma coluna ao mesmo por meio do método np.newaxis. Tenha em mente que para poder haver o correto cruzamento entre tais dados os mesmos devem estar em formato matricial equivalente, uma diferença no número de linhas ou de colunas irá gerar erro.

Encerrando esta parte, realizamos uma verificação onde se nossa variável descritores, declarada no início do código, ainda estiver sem dados atribuídos, para a mesma serão atribuídos os dados de descritor_facial. Caso a mesma já possua dados iniciais, terá seu conteúdo atualizado concatenando os dados de descritores e descritor_facial. Raciocine que ao início do processo tal variável de fato não terá nenhum dado atribuído, e a mesma será constantemente retroalimentada com novos dados que deverão atualizar a mesma.

Por fim, para tornar ativo este processo de retroalimentação, índice em sua posição idx recebe o conteúdo de data e idx tem seu valor incrementado em 1.

```
1  np.save("descritores.npy", descritores)
2  with open("indices.pickle", 'wb') as f:
3      cPickle.dump(indice, f)
4
```

Finalizando esta etapa, assim que todas as funções forem executadas e todas imagens tenham sido processadas via rede neural, podemos salvar este arquivo com os dados de aprendizado para que a partir dos mesmos realizemos novos testes.

Sendo assim, por meio da função np.save( ) criamos o arquivo descritores.npy com o conteúdo de descritores.

De forma parecida, abrimos uma sessão onde geramos e abrimos um arquivo por meio da função open( ) parametrizada com índices.pickle e wb, onde dentro dessa sessão, por meio da função cPickle.dump( ) alimentamos este arquivo com os dados de índice.

Caso não haja nenhum erro de leitura de arquivos ou de sintaxe, ao executar este código teremos no diretório raiz de nossos arquivos os arquivos gerados descritores.npy e índices.pickle.

Partindo para a etapa final deste exemplo, a partir do momento em que temos os arquivos de aprendizado gerados pela rede neural, podemos criar uma nova estrutura de código dedicada apenas a realizar testes a partir desses dados.

```
1  import os
2  import glob
3  import _pickle as cPickle
4  import dlib
5  import cv2
6  import numpy as np
7
```

Sendo assim, inicialmente realizamos as mesmas importações como na etapa anterior.

```
1  detector_faces = dlib.get_frontal_face_detector()
2  detector_pontos = dlib.shape_predictor("shape_predictor_68_face_landmarks.dat")
3  reconhecimento_facial = dlib.face_recognition_model_v1("dlib_face_recognition_resnet_model_v1.dat")
4
```

Também criamos nossos detectores exatamente da mesma forma.

```
1  indices = np.load('indices.pickle', allow_pickle=True)
2  descritores = np.load('descritores.npy')
3  limiar = 0.5
4
```

Na sequência declaramos nossa variável índices que por meio da função np.load( ) carrega os dados do arquivo índices.pickle. Dependendo da versão da biblioteca Numpy que você estiver usando, por uma questão de compatibilidade pode ser necessário declarar manualmente o parâmetro allow_pickle=True para que o carregamento deste tipo de arquivo seja permitido.

Do mesmo modo para nossa variável descritores, por intermédio da função np.load( ) carregamos o conteúdo do arquivo descritores.npy.

Também definimos uma variável de nome limiar, com valor inicial 0.5, que nos servirá como parâmetro para realizar testes onde nossa margem de acertos for baixa, ajustando a sensibilidade com que o algoritmo irá processar os dados de aprendizado com a imagem teste visando coincidir pontos e características.

```
1  for data in glob.glob(os.path.join("img_teste", "*.jpg")):
2      imagem = cv2.imread(data)
3      faces = detector_faces(imagem, 1)
4      for face in faces:
5          e, t, d, b = (int(face.left()),
6                        int(face.top()),
7                        int(face.right()),
8                        int(face.bottom()))
9          pontos = detector_pontos(imagem, face)
10         descritor = reconhecimento_facial.compute_face_descriptor(imagem,
11                                                                   pontos)
12         lista_caracteristicas = [fd for fd in descritor]
13         array_descritor = np.asarray(lista_caracteristicas,
14                                      dtype=np.float64)
15         array_descritor = array_descritor[np.newaxis, :]
16
```

Dando sequência criamos de forma muito parecida com a etapa anterior, um laço for que percorrerá as imagens, agora da pasta img_teste. O processo é feito usando do laço for que percorre com a função glob.glob( ) parametrizada com os.path.join( ) parametrizada com img_teste e *.jpg todo e qualquer arquivo situado neste diretório, atribuindo seus dados a variável temporária data.

Na sequência é lida cada imagem por meio da função cv2.imread( ) parametrizada com data e atribuindo seus dados para imagem.

Do mesmo modo é instanciado nosso detector_faces( ) parametrizado com imagem e atribuindo seus dados retornados apara faces.

Em seguida é criado um novo laço for, que percorrerá cada face situada em faces, obtendo suas coordenadas e guardando seus valores em e, t, d e b, respectivamente.

Indentado a este laço for criamos uma variável de nome pontos, que instancia nosso detector_pontos( ) parametrizando o mesmo com os dados oriundos de imagem e face.

Também é criado nosso descritor, que para reconhecimento_facial chama a função compute_face_descriptor( ) parametrizada com imagem e pontos.

Exatamente da mesma forma como no exemplo anterior, essa parte termina gerando dados para lista_características que tem seu conteúdo convertido para array do tipo numpy e atribuído para array_descritor.

Tenha em mente que nessas etapas, os dados lidos para cada imagem de teste terão seus pontos e características mapeados e confrontados com os dados de aprendizado de nossa rede neural. O reconhecimento de um determinado rosto, ao menos neste tipo de processamento, se dará pela quantidade de pontos que coincidem entre os dados treinados e testados.

```
17          pontos_coincidentes = np.linalg.norm(array_descritor - descritores,
18                                               axis=1)
19          print(f'Margem de Coincidência: {pontos_coincidentes}')
20          minimo_permitido = np.argmin(pontos_coincidentes)
21          print(minimo_permitido)
22          margem_minima_permitida = pontos_coincidentes[minimo_permitido]
23          print(margem_minima_permitida)
24
25          if margem_minima_permitida <= limiar:
26              nome = os.path.split(indices[minimo_permitido])[1].split(".")[0]
27          else:
28              nome = ' '
29
30          cv2.rectangle(imagem, (e, t), (d, b), (0, 255, 255), 2)
31          texto = "{} {:.4f}".format(nome, margem_minima_permitida)
32          cv2.putText(imagem, texto, (d, t), cv2.FONT_HERSHEY_TRIPLEX, 0.7, (0
33
```

Dessa forma, ainda indentado ao laço for acima, criamos uma variável de nome pontos_coincidentes que chama a função np.linalg.norm( ) parametrizada com a subtração entre os dados de array_descritor e dscritores.

Este dado retornado pode ser facilmente escalonado e usado como parâmetro para, por exemplo, em um intervalo de 0 a 100% nos dizer o quanto o rosto reconhecido na imagem de teste é condizente com os rostos usados para treino.

Por meio da função print( ) criamos um simples retorno indicando qual é esta margem de coincidência alcançada nos testes.

Também podemos realizar algumas codificações adicionais como a declaração de uma variável de nome mínimo_permitido que

por meio da função np.argmin( ) pode retornar o valor mais discrepante encontrado nos testes.

Da mesma forma podemos usar apenas o valor de mínimo_permitido encontrado em pontos_coincidentes para definir uma margem_minima_permitida, retornando ao usuário estes dados.

Finalizando esta parte, criamos uma estrutura condicional onde caso o valor de margem_minima_permitida seja igual ou menor ao definido no início de nosso código para limiar (0.5 representando 50%), é retornado como dado o nome da pessoa em questão com base no nome dos arquivos, e para os outros rostos que possam estar na mesma cena, nenhum nome.

Por fim, caso o processo anterior funcione corretamente como esperado, desenhamos uma caixa de marcação sobre os rostos detectados por meio da função cv2.rectangle( ), assim como para o rosto com melhor pontuação, marcado como o rosto reconhecido dentre os outros, colocamos uma simples legenda com o nome da pessoa em questão obtida na estrutura condicional anterior.

Executando nosso código, como esperado, a maioria dos rostos foram detectados corretamente, destes rostos, o com pontuação maior na margem de coincidência era de fato o do ator Sylvester Stallone, desse modo, ao lado de sua caixa de marcação, além da pontuação está uma pequena legenda com seu nome baseado nos arquivos de treino.

```
reconhecimento_teste
C:\Users\Fernando\anaconda3\envs\ReconhecimentoDlib\python.exe C:/Use
Margem de Coincidência: [0.90864681 0.89868876 0.90759447 0.89796222
 0.92405652 0.90223824 0.91437551 0.92572902 0.9359022 ]
3
0.8979622238987085
Margem de Coincidência: [0.83263275 0.76743064 0.78911124 0.71672352
 0.76859254 0.75408598 0.79002979 0.7215465  0.80229227]
3
0.7167235219605709
Margem de Coincidência: [0.91603431 0.89909694 0.9426429  0.91743077
 0.91421258 0.94706438 0.95226368 0.90778728 0.93809922]
1
0.8990969420022907
Margem de Coincidência: [0.75623943 0.72553733 0.81625517 0.71976962
 0.74211845 0.77328113 0.77508911 0.67972897 0.79166497]
9
0.6797289699451916
Margem de Coincidência: [0.72405281 0.72219307 0.78776839 0.72178733
```

Via terminal é possível ver o log onde vemos detalhadamente a margem de coincidência entre o mapa de características da imagem de teste quando comparado com os dados de aprendizado salvos em nosso descritor.npy.

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 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”.

Raciocine que nos próximos exemplos sim estaremos trabalhando com modelos de redes neurais convolucionais, mas modelos voltados a visão computacional, onde criaremos manualmente, passo a passo, arquiteturas de redes neurais artificiais capazes de detectar, reconhecer objetos e aprender com esse processo.

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.

* Este capítulo possui uma formatação um pouco diferente dos demais, pois aqui estaremos por uma questão didática trabalhando diretamente na IDE Spyder, sendo assim, as capturas das telas estão em tamanho e formato diferente.

Reconhecimento de Dígitos Manuscritos

Para darmos início a esta parte dos estudos de redes neurais artificiais convolucionais, 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.

Digits Dataset.py

```
1 from sklearn import datasets
2
```

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.

```
print(base.images[0])

```

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.

```
import matplotlib.pyplot as plt

```

Partindo para o que interessa, vamos fazer a plotagem deste número. Para isto, como de praxe, importamos a matplotlib.pyplot.

```
plt.figure(figsize = (2,2))
plt.imshow(base.images[0],
           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 independentemente 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 o módulo 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 Support 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.

```
import numpy as np
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.

```
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
```

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.

```
regressor = LogisticRegression()
regressor.fit(entradas,saidas)
previsor_regl = regressor.predict([rgb2gray(imagem)])
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.

Classificação de Dígitos Manuscritos

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).

```
9 (etreino,streino),(eteste,steste) = mnist.load_data()
10
```

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).

```
11 plt.imshow(etreino[0], cmap = 'gray')
12 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.

```
classificador.compile(loss = 'categorical_crossentropy',
                      optimizer = 'adam',
                      metrics = ['accuracy'])
classificador.fit(etreino,
                  streino,
                  batch_size = 128,
                  epochs = 10,
                  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.

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.

```
1 from keras.datasets import mnist
2 from keras.models import Sequential
3 from keras.layers import Dense, Flatten, Conv2D, MaxPooling2D
4 from keras.utils import np_utils
5 import numpy as np
6 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.

```
8 seed = 5
9 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.

```
11 (etreino,streino), (eteste,steste) = mnist.load_data()
12 entradas = etreino.reshape(etreino.shape[0], 28, 28, 1)
13 entradas = entradas.astype('float32')
14 entradas /= 255
15 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.

```
17 kfold = StratifiedKFold(n_splits = 5, shuffle = True, random_state = seed)
18 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.

```
20 a = np.zeros(5)
21 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.

```
23 for evalcruzada,svalcruzada in kfold.split(entradas,
24                                            np.zeros(shape=(saidas.shape[0],1)
25     classificador = Sequential()
26     classificador.add(Conv2D(32, (3,3), input_shape=(28,28,1), activation='rel
27     classificador.add(MaxPooling2D(pool_size = (2,2)))
28     classificador.add(Flatten())
29     classificador.add(Dense(units = 128, activation = 'relu'))
30     classificador.add(Dense(units = 10, activation = 'softmax'))
31     classificador.compile(loss = 'categorical_crossentropy', optimizer='adam',
32                           metrics = ['accuracy'])
33     classificador.fit(entradas[evalcruzada], saidas[evalcruzada],
34                       batch_size = 128, epochs = 5)
35     precisao = classificador.evaluate(entradas[svalcruzada], saidas[svalcruzac
36     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]
resultados	list	5	[0.0875833333333334, 0.08525, 0.083333

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.

Classificação de Imagens de Animais

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.

Nome	Data de modificaç...	Tipo	Tamanho
dataset	19/08/2019 16:05	Pasta de arquivos	
Bichos Dataset.py	19/08/2019 15:24	Arquivo PY	3 KB

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.

```
6 gerador_treino = ImageDataGenerator(rescale = 1.0/255,
7                                     rotation_range = 7,
8                                     horizontal_flip = True,
9                                     shear_range = 0.2,
10                                    height_shift_range = 0.07,
11                                    zoom_range = 0.2)
12 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.

Classificação de Imagens via TensorFlow

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.

```
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)
])
```

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.

```
classificador.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
classificador.fit(etreino,
                  streino,
                  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 o valor real absoluto, mas sim algo muito próximo a 100%)

Dessa forma, sua rede neural via TensorFlow já está treinada e pronta para execução de testes.

DETECÇÃO DE OBJETOS COM YOLO + DARKNET

Uma vez entendidos os conceitos básicos de uma rede neural convolucional, vamos explorar algumas últimas possibilidades, porém agora dentro de modelos proprietários.

Tudo o que foi visto até então foi por meio de código aberto de bibliotecas acessórias ao Python para oferecer ferramentas de visão computacional e redes neurais artificiais, mas é interessante raciocinar que além destas, temos ferramentas excelentes para uso não comercial e/ou restrito para fins de estudo, onde temos acesso as ferramentas apenas em sua usabilidade, e não em suas camadas internas de código.

Nesse contexto usaremos de um dos modelos de visão computacional combinado a redes neurais artificiais chamado YOLO, que associado a rede neural Darknet, é uma das mais avançadas ferramentas para detecção e identificação de objetos.

Documentação disponível em: https://pjreddie.com/darknet/yolo/

Como seu próprio enunciado faz questão de dizer, a rede YOLO é um detector de objetos em tempo real, tanto para imagens quanto para vídeo, onde associada ao COCO Dataset e ao modelo de rede neural Darknet, possui performance e confiabilidade muito superior aos demais métodos e meios disponíveis no mercado.

Documentação disponível em: https://cocodataset.org/

COCO Dataset, por sua vez, é uma base de dados para redes neurais artificiais convolucionais treinada com nada menos que 1,5 milhão de objetos mapeados e classificados em 80 categorias distintas, onde indexados e rotulados existem mais de 250 mil pontos de interesse para identificação dos mais diversos tipos de objeto.

Para este exemplo, estaremos trabalhando diretamente sobre o ambiente do Google Colab, dessa forma, não será necessária a instalação local de nenhuma biblioteca ou ferramenta adicional.

A única configuração necessária é definir o tipo de acelerador para GPU caso o mesmo não esteja definido dessa forma.

```
1  !git clone https://github.com/AlexeyAB/darknet
2
```

```
Cloning into 'darknet'...
remote: Enumerating objects: 16, done.
remote: Counting objects: 100% (16/16), done.
remote: Compressing objects: 100% (9/9), done.
remote: Total 14437 (delta 7), reused 10 (delta 5), pack-reused 14421
Receiving objects: 100% (14437/14437), 13.15 MiB | 4.85 MiB/s, done.
Resolving deltas: 100% (9810/9810), done.
```

Partindo para prática, inicialmente no lugar de importar bibliotecas, módulos e pacotes como estávamos acostumados, iremos clonar a estrutura e conteúdo de um repositório no GitHub.

Por meio do comando acima clonamos para nosso ambiente virtualizado a arquitetura da rede neural artificial darknet.

```
1  ls
2
```

```
darknet/  sample_data/
```

Uma vez realizada a cópia de todos arquivos darknet, por estarmos trabalhando em um ambiente virtualizado baseado em Linux, podemos diretamente das células executar alguns comandos básicos.

Por meio do comando ls nos é retornado que temos dois diretórios base, sendo eles darknet e sample_data.

* Importante salientar que justamente por estarmos usando de uma estrutura proprietária, a usaremos sem realizar qualquer tipo de alteração, seja em nome de arquivos ou diretórios, seja em parâmetros de configuração.

```
cd darknet

/content/darknet
```

Por meio do comando cd darknet entramos na pasta darknet.

```
ls

3rdparty/                darknet.py               net_cam_v3.sh*
build/                   darknet_video.py         net_cam_v4.sh*
build.ps1*               data/                    README.md
build.sh*                image_yolov3.sh*         results/
cfg/                     image_yolov4.sh*         scripts/
cmake/                   include/                 src/
CMakeLists.txt           json_mjpeg_streams.sh*   video_yolov3.sh*
DarknetConfig.cmake.in   LICENSE                  video_yolov4.sh*
darknet_images.py        Makefile
```

Novamente por meio do comando ls podemos ver o conteúdo da pasta darknet.

```
1  !make
2

mkdir -p ./obj/
mkdir -p backup
chmod +x *.sh
g++ -std=c++11 -std=c++11 -Iinclude/ -I3rdparty/stb/include -Wall -Wfa
g++ -std=c++11 -std=c++11 -Iinclude/ -I3rdparty/stb/include -Wall -Wfa
./src/http_stream.cpp: In member function 'bool JSON_sender::write(con
./src/http_stream.cpp:249:21: warning: unused variable 'n' [-Wunused-v
                int n = _write(client, outputbuf, outlen);
                    ^
./src/http_stream.cpp: In function 'void set_track_id(detection*, int,
./src/http_stream.cpp:863:27: warning: comparison between signed and u
        for (int i = 0; i < v.size(); ++i) {
                          ~~^~~~~~~~~~
```

Na sequência é necessário compilar a rede neural darknet para que a mesma possa ser instanciada e executada dentro de nosso ambiente virtualizado. Isto é feito pelo simples comando !make.

Ao executar esta célula será gerado um retorno que nada mais é do que um log do processo de instalação, neste é possível ver alguns avisos porém não é necessário dar atenção a eles, o processo ocorrerá gerando alguns avisos de incompatibilidade mas ao final da compilação o modelo estará pronto para uso.

```
1  !wget https://github.com/AlexeyAB/darknet/releases/download/darknet_yolo_v3_optimal/yolov4.weights
2

--2020-11-20 17:08:48--  https://github.com/AlexeyAB/darknet/releases/download/darknet_yolo_v3_optimal/yolov4.weights
Resolving github.com (github.com)... 140.82.121.4
Connecting to github.com (github.com)|140.82.121.4|:443... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://github-production-release-asset-2e65be.s3.amazonaws.com/75388965/ba4b6380-889c-11ea-9751-f994f5961796?X-Am
--2020-11-20 17:08:48--  https://github-production-release-asset-2e65be.s3.amazonaws.com/75388965/ba4b6380-889c-11ea-9751-f9
Resolving github-production-release-asset-2e65be.s3.amazonaws.com (github-production-release-asset-2e65be.s3.amazonaws.com).
Connecting to github-production-release-asset-2e65be.s3.amazonaws.com (github-production-release-asset-2e65be.s3.amazonaws.c
HTTP request sent, awaiting response... 200 OK
Length: 257717640 (246M) [application/octet-stream]
Saving to: 'yolov4.weights'

yolov4.weights      100%[===================>] 245.78M  36.1MB/s    in 7.5s

2020-11-20 17:08:56 (32.7 MB/s) - 'yolov4.weights' saved [257717640/257717640]
```

Em seguida é necessário baixar o arquivo dos pesos treinados com base no COCO dataset.

Para isso, por meio do comando !wget https://github.com/AlexeyAB/darknet/releases/download/darknet_yolo_v3_optimal/yolov4.weights

é realizado o download do arquivo. Lembre-se que para este exemplo estamos usando de um modelo pronto e pré-treinado para reconhecimento de objetos, por sorte temos disponíveis para uso alguns arquivos de aprendizagem da rede darknet. Neste caso em particular estaremos usando o arquivo treinado para versão 4 da YOLO.

```
from google.colab import files
arquivos = files.upload()

```

```
Browse...  criancas.jpg
criancas.jpg(image/jpeg) - 228278 bytes, last modified: n/a - 100% done
Saving criancas.jpg to criancas.jpg
```

Próxima etapa é carregar de fato o arquivo que será submetido a teste, para este exemplo, por meio do código acima realizamos o carregamento de uma imagem de nome crianças.jpg.

```
ls

```

```
3rdparty/                darknet_images.py        net_cam_v4.sh*
backup/                  darknet.py               obj/
build/                   darknet_video.py         README.md
build.ps1*               data/                    results/
build.sh*                image_yolov3.sh*         scripts/
cfg/                     image_yolov4.sh*         src/
cmake/                   include/                 video_yolov3.sh*
CMakeLists.txt           json_mjpeg_streams.sh*   video_yolov4.sh*
criancas.jpg             LICENSE                  yolov4.weights
darknet*                 Makefile
DarknetConfig.cmake.in   net_cam_v3.sh*
```

Listando novamente o diretório é possível ver que a imagem foi carregada com sucesso.

```
!./darknet detect cfg/yolov4.cfg yolov4.weights criancas.jpg

mini_batch = 1, batch = 8, time_steps = 1, train = 0
   layer   filters  size/strd(dil)      input                output
   0 conv     32       3 x 3/ 1    608 x 608 x   3 ->  608 x 608 x  32 0.639 BF
   1 conv     64       3 x 3/ 2    608 x 608 x  32 ->  304 x 304 x  64 3.407 BF
   2 conv     64       1 x 1/ 1    304 x 304 x  64 ->  304 x 304 x  64 0.757 BF
   3 route  1                                     ->  304 x 304 x  64
   4 conv     64       1 x 1/ 1    304 x 304 x  64 ->  304 x 304 x  64 0.757 BF
   5 conv     32       1 x 1/ 1    304 x 304 x  64 ->  304 x 304 x  32 0.379 BF
   6 conv     64       3 x 3/ 1    304 x 304 x  32 ->  304 x 304 x  64 3.407 BF
   7 Shortcut Layer: 4,  wt = 0, wn = 0, outputs: 304 x 304 x  64 0.006 BF
   8 conv     64       1 x 1/ 1    304 x 304 x  64 ->  304 x 304 x  64 0.757 BF
   9 route  8 2                                   ->  304 x 304 x 128
  10 conv     64       1 x 1/ 1    304 x 304 x 128 ->  304 x 304 x  64 1.514 BF
  11 conv    128       3 x 3/ 2    304 x 304 x  64 ->  152 x 152 x 128 3.407 BF
  12 conv     64       1 x 1/ 1    152 x 152 x 128 ->  152 x 152 x  64 0.379 BF
  13 route  11                                    ->  152 x 152 x 128
  14 conv     64       1 x 1/ 1    152 x 152 x 128 ->  152 x 152 x  64 0.379 BF
  15 conv     64       1 x 1/ 1    152 x 152 x  64 ->  152 x 152 x  64 0.189 BF
  16 conv     64       3 x 3/ 1    152 x 152 x  64 ->  152 x 152 x  64 1.703 BF
  17 Shortcut Layer: 14,  wt = 0, wn = 0, outputs: 152 x 152 x  64 0.001 BF
  18 conv     64       1 x 1/ 1    152 x 152 x  64 ->  152 x 152 x  64 0.189 BF
  19 conv     64       3 x 3/ 1    152 x 152 x  64 ->  152 x 152 x  64 1.703 BF
```

Uma vez que temos nosso modelo de rede neural compilado, com seus pesos carregados e o arquivo a ser testado pronto para uso, podemos executar a rede neural por meio do comando acima.

Por meio de !./darknet detect estaremos instanciando e inicializando nossa rede neural darknet; Como segundo parâmetro passamos o arquivo de configurações da rede neural, com seu caminho absoluto e instanciando o arquivo yolov4.cfg; Para terceiro parâmetro referenciamos o arquivo dos pesos yolov4.weights; Por fim como último parâmetro passamos o nome do arquivo para teste.

```
[yolo] params: iou loss: ciou (4), iou_norm: 0.07, obj_norm: 1.00, cls
nms_kind: greedynms (1), beta = 0.600000
Total BFLOPS 128.459
avg_outputs = 1068395
Loading weights from yolov4.weights...
 seen 64, trained: 32032 K-images (500 Kilo-batches_64)
Done! Loaded 162 layers from weights-file
 Detection layer: 139 - type = 28
 Detection layer: 150 - type = 28
 Detection layer: 161 - type = 28
criancas.jpg: Predicted in 20.802000 milli-seconds.
person: 97%
person: 97%
person: 87%
dog: 92%
```

Ao executar esta célula é possível acompanhar todo processamento da rede neural em detalhes, desde os processos de convolução até a projeção do resultado final.

Repare que ao final do log é possível ver que o arquivo crianças.jpg foi processado em 20 milissegundos, foram identificadas 3 pessoas com margens de acerto entre 87 a 97% e um cachorro com margem de acerto de 92%.

```
import cv2
import matplotlib.pyplot as plt

```

Para exibirmos o resultado em uma célula realizamos a importação das bibliotecas opencv e matplotlib.

```
def mostrar(arquivo):
  img = cv2.imread(arquivo)
  fig = plt.gcf()
  fig.set_size_inches(12,8)
  plt.axis('off')
  plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))

  plt.show()

```

Em seguida criamos uma função de nome mostrar( ) que recebe um arquivo como parâmetro.

Dentro do corpo dessa função simplesmente criamos uma simples codificação para que a imagem possa ser exibida corretamente no próprio campo de retorno da célula.

```
1  mostrar('predictions.jpg')
2
```

Chamando nossa função mostrar( ) parametrizando a mesma com predictions.jpg (arquivo gerado como retorno pela rede neural já com as bounding boxes e labels sobrepostas), eis que temos o retorno esperado, as crianças e o cachorro identificados na imagem com seus respectivos rótulos e margem de precisão quanto a identificação.

EXERCÍCIOS RESOLVIDOS E COMENTADOS

1 - Crie três variáveis com três tipos de dados diferentes, respeitando sua sintaxe:

```
nome = 'Fernando'
ano = 1987
valor = 19.99
```

Para declarar uma variável basta dar um nome para a mesma, seguido do operador de atribuição “ = “ e do conteúdo a ser atribuído para essa variável.

A forma como o atributo da variável é escrita fará com que o interpretador identifique automaticamente o tipo de dado em questão. Nesse caso, para a variável nome temos uma string, da mesma forma para variável ano temos um int assim como para variável valor temos um dado atribuído do tipo float.

---

2 - Crie um comentário de no máximo uma linha:

```
# Exemplo de comentário simples, de até uma linha.
```

Comentários curtos em Python devem iniciar com o símbolo “ # “, dessa forma o conteúdo desta linha de código será simplesmente ignorado pelo interpretador.

---

3 - Crie um comentário de mais de uma linha:

```
'''Apenas explorando a possibilidade de criar um
  comentário composto de mais de uma linha, que
  permite uma explicação mais elaborada sobre o
  código em questão.'''
```

Todo comentário maior que uma linha em Python deve ter seu conteúdo entre aspas triplas simples ‘’’ ‘’’ ou aspas triplas duplas “”” “””, de acordo com a preferência do desenvolvedor.

---

4 - Escreva um programa que mostra em tela a mensagem: Olá Mundo!!!

```
print('Olá Mundo!!!')
```

Usando da função print( ) por sua vez parametrizada com a string 'Olá Mundo!!!', ao executar o código será exibido em tela / em terminal a mensagem Olá Mundo!!!.

O retorno será:
Olá Mundo!!!

---

5 - Crie uma variável nome e atribua para a mesma um nome digitado pelo usuário:

```
nome = input('Por favor, digite o seu nome: ')
```

De acordo com o enunciado do exercício, inicialmente declaramos uma variável de nome nome, que por meio da função input( ) pede que o usuário digite algo, nesse caso, seu próprio nome.

Ao terminar de digitar o nome e pressionar a tecla ENTER, o conteúdo digitado estará atribuído para a variável nome.

---

6 - Exiba em tela o valor e o tipo de dado da variável num1: Sendo num1 = 1987.

```
num1 = 1987

print(num1)
print(type(num1))
```

Inicialmente é declarada uma variável de nome num1, que por sua vez tem atribuída o número int 1987. Parametrizando a função print( ) com a variável num1, obteremos como retorno o valor da mesma.

Já quando parametrizamos nossa função print( ) com a função type( ) por sua vez parametrizada com a variável num1, o retorno esperado é que seja exibido em tela o tipo de dado associado a essa variável.

O retorno será:
1987
Int

---

7 - Peça para que o usuário digite um número, em seguida exiba em tela o número digitado.

```
num = input('Digite um número: ')

print(f'O número digitado é : {num}')
```

Declarada a variável num, por sua vez parametrizada com a função input( ) que interage com o usuário pedindo que o mesmo digite um número, podemos exibir esse número em tela via função print( ).

Usando de f'strings e máscaras de substituição podemos criar um retorno um pouco mais elaborado, incorporando na string o conteúdo da variável num.

Supondo que o usuário tenha digitado 106, o retorno será:
O número digitado é: 106

---

8 - Peça para que o usuário digite um número, em seguida o converta para float, exibindo em tala tanto o número em si quanto seu tipo de dado.

```
num = input('Digite um número: ')
num = float(num)

print(num)
print(type(num))
```

Uma vez criada nossa variável num, com seu conteúdo vindo da interação com o usuário, podemos atualizar o conteúdo dessa variável, mudando inclusive seu tipo de dado. Para isso, nossa variável num recebe como atributo o método float( ) parametrizado com ela mesma.

Dessa forma, o conteúdo atribuído a variável num é convertido de formato e salvo sobrescrevendo o conteúdo antigo dessa variável. Por meio da função print( ) podemos exibir em tela tanto o tipo quanto o conteúdo da variável num.

Supondo que o usuário tenha digitado 52, o retorno será:
52
float

---

9 - Crie uma lista com 5 nomes de pessoas:

```
nomes = ['Ana', 'Carlos', 'Daiane', 'Fernando', 'Maria']

print(nomes)
```

Uma lista em Python, tipo de dado muito similar a uma array em outras linguagens de programação, é uma estrutura de dados que serve para guardar mais de um dado associado a uma variável, sem distinção da quantia nem do tipo de dado em questão, a única ressalva é que para que tal tipo de estrutura de dado seja reconhecida corretamente pelo interpretador do Python, a mesma deve obedecer a sintaxe, toda e qualquer lista tem seus dados dentro de “ [ ] “ colchetes.

O retorno será:
[‘Ana’, ‘Carlos’, ‘Daiane’, ‘Fernando’, ‘Maria’]

---

10 - Mostre o tamanho da lista nomes / o número de elementos da lista nomes:

Mostre separadamente apenas o terceiro elemento dessa lista:

```
nomes = ['Ana', 'Carlos', 'Daiane', 'Fernando', 'Maria']

print(len(nomes))

print(nomes[3])
```

Em nossa função print( ) usando do método len( ) por sua vez parametrizado com a variável nomes, o retorno esperado é o número de elementos que compõe essa lista.

Para instanciar apenas um determinado elemento, devemos fazer referência ao mesmo por meio de seu número de índice. Cada elemento de uma lista possui internamente um valor de índice, sendo o primeiro elemento o valor 0, o segundo elemento o valor 1, e assim por diante.

Para exibir em tela apenas o terceiro elemento da lista nomes, basta parametrizar nossa função print( ) com nomes[ ] fazendo referência ao índice de número 3.

O retorno será:
5
Fernando

---

11 - Some os valores das variáveis num1 e num2: Sendo num1 = 52 e num2 = 106. Por fim exiba em tela o resultado da soma.

```
num1 = 52
num2 = 106

print(num1 + num2)
```

Declaradas as variáveis num1 e num2 com seus respectivos atributos, podemos diretamente dentro da função print( ) executar a operação de soma destas variáveis, bastando instanciar as mesmas em nossa função print e definindo a operação de soma por meio do operador “ + “.

O retorno será:
158

---

12 - Some os valores das variáveis num1 e num2, atribuindo o resultado da soma a uma nova variável homônima. Exiba em tela o conteúdo dessa variável.

```
num1 = 52
num2 = 106

soma = num1 + num2

print(soma)
```

No exemplo anterior, ao executar a operação matemática diretamente como parâmetro de nossa função print( ) o resultado, foi exibido em tela porém após o encerramento desse bloco de código tal dado, como esperado, foi descarregado da memória.

Para guardar essa informação de forma "permanente" é necessário realizar a operação atribuída a uma variável. Desse modo, podemos exibir o dado em tela parametrizando a função print( ) com a variável em questão, e encerrado o processo o dado continua guardado na variável para reutilização.

O retorno será:
158

---

13 - Subtraia os valores de num1 e num2:

```
num1 = 52
num2 = 106

print(num1 - num2)

# ou

subtracao = num1 - num2
print(subtracao)
```

Para subtrair o valor atribuído a num1 do valor atribuído a num2, basta fazer o uso do operador " – ", diretamente em nossa função print( ) ou atribuindo esse resultado a uma variável, como já vimos anteriormente.

O retorno será:
-54

---

14 - Realize as operações de multiplicação e de divisão entre os valores das variáveis num1 e num2:

```
num1 = 52
num2 = 106

print(num1 * num2)

print(num1 / num2)
```

*Usando das mesmas variáveis com seus respectivos valores, como nos exemplos anteriores, para multiplicarmos os valores dessas variáveis basta usar do operador " * ", da mesma forma, para realizar a divisão simples dos valores de num1 e num2, usamos do operador " / ", obtendo assim o resto da divisão.*

---

15 - Eleve o valor de num1 a oitava potência, sendo num1 = 51:

```
num1 = 51

print(num1 ** 8)
```

*Para efetuar uma exponenciação em Python usamos do operador “ ** “.*

O retorno de 51 elevado à oitava potência, nesse caso é o valor que equivale a 51 x 51 x 51 x 51 x 51 x 51 x 51 x 51, resultando em 45767944570401.

---

16 - Escreva um programa que pede que o usuário dê entrada em dois valores, em seguida, exiba em tela o resultado da soma, subtração, multiplicação e divisão desses números:

```
num1 = int(input('Digite o primeiro número: '))
num2 = int(input('Digite o segundo número: '))

print(f'A soma de {num1} com {num2} é: {num1 + num2}')
print(f'A subtração de {num1} com {num2} é: {num1 - num2}')
print(f'A multiplicação de {num1} com {num2} é: {num1 * num2}')
print(f'A divisão de {num1} por {num2} é: {num1 / num2}')
```

Como feito em exemplos anteriores, por meio da função input( ) podemos pedir que o usuário digite dois números que ficarão atribuídos a duas variáveis distintas num1 e num2.

Em seguida, por meio de nossa função print( ) por sua vez parametrizada por uma f'string, conseguimos em máscaras de substituição instanciar tanto os valores de num1 quanto os valores de num2, realizando também as devidas operações matemáticas dentro de uma máscara de substituição.

---

17 - Dadas duas variáveis num1 e num2 com valores 100 e 89, respectivamente, verifique se o valor de num1 é maior que o de num2:

```
num1 = 100
num2 = 89

print(num1 > num2)
```

Nesse caso, declaradas as variáveis com seus respectivos valores podemos diretamente via função print( ) realizar a comparação lógica. Para isso, simplesmente parametrizamos nossa função print( ) com as variáveis em questão, usando entre as mesmas o operador lógico " > " para verificar se o valor atribuído a num1 é maior que o valor atribuído a num2.

Nesse caso, por se tratar de uma expressão lógica, o retorno será: True.

---

18 - Verifique se os valores de num1 e de num2 são iguais:

```
num1 = 100
num2 = 89

print(num1 == num2)
```

Para verificar de forma lógica se os valores de duas variáveis é igual ou ao menos equivalente, usamos do operador lógico " == ", que para esse exemplo retornará False, uma vez que os valores das variáveis em questão de fato não são iguais.

---

19 - Verifique se os valores de num1 e de num2 são diferentes:

```
num1 = 100
num2 = 89

print(num1 != num2)
```

De forma parecida com o exemplo anterior, agora usando do operador lógico “ != “ podemos verificar se os conteúdos de duas variáveis são diferentes. Nesse caso, o esperado é que seja retornado True, pois os valores atribuídos para num1 e num2 são diferentes entre si.

---

20 - Verifique se o valor de num1 é igual ou menor que 100:

```
num1 = 100

print(num1 <= 100)
```

Lembrando que aqui temos uma estrutura condicional composta, ou igual ou menor que, bastando apenas uma dessas proposições ser verdadeira para que o retorno gerado seja True. Em nosso exemplo, num1 não é menor mas é igual a 100, logo, o retorno será True.

---

21 - Verifique se os valores de num1 e de num1 são iguais ou menores que 100.

```
num1 = 100
num2 = 89

print(num1 <= 100 and num2 <= 100)
```

Para esse exemplo temos de fato uma estrutura condicional composta, onde além das condições de cada expressão, temos o operador and que determina que ambas expressões também sejam verdadeiras para que seja obtido um retorno True.

Em outras palavras temos na primeira expressão "o valor de num1 é menor ou é igual a 100" assim como na segunda expressão temos "o valor de num2 é menor ou é igual a 100", entre as mesmas temos o operador and, que por sua vez coloca a condição "a primeira expressão e a segunda expressão são verdadeiras?", somente retornando True caso ambas sejam.

Pela tabela verdade AND, True e True = True. (caso uma das expressões fosse False, o retorno seria False. Ex: True e False = False, False e True = False.)

**A mesma expressão poderia ser reescrita de forma reduzida como: print(num1 and num2 <= 100), gerando o mesmo retorno.*

---

22 - Verifique se os valores de num1 ou de num2 são iguais ou maiores que 100:

```
num1 = 100
num2 = 89

print(num1 >= 100 or num2 >= 100)
```

Diferentemente do exercício anterior, agora usando do operador or podemos criar expressões lógicas onde basta uma das expressões ser verdadeira para já validar todo o contexto.

Note que na primeira expressão, o valor de num1 não é maior, mas é igual a 100, já na segunda expressão o valor de num2 não é maior, mas é igual a 100, em função do operador or, mesmo caso uma dessas expressões fosse False o retorno seria True.

---

23 - Verifique se o valor de num1 consta nos elementos de lista1. Sendo num1 = 100 e lista1 = [10, 100, 1000, 10000, 100000].

```
num1 = 100

lista1 = [10, 100, 1000, 10000, 100000]

print(num1 in lista1)
```

Aqui temos uma expressão lógica fazendo o uso do operador in, que basicamente nos é útil para verificar se um determinado dado/valor consta dentro de uma variável/objeto.

Nesse caso, se o valor atribuído para num1 constar como um dos elementos de lista1, o retorno será True.

---

24 - Crie duas variáveis com dois valores numéricos inteiros digitados pelo usuário, caso o valor do primeiro número for maior que o do segundo, exiba em tela uma mensagem de acordo, caso contrário, exiba em tela uma mensagem dizendo que o primeiro valor digitado é menor que o segundo:

```
num1 = int(input('Digite o primeiro número: '))
num2 = int(input('Digite o segundo número: '))

if num1 > num2:
  print('O primeiro número digitado é o maior!')
else:
  print('O segundo número digitado é o maior!')
```

Lembrando que em uma estrutura condicional simples, criamos um objetivo a ser alcançado/atingido, indentando blocos de código de acordo com as condições importas.

Nesse caso, como temos apenas dois possíveis desfechos de acordo com a condição, supondo que o usuário digitou 25 e 26, respectivamente, o resultado exibido em tela seria 'O segundo número digitado é o maior', uma vez que a primeira condição (se num1 for maior que num2) não é válida.

---

25 - Peça para que o usuário digite um número, em seguida exiba em tela uma mensagem dizendo se tal número é PAR ou se é ÍMPAR:

```
num = int(input('Digite um número: '))

if (num % 2) == 0:
    print(f'{num} é PAR')
else:
    print(f'{num} é ÍMPAR')
```

Após criar a linha de código responsável por pedir ao usuário que o mesmo digite um número, validando esse número como do tipo inteiro e atribuindo o número em si a uma variável, vamos a estrutura condicional.

Para verificar se um determinado número é par, simplesmente o resto da divisão desse número por 2 deve ser igual a 0. Logo, criamos uma condição onde se o resto da divisão do valor de num por 2 for igual a 0, é exibida uma mensagem dizendo que o mesmo é PAR, caso essa condição não seja verdadeira, é exibida em tela uma outra mensagem, dessa vez dizendo que o número em questão é ÍMPAR.

Supondo que o usuário tenha digitado o número 15, o retorno gerado é ’15 é ÍMPAR’.

---

26 - Crie uma variável com valor inicial 0, enquanto o valor dessa variável for igual ou menos que 10, exiba em tela o próprio valor da variável. A cada execução a mesma deve ter seu valor atualizado, incrementado em 1 unidade.

```
var = 0

while var <= 10:
  print(var)
  var = var + 1
```

Toda vez que se faz necessário repetir uma determinada instrução enquanto um determinado objetivo não é atingido, podemos realizar essa repetição por meio da estrutura de repetição while.

Quando usamos while, definimos uma condição a ser alcançada ou validada, e para que a execução do código não entre em um loop infinito, também usamos nessa estrutura de código de uma variável de controle.

Repare que declaramos nossa estrutura condicional definindo que enquanto o valor de var for igual ou menor que 10, é exibido em tela por meio de nossa função print( ) o valor de var, em seguida atualizamos o valor de var somando ao seu valor atual uma unidade. Dessa forma o processo se repetirá até que var tenha o valor 10, encerrando o processo nesse ponto já que a condição definida foi alcançada.

O retorno será:
0
1
2
3
4
5
6
7

8
9
10

---

27 - Crie uma estrutura de repetição que percorre a string ‘Nikola Tesla’, exibindo em tela letra por letra desse nome:

```
for i in 'Nikola Tesla':
  print(i)
```

Sempre que estamos percorrendo o conteúdo de uma determinada variável que sabemos ter um número finito de elementos, podemos fazer a leitura de cada um destes elementos, um a um a cada laço de repetição, por meio de for.

Para a estrutura lógica do for é criada uma variável temporária (nesse caso “ i “) que a cada laço de repetição / a cada execução, irá ler um dos elementos atribuindo para si seu dado/valor, exibindo em tela seu conteúdo, até que tenha percorrido todos os elementos dessa variável.

Lembrando que o laço for consegue percorrer elementos de listas, dicionários e outras estruturas de dados parecidas, assim como em caso de uma string, caractere por caractere da mesma como se fosse um elemento individual.

Nesse caso o retorno será:
N
i
k
o
l
a

T
e
s
l
a

---

28 - Crie uma lista com 8 elementos de uma lista de compras de supermercado, por meio de um laço de repetição for liste individualmente cada um dos itens dessa lista.

```
lista10 = ['Água', 'Açúcar', 'Arroz', 'Pão', 'Leite', 'Massa', 'Carne', 'Refr
igerante']

for i in lista10:
  print(i)
```

Quando estamos a percorrer os elementos de uma lista por meio de um laço for, cada elemento dentro de seu separador será lido individualmente, tendo seu conteúdo (independentemente do tipo de dado) associado a variável temporária i.

Nesse caso o retorno será:
Água
Açúcar
Arroz
Pão
Leite
Massa
Carne
Refrigerante

---

29 - Crie um programa que lê um valor de início e um valor de fim, exibindo em tela a contagem dos números dentro desse intervalo.

```
inicio = int(input('Digite o número onde começa a contagem: '))
fim = int(input('Digite o número onde termina a contagem: '))

for i in range(inicio, fim+1):
  print(i)
```

Sempre que temos um intervalo numérico com início e fim pré-estabelecidos, podemos usar do método range( ) para ler todos elementos desse intervalo. Associando o operador lógico in, usando range( ) dentro de um laço for, é possível percorrer e iterar sobre cada elemento dentro do intervalo.

Sendo assim, declaradas as variáveis inicio e fim que recebem números digitados pelo usuário, em nossa estrutura de repetição parametrizamos nosso método range( ) com os dados de inicio e de fim incrementado de 1. É necessário fazer essa pequena codificação adicional pois em Python quando estamos lendo números dentro de um intervalo, o último número lido não é contabilizado, mas serve como gatilho para que o processo seja encerrado naquele ponto.

Em outras palavras, apenas como exemplo, para percorrer elementos dentro de um intervalo entre 0 a 10 temos de parametrizar nosso range( ) com números entre 0 e 11.

Supondo que o usuário deu entrada dos números 20 e 30 o retorno será:
20
21
22
23
24
25
26

27
28
29
30

*Caso não houvéssemos definido fim+1 como parâmetro em range( ) a contagem encerraria em 29, não atendendo o enunciado da questão.

---

30 - Crie um programa que realiza a contagem de 0 a 20, exibindo apenas os números pares:

```
for i in range(0, 21):
  if i % 2 == 0:
    print(i)
```

Da mesma forma como no exemplo anterior, usando do método range( ) parametrizado com o valor de início e de fim (acrescido em uma unidade), podemos definir que serão exibidos apenas os números pares simplesmente criando uma condição onde apenas serão exibidos os números os quais o resto de sua divisão por 2 seja 0.

```
for i in range(0, 21, 2):
  print(i)
```

Uma forma alternativa que temos para resolver esse exercício é usando do terceiro parâmetro em justaposição do método range( ) que justamente define de quantos em quantos elementos devem ser retornados na função.

Como para nosso exemplo estamos exibindo números pares, que logicamente, sequencialmente são contados de dois em dois, parametrizando o método range( ) em seu terceiro parâmetro justaposto com o número 2, indiretamente exibiremos apenas os números pares.

O retorno será:
0
2
4
6
8
10
12
14

16
18
20

---

31 - Crie um programa que realiza a Progressão Aritmética de 20 elementos, com primeiro termo e razão definidos pelo usuário:

```
termo = int(input('Digite o primeiro termo: '))
razao = int(input('Digite a razão: '))
pa = termo + (20 -1) * razao

for i in range(termo, pa + razao, razao):
  print(i)
```

Lembrando que uma progressão aritmética é uma operação onde definimos um número inicial e uma constante, também chamados de termo e razão, respectivamente. A progressão em si nada mais é do que a soma do termo anterior com a constante.

Para nosso exercício pedimos que o usuário dê entrada tanto no termo (valor inicial) quanto na razão (constante) por meio da função input( ) atribuindo esses valores a suas respectivas variáveis.

Na sequência criamos uma variável de nome pa que contextualizando a fórmula de uma progressão aritmética, pega o valor de termo, somando com a constante multiplicada pela razão. Lembrando de realizar a subtração da constante em 1 para que tenhamos o gatilho para encerrar a progressão dentro do valor definido.

A partir daí, podemos simplesmente criar um laço for que usando do método range( ) define o valor de termo como valor inicial, pa + razão como valor final, exibindo no intervalo estipulado por razão. Sendo que a cada execução do laço é exibido em tela o valor da progressão por meio da função print( ).

Supondo que para termo o usuário tenha digitado 10, e para razão tenha digitado 3, o retorno será:
10
13
16

19
22
25
28
31
34
37
40
43
46
49
52
55
58
61
64
67

---

32 - Crie um programa que exibe em tela a tabuada de um determinado número fornecido pelo usuário:

```
x = int(input('Digite um Número: '))

for num in range(1, 11):
    print(f'{x} X {num} = {x * num}')
```

Para gerar uma simples tabuada podemos fazer o uso do método range( ), nos poupando serviço de realizar cada multiplicação do número fornecido por outro número.

Inicialmente criamos uma variável de nome x que recebe do usuário um número por meio da função input( ), validando o mesmo como do tipo int pois em uma tabuada básica não temos números com casas decimais.

Em seguida por meio de um laço for, usando do método range( ) percorreremos um intervalo de 1 a 10, a cada execução retornando um valor para num. Indentado para esse laço for simplesmente exibimos em tela que o determinado número, multiplicado pelo valor atual de num, resulta em um dado valor, nesse caso, o próprio valor de x multiplicado pelo valor de num.

Tendo o usuário digitado 7, o retorno será:
7 X 1 = 7
7 X 2 = 14
7 X 3 = 21
7 X 4 = 28
7 X 5 = 35
7 X 6 = 42
7 X 7 = 49
7 X 8 = 56
7 X 9 = 63
7 X 10 = 70
---

33 - Crie um programa que realiza a contagem regressiva de 20 segundos:

```
from time import sleep

for i in range(20, -1, -1):
  print(i)
  sleep(1)
print('Contagem Terminada!!!')
```

Para esse exercício, uma forma automatizada e otimizada de gerar um mecanismo de contagem regressiva é usando do módulo sleep da biblioteca time. Para isso, via comando from time import sleep importamos tal módulo e suas funcionalidades.

Na sequência, para gerar a contagem regressiva podemos usar de um laço for com o método range( ) de modo que o valor inicial da contagem é 20, o valor final definido em -1 fará a contagem decrescente até 0, sendo o último parâmetro, referente ao intervalo de números, definido também como negativo para que a contagem aconteça de forma decrescente.

Indentado ao bloco do laço for, a cada execução é exibida em tela o número referente ao passo, assim como chamando a função sleep( ) parametrizada em 1, definimos que a cada execução do laço haverá um tempo de espera de 1 segundo. Dessa forma, teremos não somente uma contagem de 20 para 0 mas tal contagem será exibida sincronizada de um em um segundo.

Fora do laço for, por meio da função print( ), exibimos uma mensagem quando a contagem regressiva é encerrada.

O retorno será:
20
19
18
17
16
15

14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
Contagem Terminada!!!

---

34 - Crie um programa que realiza a contagem de 1 até 100, usando apenas de números ímpares, ao final do processo exiba em tela quantos números ímpares foram encontrados nesse intervalo, assim como a soma dos mesmos:

```
contador = 0
soma = 0

for i in range(1, 101):
  if i % 3 == 0:
    soma += i
    contador += 1

print(f'Foram encontrados {contador} números ímpares.')
print(f'A soma destes números é: {soma}!!!')
```

Para esse caso, novamente temos um problema envolvendo muito mais lógica do que estrutura de dados em si. Raciocine que ao mesmo tempo em que identificamos cada número ímpar precisamos o guardar em uma variável para que seja possível o usar na soma do total.

Para isso criamos duas variáveis, uma para controle (contador) e uma que fará a soma dos elementos (soma), ambas inicialmente zeradas pois serão incrementadas a cada laço de repetição.

Para o laço em si criamos um laço for que realiza a contagem de 1 até 100, retornando essa contagem para a variável temporária i. Dentro do bloco de código do laço for criamos uma estrutura condicional onde se o módulo da divisão de i por 3 deve ser igual a 0. Sempre que essa condição for verdadeira, a variável soma é atualizada somando para si o valor de i, da mesma forma, a variável contador recebe o incremento em uma unidade.

Por fim simplesmente exibimos em tela via função print( ) o que o enunciado da questão nos pede.

O retorno será:

Foram encontrados 33 números ímpares.
A soma destes números é: 1683!!!

---

35 - Crie um programa que pede ao usuário que o mesmo digite um número qualquer, em seguida retorne se esse número é primo ou não, caso não, retorne também quantas vezes esse número é divisível:

```
numero = int(input('Digite um número: '))
divisoes = 0

for i in range(1, numero + 1):
  if numero % i == 0:
    divisoes += 1

if divisoes == 2:
  print(f'{numero} é primo!!!')
  print(f'{numero} é divisível por 1 ou por {numero}')
else:
  print(f'{numero} não é primo!!!')
  print(f'{numero} é divisível {divisoes} vezes...')
```

Lembrando que um número primo é aquele que apenas é divisível por 1 ou por ele mesmo, temos de criar essa condição lógica para poder solucionar esse exercício.

Inicialmente por meio da função input( ) pedimos que o usuário digite um número, guardando esse valor na variável numero. Também é criada uma variável divisões que inicialmente está zerada.

Na sequência criamos um laço for que percorrerá de 1 até o número digitado pelo usuário + 1 via método range( ) retornando os valores para i a cada laço de repetição. Dentro do bloco referente ao nosso laço for criamos uma estrutura condicional onde se o módulo da divisão de número por i for igual a 0, divisoes recebe um incremento.

Por fim, podemos criar uma simples estrutura condicional onde se o último valor atribuído a divisoes for igual a 2, exibimos em

tela a mensagem referente a um número primo, caso contrário, exibimos em tela a mensagem referente a um número não primo.

Supondo que o usuário digitou 97, o retorno será:
97 é primo!!!
97 é divisível por 1 ou por 97.

Supondo que o usuário tenha digitado 98, o retorno será:
98 não é primo!!!
98 é divisível 6 vezes...

---

36 - Crie um programa que pede que o usuário digite um nome ou uma frase, verifique se esse conteúdo digitado é um palíndromo ou não, exibindo em tela esse resultado.

```
frase = str(input('Digite uma palavra ou frase: ')).strip().upper()
palavras = frase.split()
caracteres = ''.join(palavras)
fraseinvertida = ''

for i in range(len(caracteres) -1, -1, -1):
  fraseinvertida += caracteres[i]

print(caracteres, fraseinvertida)

if fraseinvertida == caracteres:
  print('É um palíndromo!!!')
else:
  print('Não é um palíndromo...')
```

Um palíndromo, caso você não lembre, é quando temos um nome ou até mesmo uma frase que pode ser lida por igual da maneira regular ou de trás para frente.

Desse modo, ao pedir que o usuário dê entrada em uma palavra ou frase temos de fazer algumas validações, primeira delas, garantir que todo conteúdo digitado seja um texto, isso é feito aplicando o método .str( ).

Também aplicamos um .strip( ) para remover possíveis espaços sobrando tanto no começo quanto no fim do texto e um .upper( ) para converter todos seus caracteres para maiúsculo, caso contrário, o interpretador Python pode não reconhecer corretamente um palíndromo, já fazendo essas validações temos a garantia que todo e qualquer texto será lido da mesma forma.

Em seguida, para nossa variável palavra aplicamos sobre nossa frase o método .split( ) que por sua vez irá separar todos os caracteres que compõe nossa string. De modo parecido criamos

uma variável de nome caracteres que como atributo tem o retorno do método join( ) por sua vez parametrizado com palavras de modo a juntar todos os espaços. Essa validação é necessária para que em casos de frases, os espaços entre as palavras também sejam removidos.

Dando sequência, é criada uma variável de nome fraseinvertida que inicialmente só tem como atributo uma string vazia.

Na sequência por meio de um laço for, usando de range( ) percorremos cada elemento de nossa variável caracteres de trás para frente, atualizando nossa variável fraseinvertida com cada caractere lido, preenchendo a mesma de forma que teremos os mesmos caracteres de caracteres mas em forma espelhada.

Por fim, por meio de uma estrutura condicional simples, definimos que se o conteúdo de fraseinvertida for igual ao de caracteres, exibiremos em tela que neste caso temos um palíndromo, caso contrário, retornamos que não.

Supondo que o usuário tenha digitado ‘Socorram me subi no onibus em Marrocos’, o retorno será:
SOCORRAMMESUBINOONIBUSEMMARROCOS
SOCORRAMMESUBINOONIBUSEMMARROCOS
É um palíndromo!!!

Supondo que o usuário tenha digitado ‘Fernando Feltrin’, o retorno será:
FERNANDOFELTRIN NIRTLEFODNANREF
Não é um palíndromo...

---

37 - Declare uma variável que por sua vez recebe um nome digitado pelo usuário, em seguida, exiba em tela uma mensagem de boas vindas que incorpora o nome anteriormente digitado, fazendo uso de f'strings.

```
nome = input('Digite o seu nome: ')

print(f'Bem-
Vindo(a) {nome}, é um prazer te receber em nosso Hotel.')
```

Quando queremos usar de strings mais elaboradas, não somente texto mas podendo agregar/integrar a esse texto diversas funcionalidades, usamos do conceito de interpolação, na sintaxe mais moderna da linguagem Python, chamada de f'strings e máscaras de substituição.

Ao declarar f' o interpretador Python já identifica todo e qualquer conteúdo subsequente como uma string "com super poderes", podendo a qualquer momento usar de uma máscara de substituição representada por " { } " chaves, sendo que dentro das mesmas pode ser inserido todo e qualquer conteúdo, desde o conteúdo de uma variável até métodos de classe instanciando objetos e realizando funções.

Em nosso exemplo, na f'string posta como parâmetro de nossa função print( ) no lugar da máscara {nome} será colocado o conteúdo da variável nome.

Supondo que o usuário digitou Maria, o retorno será:
Bem-Vindo(a) Maria, é um prazer te receber em nosso Hotel.

---

38 - Peça para que o usuário digite um número, diretamente dentro da função print( ) eleve esse número ao quadrado, exibindo o resultado incorporado a uma mensagem:

```
num = int(input('Digite um número: '))

print(f'{num} elevado ao quadrado resulta {num ** 2}')
```

Novamente, usando de f'strings e máscaras de substituição podemos criar interações entre uma string, dados e operações, até mesmo usando desses dados diretamente em funções de saída como nossa velha conhecida print( ).

Nesse caso, é recebido do usuário um número por meio da função input( ), atribuindo esse dado/valor a variável num.

Na sequência, em nossa função print( ) usando de f'strings, na primeira máscara de substituição será instanciada o conteúdo da variável num, assim como na segunda máscara estaremos realizando a operação de elevar o valor de num ao quadrado.

Supondo que o usuário tinha digitado 8, o resultado será:
8 elevado ao quadrado resulta 16

---

39 - Dada a seguinte lista: nomes = [‘Ana’, ‘Carlos’, ‘Daiane’, ‘Fernando’, ‘Maria’], substitua o terceiro elemento da lista por ‘Jamile’:

```
nomes = ['Ana', 'Carlos', 'Daiane', 'Fernando', 'Maria']

nomes[2] = 'Jamile'

print(nomes)
```

Para substituir de forma permanente um determinado elemento de uma lista, precisamos atualizar a variável a qual a lista está associada. Sendo assim, fazendo referência a variável nomes em sua posição de índice 2 estaremos trabalhando com o terceiro elemento da lista, atribuindo para este elemento a nova string ‘Jamile’, ao exibir em tela o conteúdo dessa variável teremos de fato a lista com esta alteração.

O retorno será:
[‘Ana’, ‘Carlos’, ‘Jamile’, ‘Fernando’, ‘Maria’]

---

40 - Adicione o elemento ‘Paulo’ na lista nomes:

```
nomes = ['Ana', 'Carlos', 'Jamile', 'Fernando', 'Maria']

nomes.append('Paulo')

print(nomes)
```

Para simplesmente adicionarmos um novo elemento a uma lista já existente, usamos do método append( ) parametrizado com o dado/valor a ser adicionado. Lembrando que usando do método append( ) o elemento em questão sempre será adicionado como último elemento da lista.

O retorno será:
[‘Ana’, ‘Carlos’, ‘Jamile’, ‘Fernando’, ‘Maria’, ‘Paulo’]

---

41 - Adicione o elemento ‘Eliana’ na lista nomes, especificamente na terceira posição da lista:

```
nomes = ['Ana', 'Carlos', 'Jamile', 'Fernando', 'Maria', 'Paulo']

nomes.insert(2, 'Eliana')

print(nomes)
```

Quando necessitamos incluir um novo elemento em uma posição específica da lista, temos de fazer o uso do método insert( ), parametrizando o mesmo com o número do índice a ser usado, seguido do dado/valor a ser inserido naquela posição.

O retorno será:
[‘Ana’, ‘Carlos’, ‘Eliana’, ‘Jamile’, ‘Fernando’, ‘Maria’, ‘Paulo’]

---

42 - Remova o elemento ‘Carlos’ da lista nomes:

```
nomes = ['Ana', 'Carlos', 'Jamile', 'Fernando', 'Maria', 'Paulo']

nomes.remove('Carlos')

print(nomes)
```

Para remover um elemento específico de uma lista, podemos usar do método remove( ) por sua vez parametrizado com o nome do elemento. Caso o elemento não exista entre os elementos que compõe a lista, será gerado um erro “x not in list” pois o elemento não consta na lista.

O retorno será:
[‘Ana’, ‘Eliana’, ‘Jamile’, ‘Fernando’, ‘Maria’, ‘Paulo’]

---

43 - Mostre o segundo, terceiro e quarto elemento da lista nomes. Separadamente, mostre apenas o último elemento da lista nomes:

```
nomes = ['Ana', 'Carlos', 'Jamile', 'Fernando', 'Maria', 'Paulo']

print(nomes[1:4])

print(nomes[-1])
```

Para instanciar e exibir determinados elementos de uma lista, basta que façamos referência a seus respectivos números de índice.

Nesse caso, para exibir o segundo, terceiro e quarto elementos (em outras palavras do segundo ao quarto elemento) podemos instanciar a variável nomes[ ] fazendo referência aos índices 1:4 (lembrando que em Python todo e qualquer índice começa em 0, assim como o último elemento lido não é exibido, servindo internamente apenas como gatilho para encerramento de contagem de elementos por parte do interpretador).

De forma parecida, para instanciar e exibir especificamente apenas o último elemento de uma lista, usamos a notação -1 como referência de índice.

O retorno será:
[‘Carlos’, ‘Jamile’, ‘Fernando’]
Paulo

---

44 – Crie um dicionário via método construtor dict( ), atribuindo para o mesmo ao menos 5 conjuntos de chaves e valores representando objetos e seus respectivos preços:

```
dicionario = dict(Pão = 'R$ 1,99',
                  Açúcar = 'R$ 4,99',
                  Café = 'R$ 3,49',
                  Macarrão = 'R$ 3,99',
                  Carne = 'R$ 16,99')

print(dicionario)
```

Para criar um dicionário em Python usando de seu método construtor, inserindo dados manualmente um a um, é necessário que se respeite algumas características.

Inicialmente todo e qualquer dicionário deve estar atribuído a uma variável, logo após usando do método dict( ) nosso interpretador espera que os dados que serão repassados como parâmetro irão compor os campos de um dicionário, em outras palavras, suas chaves e valores.

Por fim, ao inserir dados de forma manual, no campo de parâmetros devemos colocar tais dados como se estivéssemos declarando variáveis, ou seja, dando um nome, usando do operador de atribuição " = " seguido do atributo, porém, internamente o que o método construtor de dicionários fará é converter nossa "variável" em chave, assim como seu "atributo" em valor.

Em nosso exemplo, apenas pegando um dos itens para exemplificar, Café = 'R$ 3,49' será convertido em 'Café':'R$ 3,49', sendo Café a chave e R$ 3,49 o valor deste item no dicionário.

Por meio da função print( ), parametrizada com o conteúdo da variável dicionário, podemos ver o dicionário em si em sua estrutura.

O retorno será:
{'Pão' : 'R$ 1,99', 'Açúcar' : 'R$ 4,99', 'Café' : 'R$ 3,49', 'Macarrão' : 'R$ 3,99', 'Carne' : 'R$ 16,99'}

---

45 – Crie um dicionário usando o construtor de dicionários do Python, alimente os valores do mesmo com os dados de duas listas:

```
itens = ['Caneta', 'Lápis', 'Borracha', 'Caderno']
precos = ['1,99', '0,99', '0,50', '9,90']

dicionario1 = dict(keys = itens, values = precos)

print(dicionario1)
print(type(dicionario1))
```

Quando estamos trabalhando com o tipo de dado dicionário em Python temos algumas particularidades, primeira delas que um dicionário pode ser criado diretamente pela sintaxe de todo e qualquer conteúdo disposto entre { } chaves, assim como usar do método construtor dict( ) onde o que é passado como parâmetro para este método se torna dados de um dicionário.

Importante levar em consideração que um dicionário obrigatoriamente possui seus dados dispostos em "chave : valor" sendo assim, quando criamos um dicionário via método construtor devemos inserir estes tipos de dados em seus respectivos campos.

De acordo com o enunciado, criamos as listas, em seguida criamos uma variável de nome dicionario1 que por sua vez usando do método construtor dict( ) repassa para keys os dados da lista itens, da mesma forma repassando para values os dados da lista precos.

Via função print( ), parametrizada com o conteúdo de dicionário1 e na sequência parametrizada com o tipo da variável dicionario1, podemos gerar um retorno.

O retorno será:
{'keys':['Caneta', 'Lápis', 'Borracha', 'Caderno'], 'values':['1,99', '0,99', '0,50', '9,90']}
<class 'dict'>

---

46 - Crie uma simples estrutura de dados simulando um cadastro para uma loja. Nesse cadastro deve conter informações como nome, idade, sexo, estado civil, nacionalidade, faixa de renda, etc... Exiba em tela tais dados:

```
cadastro = {'Nome':'Paulo',
            'Sexo':'Masculino',
            'Idade':38,
            'Nacionalidade':'Brasileiro',
            'Estado Civil':'Divorciado',
            'Escolaridade':['Ensino Superior',
                            'Doutorado'],
            'Ocupação':'Professor',
            'Renda':'1999.00 - 2999.00'}

print(cadastro)
```

Nesse exercício temos a orientação de criar uma estrutura de dados onde certos campos devem guardar certas informações, o que condiz perfeitamente com a estrutura de dado de um dicionário em Python, onde podemos atribuir a uma variável uma série de dados organizados em chaves e valores.

Lembrando que pela sintaxe um dicionário é uma estrutura identificada por " { } " chaves, sendo seus dados internos organizados em " chave : valor " aceitando todo e qualquer tipo de dado para esses campos, inclusive dicionários dentro de dicionários, respeitando sempre a sintaxe dos tipos de dados usados.

Para cada campo criado de acordo com o enunciado da questão, respeitando a sintaxe, criamos um preenchimento com dados fictícios, atendendo todos os requisitos da questão.

O retorno será:
{'Nome':'Paulo', 'Sexo':'Masculino', 'Idade':38, 'Nacionalidade':'Brasileiro', 'Estado Civil':'Divorciado', 'Escolaridade': ['Ensino Superior','Doutorado'], 'Ocupação':'Professor', 'Renda':'1999.00 – 2999.00'}

---

47 - Crie um programa que recebe dados de um aluno como nome e suas notas em supostos 3 trimestres de aula, retornando um novo dicionário com o nome do aluno e a média de suas notas:

```
aluno = [{'Nome':'Fernando', 'Notas': [62, 73, 90]}]

def calcula_media(aluno):
  notas = []
  for media in aluno:
    if len(media['Notas']) > 0:
      temp = round(sum(media['Notas'])/len(media['Notas']))
    else:
      temp = 0
    notas.append({'Nome':media['Nome'],'Média das notas':temp})
  print(notas)

media_estudante = calcula_media(aluno)
```

Para um exercício como esse, teremos de por meio de uma função desempacotar os dados de um aluno, realizando algumas operações e validações para o mesmo, por fim retornando um novo dicionário.

Para isso, inicialmente analisando nossa variável aluno, podemos notar que temos dicionário e lista dentro de lista, de modo que temos que temos que iterar sobre chaves e valores do dicionário, de modo a separar o campo referente as notas para que possamos realizar o cálculo da média das mesmas.

Sendo assim, criamos uma função de nome calcula_media( ) que recebe os dados de aluno como parâmetro. No bloco da função criamos uma variável de nome notas que inicialmente possui como atributo uma lista vazia. Em seguida, por meio de um laço for percorremos os dados de aluno, mais especificamente onde consta o campo 'Notas', validando que caso seu tamanho seja maior que 0, é realizado a divisão entre a soma dos valores de notas pelo tamanho de notas. Em outras palavras, para realizar a média,

pegamos todos os valores, somamos os mesmos, dividindo pelo número de trimestres nesse caso, retornando esses dados para temp.

Caso essa condição (número de notas disponíveis no trimestre) não for verdadeira, a variável temp recebe o valor zero.

Finalizando esse bloco, a variável notas é atualizada por meio do método append( ), recebendo em forma de dicionário o nome do aluno e a média das notas guardadas anteriormente na variável temp.

Por fim, é exibido em tela o conteúdo de notas.

O retorno será:
{‘Nome’: ‘Fernando’, ‘Média das notas’: 75}

---

48 - Crie um sistema de perguntas e respostas que interage com o usuário, pedindo que o mesmo insira uma resposta. Caso a primeira questão esteja correta, exiba em tela uma mensagem de acerto e parta para a próxima pergunta, caso incorreta, exiba uma mensagem de erro e pule para próxima pergunta:

```
base = {
    'Pergunta 01 ':{
        'pergunta':' Quanto é 4 X 4 ? ',
        'alternativas':{'a':'12','b':'24','c':'16','d':'20'},
        'resposta_certa':'c',
    },
    'Pergunta 02 ':{
        'pergunta':' Quanto é 6 / 3 ? ',
        'alternativas':{'a':'2','b':'1','c':'3','d':'4'},
        'resposta_certa':'a',
    },
}

respostas_certas = 0

for pkeys, pvalues in base.items():
  print(f'{pkeys}:{pvalues["pergunta"]}')

  for rkeys, rvalues in pvalues['alternativas'].items():
    print(f'[{rkeys}]: {rvalues}')

  resposta = input('Escolha uma alternativa: [a],[b],[c] ou [d]')

  if resposta == pvalues['resposta_certa']:
    print('Resposta Correta!!!')
    respostas_certas += 1
  else:
    print('Resposta Incorreta!!!')

if respostas_certas == 0:
```

```
  print('Você não acertou nenhuma questão.')
elif respostas_certas == 1:
  print('Você acertou apenas uma questão.')
else:
  print('Você acertou todas as questões.')
```

De acordo com o enunciado da questão, uma vez que temos uma base de dados de perguntas e respostas, podemos associar que essa base pode ser criada com um dicionário onde suas chaves e valores se tornam perguntas e respostas, respectivamente.

Raciocine que como teremos mais de uma pergunta de múltipla escolha, podemos abstrair essa ideia para um dicionário dentro de outro dicionário, importante raciocinar também que para cada pergunta teremos a pergunta em si, as alternativas e a resposta, para que possamos assim interagir com o usuário, verificando se o mesmo acertou ou errou em suas respostas.

Inicialmente criamos uma variável de nome base, que por sua vez tem um dicionário onde a primeira chave é 'Pergunta 01 ', sendo seu valor outro dicionário com a pergunta, as alternativas dentro de outro dicionário e a resposta certa para essa questão. O mesmo é feito para 'Pergunta 02' e poderia ser replicado para quantas perguntas fossem necessárias.

Na sequência declaramos a variável respostas_certas, inicialmente com valor zero. Em seguida criamos um laço for que desempacotará dos itens de nossa base as chaves e valores referentes as perguntas. Dentro desse mesmo bloco é criado outro laço for para que percorra da pergunta, quais chaves e valores correspondem as alternativas.

Uma vez criada a estrutura que exibirá em tela as perguntas e suas respectivas alternativas, podemos pedir que o usuário dê entrada na alternativa o qual considera a correta de acordo com a pergunta.

Na sequência criamos uma estrutura condicional onde se o valor de resposta for igual ao valor situado no campo 'resposta_certa' de nossa pergunta, é exibido em tela uma

mensagem de acerto, também atualizando o valor da variável respostas_certas em 1 unidade. Caso contrário, é simplesmente exibida em tela uma mensagem de erro.

Por fim, fora do laço for, novamente no escopo global de nosso código, criamos uma simples estrutura condicional que com base no valor de respostas_certas exibe em tela o número de questões que o usuário acertou.

Nesse caso, supondo que para a primeira questão o usuário tenha digitado ‘ d ‘ e para a segunda questão tenha digitado ‘ a ‘, o retorno será:
Resposta Incorreta!!!
Resposta Correta!!!
Você acertou apenas uma questão.

---

49 - Crie uma função de nome funcao1, que por sua vez não realiza nenhuma ação:

```
def funcao1():
  pass
```

Em Python podemos declarar uma função qualquer, mesmo sem um propósito definido inicialmente, para isso definimos a função como de costume, respeitando sua sintaxe, e no corpo da função simplesmente colocamos a palavra reservada pass, que internamente dá a instrução ao interpretador para que essa linha, ou bloco de código, seja ignorado.

Nesse caso, nenhum retorno é gerado.

---

50 - Atribua a função funcao1 a uma variável:

```
def funcao1():
  pass

var1 = funcao1()
```

Após criada a estrutura de uma função qualquer, podemos declarar uma variável que tem atribuída para si uma função. Dessa forma, quando fizermos o uso dessa variável estaremos instanciando e inicializando sua função. No jargão popular, a variável var1 está chamando a função funcao1( ).

Nesse caso, nenhum retorno é gerado.

---

51 - Crie uma função que retorna um valor padrão:

```
def msg():
  return 0

msg()
```

Uma função pode ou não retornar algum dado/valor para atualizar uma variável ou apenas para controle. Em muitas linguagens de programação existe a convenção de que quando um script/código é executado sem erros, o mesmo pode internamente retornar o valor 0, assim como quando houverem erros pode retornar o valor 1.

Sendo assim, em Python podemos perfeitamente dentro de uma função criar um retorno para controle, apenas para verificar se o bloco de código da função foi executado corretamente assim como em estruturas mais robustas podemos usar do próprio "resultado" do processamento da função para atualizar algum elemento do escopo global do código como uma variável por exemplo.

O retorno será:
0

---

52 - Crie uma função que exibe em tela uma mensagem de boas-vindas:

```
def mensagem():
  print('Bem Vindo(a)!!!')

var1 = mensagem()
```

Uma vez definida a função mensagem( ), sem parâmetros mesmo, em seu corpo colocamos uma função print( ) por sua vez parametrizada com a string 'Bem Vindo(a)!!!'.

Na sequência declaramos uma variável de nome var1 que instancia e inicializa a função mensagem( ), dessa forma, ao executar nosso código, nossa variável var1 irá chamar a função mensagem( ) que por sua vez realizará sua função programada, nesse caso, exibir em tela a mensagem parametrizada em print( ).

O retorno será:
Bem Vindo(a)!!!

---

53 - Crie uma função que recebe um nome como parâmetro e exibe em tela uma mensagem de boas-vindas. O nome deve ser fornecido pelo usuário, incorporado na mensagem de boas-vindas da função:

```
def mensagem(nome):
  print(f'Bem vindo(a) {nome}')

nome = input('Digite o seu nome: ')
nome = mensagem(nome)
```

Inicialmente definimos nossa função mensagem( ) que receberá algum dado/valor a ser usado como parâmetro em nome. Dentro do corpo da função usamos da função print( ) e de f'strings para criar uma mensagem de boas-vindas onde em uma máscara de substituição { } será instanciado o nome do parâmetro nome.

Em seguida simplesmente criamos uma variável nome que por meio da função input( ) pede que o usuário digite o mesmo.

Por fim, a variável nome chama a função mensagem( ) passando como parâmetro o conteúdo da variável nome.

Supondo que o usuário tenha digitado 'Fernando', o retorno será:
Bem vindo(a) Fernando

---

54 - Crie uma função que recebe um valor digitado pelo usuário e eleva esse valor ao quadrado:

```
def exp(num):
  return num ** 2

num = int(input('Digite um número: '))
num = exp(num)

print(num)
```

*Inicialmente definimos uma função de nome exp( ) que recebe como parâmetro um número em num. Dentro do corpo da função retornamos o valor de num elevado ao quadrado, por meio da expressão num ** 2.*

Na sequência pedimos que o usuário digite um número, validando o mesmo como do tipo int, apenas por convenção, atribuindo esse valor a variável num.

Em seguida a variável num chama a função exp( ) parametrizando a mesma com o próprio valor. Por fim, via função print( ) exibimos em tela o valor de num.

Lembrando que como aqui estamos usando a variável num inicialmente para guardar o valor digitado pelo usuário, e posteriormente chamando a função, estamos atualizando o próprio valor de num com o retorno da função.

Supondo que o usuário tenha digitado 8, o retorno será:
64

---

55 - Crie uma função com dois parâmetros relacionados ao nome e sobrenome de uma pessoa, a função deve retornar uma mensagem de boas-vindas e esses dados devem ser digitados pelo usuário:

```
def boas_vindas(nome, sobrenome):
  print(f'Bem vindo(a) {nome} {sobrenome}')

nome = input('Digite o seu nome: ')
sobrenome = input('Agora digite o seu sobrenome: ')

pessoa = boas_vindas(nome, sobrenome)
```

Se tratando de funções em Python, uma função pode não ter nenhum parâmetro, assim como ter um ou mais parâmetros desde que eles tenham um propósito explícito no corpo da função.

Nessa linha de raciocínio, definimos nossa função boas_vindas( ) que por sua vez receberá dois parâmetros, nome e sobrenome. No corpo da função por meio da função print( ) usaremos de uma f'string para incorporar na mensagem de boas-vindas os dados oriundos dos parâmetros nome e sobrenome, respectivamente.

Na sequência criamos as estruturas de interação com o usuário, por meio da função input( ), pedindo que o mesmo digite seu nome e sobrenome, atribuindo esses dados as variáveis nome e sobrenome.

Por fim declaramos uma variável de nome pessoa, que chama a função boas_vindas( ) parametrizando a mesma em nome com o conteúdo da variável nome, da mesma forma parametrizando a mesma em sobrenome com o conteúdo da variável sobrenome.

Supondo que o usuário tenha digitado 'Fernando' e 'Feltrin', o retorno será:
Bem vindo(a) Fernando Feltrin

---

56 - Crie uma função com dois parâmetros, sendo um deles com um dado/valor predeterminado:

```
def boas_vindas(nome, nacionalidade = 'Brasileiro'):
  print(f'{nome} é {nacionalidade}!!!')

nome = input('Digite o seu nome: ')
ex1 = boas_vindas(nome)

nacionalidade = input('Digite sua nacionalidade: ')
ex2 = boas_vindas(nome, nacionalidade)
```

Se tratando do modo como parametrizamos uma função, em Python temos a flexibilidade de realizar algumas ações diretamente sobre os parâmetros. Uma delas é no momento da declaração dos parâmetros/argumentos de uma função, definir um valor padrão, pré-determinado. Isso é muito útil para casos onde precisamos de um determinado parâmetro fornecido pelo usuário, porém o mesmo não dá entrada neste dado, assim contornamos um possível erro a ser gerado.

Em outras palavras, ao definirmos um parâmetro com um valor padrão, caso o usuário dê entrada em um dado/valor para esse parâmetro, o mesmo será atualizado com esse dado/valor fornecido pelo usuário, caso contrário, usará do valor padrão pré-determinado.

Aqui, usando do mesmo exemplo anterior, em nossa função boas_vindas( ) temos dois parâmetros, o primeiro deles, nome, da forma como está declarado é obrigatório que tenha um dado aqui repassado por uma variável ou pelo usuário, já o segundo parâmetro, que agora tem o nome nacionalidade, possui como padrão 'Brasileiro'.

No corpo da função é criada uma f'string que incorpora tanto os dados de nome quanto de nacionalidade em uma mensagem a ser exibida em tela por meio da função print( ).

Na sequência, apenas para exemplo, criamos duas estruturas que receberão dados do usuário via input( ) guardando esses dados para as variáveis nome e nacionalidade.

Em nossa variável ex1 chamamos a função boas_vindas( ) parametrizando a mesma apenas com nome. Já para nossa variável ex2, também chamamos a função boas_vindas( ), dessa vez parametrizando a mesma com os dados de nome e de nacionalidade.

Supondo que o usuário tenha digitado ‘Maria’ e ‘Portuguesa’, o retorno para ex1 será:
Maria é Brasileiro.
Enquanto o retorno para ex2 será:
Maria é Portuguesa.

Como mencionado anteriormente, em ex1 não repassamos o parâmetro referente à nacionalidade, então foi usado o padrão ‘Brasileiro’, em ex2 repassamos tanto os dados de nome quanto de nacionalidade, dessa forma o parâmetro nacionalidade da função boas_vindas( ) teve seu dado/valor atualizado de ‘Brasileiro’ para ‘Portuguesa’.

---

57 - Crie uma função com três parâmetros, sendo dois deles com dados/valores padrão, alterando o terceiro deles contornando o paradigma da justaposição de argumentos:

```
def pessoa1(nome, idade = 18, funcao = 'técnico'):
  print(f'{nome} tem {idade} anos, e sua função é {funcao}')

p1 = pessoa1('Fernando', funcao = 'gerente')
```

Por justaposição entendemos que certos argumentos, quando passados de forma simples, obedecerão a ordem ao qual os mesmos forem inseridos. Em outras palavras, ao chamar uma função, o primeiro parâmetro repassado será instanciado para o primeiro parâmetro da função, o segundo repassado para o segundo parâmetro declarado da função, e assim sucessivamente.

Quando queremos contornar a justaposição, repassando um determinado dado/valor para um parâmetro de função em uma posição específica, podemos simplesmente instanciar o parâmetro o qual queremos repassar o dado na própria chamada da função.

Em nosso exemplo temos a função pessoa1( ) que receberá um nome, uma idade (tendo um argumento pré-definido 18) e uma função (tendo seu argumento pré-definido como 'técnico'). A função em si quando executada apenas exibe em tela uma f'string incorporando os parâmetros em uma mensagem.

Por fim declaramos uma variável de nome p1 que instancia e inicializa a função pessoa1( ) parametrizando a mesma com 'Fernando' que por justaposição será repassado para nome, e função = 'gerente' que nesse caso atualiza o parâmetro função de 'técnico' para 'gerente'. Como ao chamar a função não foi repassado nenhum dado para idade, a função usará do valor pré-determinado 18.

Nesse caso o retorno será:
Fernando tem 18 anos, e sua função é gerente

---

58 - Crie uma função que pode conter dois ou mais parâmetros, porém sem um número definido e declarado de parâmetros:

```
def msg(*args):
  print(f'Os parâmetros são: {args}')

ex2 = msg('nome = Fernando', 'idade = 33', 'profissão = professor')
```

*Em algumas situações bastante específicas pode ser que tenhamos de definir uma função onde inicialmente não sabemos ao certo quais e quantos parâmetros teremos para a mesma. Uma maneira de contornar esse problema, ao menos para definir a função e realizar alguns testes com a mesma, é usar do marcador *args. Com este tipo de marcador, podemos criar quantos parâmetros quisermos temporariamente, os parâmetros por sua vez serão justapostos e estarão usáveis pelo corpo da função no momento de sua execução.*

*Para ficar mais claro, vamos ao exemplo, onde inicialmente definimos a função msg( ) por sua vez parametrizada com *args. No corpo dessa função temos uma simples mensagem a ser exibida em tela pela função print( ) onde exibiremos tudo o que for repassado para args. Note que o marcador " * " é usado apenas no campo do parâmetro na declaração da função, apenas para que o interpretador reconheça que ali serão inseridos parâmetros posteriormente.*

Na sequência criamos uma variável de nome ex2 que chama a função msg( ) parametrizando a mesma com uma série de parâmetros, separados por vírgula, nesse caso apenas para fins de exemplo, simulando parâmetros já com dados/valores pré-determinados. Dessa forma, como parâmetros temporários para essa função teremos 'nome = Fernando', 'idade = 33' e 'profissão = professor'.

Lembrando mais uma vez que esse tipo de parâmetro é temporário, após executar uma vez essa função, esses dados serão

descarregados da memória e a função terá de ser parametrizada novamente.

O retorno será:
Os parâmetros são: (‘nome = Fernando’, ‘idade = 33’, ‘profissão = professor’)

---

59 - Crie uma função de número de parâmetros indefinido, que realiza a soma dos números repassados como parâmetro, independentemente da quantidade de números:

```
def soma(*args):
  num = 0
  for valordigitado in args:
    num += valordigitado
  print(f'O resultado da soma é: {num})
```

```
soma = soma(18, 43, 99, 1)
```

*No exemplo anterior foi mencionado que quando fazemos o uso de *args, os dados/valores repassados para esse campo se tornam parâmetros "temporários" pois uma vez que os mesmos tenham sido usados pela função são descarregados da memória e a função volta ao seu estado inicial sem parâmetros até que novamente seja parametrizada.*

*Para realizarmos uma interação mais interessante com os parâmetros que estamos usando em *args, além de simplesmente os exibir em tela, é necessário que no corpo da função realizemos o "desempacotamento" dos mesmos, atribuindo seus dados/valores a uma ou mais variáveis. Dessa forma temos dados repassados como parâmetros e de fato usáveis dentro do bloco de código da função.*

*De acordo com o que é pedido no enunciado da questão, para criar uma calculadora de soma, que realiza a soma de quantos números forem necessários, inicialmente definimos nossa função com o marcador *args.*

*Dentro do corpo dessa função, criamos uma variável de nome num, inicialmente de valor 0, a ser atualizada com dados oriundos de *args.*

*Para desempacotar os dados de *args, podemos utilizar de um laço for, que a cada laço de repetição lê o valor em *args salvando seu valor em uma variável, nesse caso, para uma simples*

*calculadora de soma, atualizando o valor de num e somando seu valor atual com cada valor novo lido em *args pelo laço for atribuído a variável temporária valordigitado.*

Por fim, para testar se nossa função soma( ) cumpre o que é esperado para a mesma, criamos uma variável de nome soma que por sua vez chama a função soma, passando como parâmetros os números 18, 43, 99 e 1.

O retorno será:
O resultado da soma é: 161

---

60 - Crie uma função que recebe parâmetros tanto por justaposição (*args) quanto nomeados (kwargs):**

```
def identificacao(*args, **kwargs):
  for n in args:
    nome = n
    #print(f'{n}')

    for k, v in kwargs.items():
      idade = k
      sexo = v
      #print(f'{idade}, {sexo}')

      print(f'Nome: {nome},  {idade},  {sexo}')

pessoa = identificacao('Fernando', idade = 33, sexo = 'M')
```

*Este é um exercício interessante pois coloca à prova diferentes níveis de desempacotamento de dados de *args e **kwargs, de modo que para conseguirmos realizar as devidas interações com os parâmetros, é de suma importância gerar a cadeia de interações na ordem e indentação correta.*

*De acordo com o enunciado da questão, teremos de fazer uso tanto de número de parâmetros indefinido, porém justapostos, quanto de parâmetros nomeados. Para isso, definimos nossa função identificacao( ) que recebe como parâmetros *args e **kwargs.*

*Apenas fazendo um adendo, a nomenclatura 'args' e 'kwargs' pode ser substituída por qualquer outro nome, como *números e **nomes por exemplo, o marcador em si que o interpretador Python identifica na verdade são os " * " e " ** ". Outro ponto que não havia sido comentado é que quando usamos de *args e **kwargs, por convenção usamos desses marcadores nessa ordem, nunca **kwargs e *args pois em algumas IDEs essa ordem pode gerar exceções.*

*Dando sequência com nosso código, para guardar o dado/valor de *args simplesmente usamos de um laço for que lê esse argumento, o salvando em uma variável.*

*Dentro desse laço criaremos um novo laço de repetição, agora com as variáveis temporárias k e v que percorrerão os argumentos em *kwargs.items( ) uma vez que os dados estarão em formato de dicionário, guardando esses dados/valores em suas respectivas variáveis.*

Por fim, podemos usar de nossa velha conhecida função print( ) para exibir em tela tais dados.

*Fora do corpo de nossa função, no escopo global do código, declaramos uma variável de nome pessoa que chama a função identificacao( ) parametrizando a mesma com 'Fernando', que por justaposição será atrelado ao campo *args, seguido de dois outros parâmetros nomeados idade e sexo, que justamente por serem nomeados irão gerar e compor os campos em *kwargs.*

O retorno será:
Nome: Fernando, idade, 33
Nome: Fernando, sexo, M

---

61 - Escreva um programa que retorna o número de Fibonacci: Sendo o número de Fibonacci um valor iniciado em 0 ou em 1 onde cada termo subsequente corresponde à soma dos dois anteriores.

```
def fibonacci(n):
 if n <= 1:
   return n
 else:
   return fibonacci(n-1) + fibonacci(n-2)

num = int(input('Digite um numero para encontrar seu Fibonacci: '))
resposta = fibonacci(num-1)
print(resposta)
```

Inicialmente criamos nossa função fibonacci( ) que receberá como parâmetro um número. Indentado dentro do bloco da função criamos a estrutura lógica para gerar um número Fibonacci. Por meio de uma estrutura condicional iniciamos validando que caso o valor de n for igual ou menor que 1 é retornado o próprio número pois não temos termos suficientes para realizar as devidas somas. Caso essa condição não seja válida, então a função retorna uma operação de soma instanciando a própria função fibonacci( ), uma vez parametrizada com n – 1 e também parametrizada com n – 2.

Em seguida criamos uma variável de nome num que recebe via input( ) um número digitado pelo usuário, validado como inteiro.

Também criamos uma variável de nome resposta, que chama a função parametrizando a mesma com o número fornecido pelo usuário salvo em num – 1, finalizando o processo exibindo em tela o resultado da operação.

Supondo que o usuário tenha digitado 8, o retorno será:
13

---

62 - Crie um programa modularizado, onde em um arquivo teremos uma lista de médicos fictícios a serem consultados, em outro arquivo, teremos a estrutura principal do programa, que por sua vez realiza o agendamento de uma consulta médica com base na interação com o usuário.

```
# Arquivo medicos.py
medicos = ['Grazielle Veiga',
           'Matheus Correa']

# Arquivo main.py
import medicos

menu = str(input('Deseja agendar uma consulta? (S ou N) ')).upper()

if menu == 'S':
  paciente = input('Por favor, digite seu nome completo: ')
  print(f'{paciente}, escolha com qual médico deseja consultar:')
```

```
  print('1 - Grazielle Veiga')
  print('2 - Matheus Correa')
  medico = int(input('Com qual médico deseja agendar consulta?'))
  if medico == 1:
    print(f'Sua consulta com a Dr.a {medicos.medicos[0]} será agenda
da.')
  if medico == 2:
    print(f'Sua consulta com o Dr. {medicos.medicos[1]} será agendad
a.')
else:
  print('Agradecemos o seu contato!!!')
```

Quando estamos trabalhando com modularização, subentendemos que os módulos os quais estaremos fazendo uso contém blocos de código isolados que podem ser utilizados por nosso programa. Uma prática muito comum em programas com muitas funcionalidades é que grande parte das mesmas sejam

modularizadas, isto nos proporciona uma performance melhor de nosso programa pois no lugar de carregar tudo o que está disponível para o mesmo, importamos apenas o necessário conforme a demanda do programa. Modularização também é muito importante para a manutenção de um programa, pois podemos implementar ou alterar funcionalidades sem que um erro na parte modularizada afete o funcionamento do programa como um todo.

Pois bem, de acordo com o enunciado então inicialmente criamos um arquivo .py de nome médicos, dentro do mesmo a variável médicos possui uma lista com nomes de médicos fictícios.

Em nosso arquivo principal main.py, inicialmente realizamos a importação do módulo médicos por meio do comando import médicos, dessa forma, assim que necessário teremos acesso aos dados contidos em nosso arquivo médicos.py.

Em seguida criamos uma variável de nome menu que pede ao usuário que digite S ou N caso queira ou não agendar uma consulta. Aqui realizamos duas validações, a primeira convertendo o que o usuário digitar para tipo string, e a segunda convertendo o que o usuário digitar para maiúsculo.

Na sequência criamos uma estrutura condicional, onde inicialmente caso o valor atribuído para menu seja igual a 'S', o usuário entrará na fase de agendamento da consulta, caso essa condição inicial não seja verdadeira, é exibido em tela apenas uma mensagem agradecendo o contato...

Em nossa estrutura condicional, validade como 'S', inicialmente temos uma variável de nome paciente que via input( ) pede que o usuário digite seu nome, em seguida criamos algumas funções print( ) exibindo em tela um simples menu para que o usuário escolha com qual médico queira consultar.

Dando continuidade, declaramos uma variável de nome medico que recebe do usuário um dado/valor correspondente a qual médico o mesmo escolheu anteriormente.

Aqui criamos uma nova estrutura condicional dentro da anterior, onde caso o que o usuário tenha digitado para variável medico seja

igual a 1, instanciamos de nosso módulo medicos a variável medicos que por sua vez, como lista, em sua posição de índice 0, fará referência a Dr.a 'Grazielle Veiga'. Caso o número digitado pelo usuário tenha sido 2, usamos do mesmo processo porém instanciando de medicos o elemento de índice 1 da lista.

Supondo que o usuário tenha digitado N ou qualquer outro caractere, o retorno será:
Agradecemos seu contato!!!

Supondo que o usuário tenha digitado S em seguida seu nome e a opção 1 do menu, o retorno será:
Fernando, escolha com qual médico deseja consultar:
1 – Grazielle Veiga
2 – Matheus Correa
Sua consulta com a Dr.a Grazielle Veiga será agendada.

---

63 - Aprimore o exemplo anterior, incluindo um módulo simulando o cadastro de usuários em um plano de saúde, apenas permitindo o agendamento de consulta caso o usuário que está interagindo com o programa conste no cadastro:

```
# Arquivo medicos.py
medicos = ['Grazielle Veiga',
           'Matheus Correa']

# Arquivo cadastro_plano_saude.py
usuarios = {'001':'Fernando Feltrin',
            '002':'Ana Clara'}

# Arquivo main.py
import medicos
import cadastro_plano_saude

usuario = str(input('Digite seu número de usuário: '))
if usuario in cadastro_plano_saude.usuarios.keys():
  if usuario == '001':
    usuario = 'Fernando'
    print('Bem-vindo Fernando!!!')
    #return usuario
  if usuario == '002':
    usuario = 'Ana Clara'
    print('Bem-vinda Ana Clara!!!')
    #return usuario
  else:
    print('Usuário desconhecido ou não cadastrado.')

  menu = str(input('Deseja agendar uma consulta? (S ou N) ')).upper(
)
  if menu == 'S':
    print('Escolha com qual médico deseja consultar:')
    #print(f'{usuario}, escolha com qual médico deseja consultar:')
    #caso tenha retornado o nome do usuário na condicional anterior
    print('1 - Grazielle Veiga')
```

```
    print('2 - Matheus Correa')
    medico = int(input('Com qual médico deseja agendar consulta?'))
    if medico == 1:
      print(f'{usuario}, sua consulta com a Dr.a {medicos.medicos[0]} e
stá agendada.')
    if medico == 2:
      print(f'{usuario}, sua consulta com o Dr. {medicos.medicos[1]} est
á agendada.')
  else:
    print('Agradecemos o seu contato!!!')

else:
  print('Usuário não cadastrado.')
```

Obedecendo o enunciado da questão, faremos algumas alterações tornando nosso programa um pouco mais robusto.

Inicialmente criamos um novo módulo, sendo este o arquivo cadastro_plano_saude.py, onde dentro do mesmo consta a variável usuarios por sua vez parametrizada com um dicionário, onde chaves e valores correspondem ao dígito identificador e ao nome do usuário, respectivamente.

Segunda alteração, ao iniciar a escrita do código de nosso arquivo main.py devemos realizar as importações tanto do módulo medicos quanto do módulo cadastro_plano_saude. Lembrando que estes módulos ao serem importados não precisam da extensão .py, porém os mesmos devem estar no mesmo diretório de nosso arquivo main.py.

Na sequência implementamos a estrutura de validação que irá identificar o usuário para fornecer ao mesmo a opção de agendamento de consulta. Sendo assim, criamos uma variável de nome usuário que por meio da função input( ) pede que o usuário digite seu dígito identificador.

Caso o número fornecido e atribuído a usuario conste nas chaves do dicionário usuarios do módulo cadastro_plano_saude, então é realizada a identificação do usuário, caso contrário, é

exibida uma mensagem de erro dizendo que o usuário é desconhecido ou não está cadastrado na base de dados.

De resto todo o restante do código permanece igual ao exemplo anterior, apenas com a diferença que agora todo esse bloco é indentado à estrutura condicional de validação de usuário.

Supondo que o usuário tenha digitado 003, o retorno será:
Usuário desconhecido ou não cadastrado.

Supondo que o usuário tenha digitado 002, seguido de S para agendar a consulta de 2 para escolher o segundo médico da lista, o retorno será:
Bem-vinda Ana Clara!!!
Escolha com qual médico deseja consultar:
1 – Grazielle Veiga
2 – Matheus Correa
Ana Clara, sua consulta com o Dr. Matheus Correa está agendada.

---

64 - Crie uma função que recebe parâmetros tanto por justaposição quanto nomeados a partir de uma lista e de um dicionário, desempacotando os elementos e reorganizando os mesmos como parâmetros da função:

```
numeros = (33, 1987, 2020)
dados = {'Nome':'Fernando', 'Profissão':'Engenheiro'}

def identificacao(*args, **kwargs):
  print(args)
  print(kwargs)

identificacao(*numeros, **dados)
```

*Aqui temos um exercício com uma proposta um pouco diferente dos exemplos anteriores onde para *args tínhamos apenas um dado/valor. Em Python é perfeitamente possível passar mais de um dado/valor que ocupará por justaposição campos de parâmetros referentes a *args, desde que os dados/valores estejam em formado de set/tupla ou conjunto. Já para os dados em **kwargs sabemos que estaremos repassando dados/valores como parâmetros nomeados, ou seja, com valores padrão pré-definidos.*

Para nosso exemplo temos uma variável de nome numeros que atribuída para si tem uma tupla com alguns números, da mesma forma demos uma variável de nome dados que como atributo tem um dicionário com alguns dados organizados em chaves e valores.

*Na sequência criamos nossa função identificacao( ) que receberá *args e **kwargs, exibindo em tela tais dados.*

*Por fim chamamos nossa função identificacao( ) repassando como parâmetros em *args para a mesma os dados da variável numeros, para isso instanciamos *numeros com o marcador " * " referente a *args. O mesmo processo é feito, repassando o conteúdo de dados como **dados, assim ocupando os campos em **kwargs.*

O retorno será:

(33, 1987, 2020)
{‘Nome’ : ‘Fernando’, ‘Profissão’ : ‘Engenheiro’}

---

65 - Crie uma classe de nome Carro e lhe dê três atributos: nome, ano e cor.

```
class Carro:
  def __init__(self, nome, ano, cor):
    self.nome = nome
    self.ano = ano
    self.cor = cor
```

Retomando o básico de orientação a objetos, pela sintaxe uma classe deve ser identificada pela palavra reservada class e por convenção costumamos dar um nome para a classe iniciado com letra maiúscula.

Sendo assim, declaramos nossa classe Carro, dentro do corpo dessa classe criamos o método construtor def __init__( ) por sua vez com os atributos de classe nome, ano e cor, além da palavra reservada self, referente ao escopo da classe. Lembrando que uma classe simples pode ou não ter um método construtor.

Indentado ao bloco do método construtor criamos os objetos self.nome, self.ano e self.cor que receberão dados quando repassados como atributos de classe.

---

66 - Crie uma classe Pessoa, instancie a mesma por meio de uma variável e crie alguns atributos de classe dando características a essa pessoa. Por fim exiba em tela alguma mensagem que incorpore os atributos de classe criados:

```
class Pessoa:
  pass

pessoa1 = Pessoa()
pessoa1.nome = 'Fernando'
pessoa1.idade = 33
pessoa1.profissao = 'Professor'

print(f'{pessoa1.nome}, {pessoa1.idade} é um {pessoa1.profissao}...'
)
```

Inicialmente criamos nossa classe Pessoa, inicialmente vazia.

Em seguida declaramos uma variável de nome pessoa1 que instancia a classe Pessoa( ), criando os atributos de classe nome, idade e profissao com seus respectivos dados.

Por fim via função print( ) e f'strings criamos uma mensagem onde nas máscaras de substituição instanciamos os objetos criados anteriormente.

O retorno será:
Fernando, 33, é um Professor...

---

67 - Crie uma classe que armazena algumas características de um carro, em seguida crie dois carros distintos, de características diferentes, usando da classe para construção de seus objetos/variáveis.

```
class Carro:
  ano = 2020
  cor = 'prata'
  modelo = 'hatch'
  opcionais = 'nenhum'

carro1 = Carro()

carro2 = Carro()
carro2.ano = 2016
carro2.cor = 'preto'
carro2.modelo = 'sedan'
```

```
print(f'Carro nº1: Ano = {carro1.ano}, Cor = {carro1.cor}, Modelo = {c
arro1.modelo}, Opcionais = {carro1.opcionais}')

print(f'Carro nº2: Ano = {carro2.ano}, Cor = {carro2.cor}, Modelo = {c
arro2.modelo}, Opcionais = {carro2.opcionais}')
```

Criada nossa classe Carro, dentro da mesma criamos alguns objetos simples ano, cor, modelo e opcionais com dados/valores padrão 2020, 'prata', 'hatch' e 'nenhum', respectivamente.

Na sequência criamos uma variável de nome carro1, que por sua vez instancia a classe Carro( ), automaticamente herdando suas características de seus atributos de classe.

De forma parecida criamos uma variável carro2, porém essa não somente instancia a classe Carro( ) mas modifica alguns dos atributos de classe para gerar um veículo diferente de carro1.

Por fim, exibimos em tela via função print( ) as características de ambos os carros fazendo uso de f'strings inserindo nas máscaras

de substituição os dados/valores dos atributos de classe referentes a cada veículo.

O retorno será:
Carro nº1: Ano = 2020, Cor = prata, Modelo = hatch, Opcionais = nenhum
Carro nº2: Ano = 2016, Cor = preto, Modelo = sedan, Opcionais = nenhum

---

68 - Crie uma classe Pessoa com método inicializador e alguns objetos de classe vazios dentro da mesma que representem características de uma pessoa:

```
class Pessoa:
  def __init__(self):
    self.nome = None
    self.idade = None
    self.altura = None
    self.peso = None
    self.sexo = None
```

Atendendo o enunciado da questão, criamos uma classe de nome Pessoa, dentro de sua estrutura criamos o método inicializador/construtor def __init__(self), assim como os objetos self.nome, self.idade, self.altura, self.peso, self.sexo, todos definidos como None.

Aqui temos dois comportamentos a ser entender, primeiro deles que como atributos de classe não foi definido nada além de self, sendo assim, ao instanciar essa classe não temos a obrigatoriedade de repassar "parâmetros" para a mesma, ainda assim todos seus objetos internos são perfeitamente instanciáveis.

Outro ponto a se destacar é que não podemos criar variáveis/objetos vazios em Python (no sentido de não lhes passar algum atributo no momento da declaração, porém, podemos atribuir para esses objetos a palavra reservada None, que identificará para o interpretador que aquele objeto inicialmente não possui nenhum dado/valor, mas que mesmo assim sua estrutura é iterável.

---

69 - Crie uma classe Inventario com os atributos de classe pré-definidos item1 e item2, a serem cadastrados manualmente pelo usuário, simulando um simples carrinho de compras:

```
class Inventario:
  def __init__(self, item1, item2):
    self.item1 = item1
    self.item2 = item2
```

```
cliente1 = Inventario('Camisa Adidas Tam GG', 'Calça Jeans Tam 50'
)

print(cliente1.item1)
print(cliente1.item2)
```

De forma parecida com alguns exercícios anteriores, aqui criaremos uma classe que servirá como um molde para criação de outros objetos. Como esses objetos devem poder ser acessados tanto de dentro quanto de fora da classe, é interessante que criemos para a classe um método inicializador/construtor.

Sendo assim, criamos nossa classe Inventario com seu método inicializador __init__(self), além é claro dos atributos de classe item1 e item2 conforme o enunciado.

Também criamos os objetos self.item1 e self.item2 que receberão os dados/valores dos atributos de classe.

Na sequência criamos uma variável de nome cliente1, que instancia nossa classe Inventario( ) "parametrizando" a mesma com 'Camisa Adidas Tam GG' e 'Calça Jeans Tam 50', respectivamente.

Por meio de nossa função print( ) é possível visualizar se de fato tais objetos foram criados e associados a variável cliente1.

O retorno será:
Camisa Adidas Tam GG
Calça Jeans Tam 50

---

70 - Crie uma classe Biblioteca que possui uma estrutura molde básica para cadastro de um livro de acordo com seu título, porém que espera a inclusão de um número não definido de títulos. Em seguida cadastre ao menos 5 livros nessa biblioteca:

```
class Biblioteca:
  def __init__(self, livro1, **kwargs):
    self.livro1 = livro1
```

```
prateleira1 = Biblioteca('LIVRO TESTE')
prateleira1.livro2 = '1984 - George Orwell'
prateleira1.livro3 = 'Duna - Frank Herbert'
prateleira1.livro4 = 'O Iluminado - Stephen King'
prateleira1.livro5 = 'O Exorcista - William Peter Blatty'
prateleira1.livro6 = 'O Hobbit - J. R. R. Tolkien'

print(prateleira1.livro4)
print(prateleira1.livro1)
```

*Criada a classe Biblioteca, em seu método inicializador/construtor temos uma particularidade, como não sabemos ao certo quantos livros irão compor essa biblioteca, uma vez que seu acervo será atualizado ao longo do tempo e não todo inicialmente, podemos usar de **kwargs como atributo de classe.*

*Como visto anteriormente em exercícios de funções, sempre que precisarmos usar de múltiplos parâmetros, porém não temos seu número fixo definido inicialmente, podemos reservar o espaço para tais parâmetros via *args e **kwargs. Em nosso exemplo estaremos criando objetos os quais possuem como atributo um título de um livro, estrutura muito parecida com a de um parâmetro nomeado.*

*Dessa forma, usando **kwargs como atributo de classe podemos perfeitamente repassar para esse campo o atributo com*

seu respectivo dado/valor tendo em vista que o mesmo será reconhecido como chave e valor de um dicionário.

*Abstraindo essa ideia em código, no método inicializador de nossa classe Biblioteca criamos um atributo de classe livro1 que associará seu dado para o objeto self.livro1 = livro1, para todos os demais, à medida que formos criando novos objetos os mesmos estarão ocupando os campos de **kwargs.*

Já fora do escopo da classe Biblioteca, criamos uma variável de nome prateleira1 que instancia e inicializa nossa classe Biblioteca( ) repassando para a mesma 'LIVRO TESTE' que por sua vez alimentará o campo de livro1.

*Em seguida, ao aplicar o método prateleira1.livro2, estaremos criando o objeto self.livro2 e o mesmo estará indexado para **kwargs. O processo é repetido para criação de 5 livros diferentes conforme o enunciado.*

Por fim, para verificar se a lógica estava correta, por meio de nossa função print( ) por sua vez parametrizada com alguns dos livros cadastrados, podemos verificar o conteúdo destes objetos.

O retorno será:
LIVRO TESTE
O Iluminado – Stephen King

---

71 - Crie uma calculadora simples de 4 operações (soma, subtração, multiplicação e divisão) usando apenas de estrutura de código orientada a objetos:

```
class Calculadora:

  def soma(self, n1, n2):
    return n1 + n2

  def subtracao(self, n1, n2):
    return n1 - n2

  def multiplicacao(self, n1, n2):
    return n1 * n2

  def divisao(self, n1, n2):
    return n1 / n2

n1 = Calculadora()

n1.multiplicacao(5, 4)
```

Criada a classe Calculadora, podemos criar métodos de classe, um para cada operação matemática, de forma que a variável/objeto que instanciar essa classe pode individualmente usar da função matemática a qual se faz necessária.

Para operação de soma, por exemplo, definimos soma( ) que recebe dois números n1 e n2, retornando para a classe a soma dos conteúdos desses parâmetros.

O mesmo é feito para cada operação, retornando apenas o valor da função matemática da mesma.

No corpo geral de nosso código criamos uma variável de nome n1 que instancia e inicializa nossa classe Calculadora( ), em seguida usando do método de classe necessário, parametrizando o mesmo com os valores a serem considerados na operação matemática.

O retorno será:
20

---

72 - Mostre via terminal a string ‘Bem vindo ao mundo da programação!!!’ de trás para frente usando indexação:

```
variavel = 'Bem vindo ao mundo da programação!!!'

print(variavel[::-1])
```

Declarada a variável variavel1 que por sua vez possui como atributo a string ‘Bem Vindo ao mundo da programação!!!, para exibir em tela essa string de trás para frente basta que para nossa variável usemos da notação [::-1], uma vez que desse modo estamos lendo todos os caracteres da string, porém realizando a leitura de cada elemento do último para o primeiro.

O retorno será:
!!!oãçamargorp ad odnum ao odniv meB

---

73 - Escreva um programa que encontre todos os números que são divisíveis por 7, mas que não são múltiplos de 5, entre 2000 e 2200 (ambos incluídos). Os números obtidos devem ser impressos em sequência, separados por vírgula e em uma única linha:

```
lista=[]
for i in range(2000, 2201):
    if (i%7==0) and (i%5!=0):
        lista.append(str(i))

print(','.join(lista))
```

Aqui temos um daqueles exercícios clássicos onde no enunciado são exigidas várias coisas a serem realizadas, necessitando que desmembremos essas instruções em partes para poder entender a lógica a ser utilizada no exercício.

Inicialmente criamos uma variável lista que inicialmente é uma lista vazia onde iremos inserir os números divisíveis por 7 que não são múltiplos de 5.

Em seguida criamos um laço for para poder percorrer o intervalo numérico definido na questão, nesse caso entre 2000 e 2200, retornando o valor de cada leitura a variável temporária i. Lembrando que ao definir o último número do intervalo devemos acrescentar 1 ao seu valor, pois o último elemento lido na verdade servirá de gatilho para que o laço interrompa seu processo.

Dentro desse laço criamos a estrutura lógica/condicional que resolve o problema proposto na questão. Aqui definimos que se o módulo da divisão de i por 7 for igual a 0 e o módulo da divisão de 5 for diferente de 0, esse número será inserido na lista via método append( ).

Repare que aqui temos uma estrutura condicional de lógica composta, em função de usar do operador lógico " and ", as duas proposições devem ser verdadeiras para que o número em questão atenda a condição que lhe é imposta.

Por fim, via função print( ) exibimos em tela os valores que foram inseridos em nossa variável lista, separados por vírgula conforme o exigido pelo enunciado da questão.

O retorno será:
2002, 2009, 2016, 2023, 2037, 2044, 2051, 2058, 2072, 2079, 2086, 2093, 2107, 2114, 2121, 2128, 2141, 2149, 2156, 2163, 2177, 2184, 2191, 2198

---

74 - Escreva um programa, uma calculadora simples de 4 operações, onde o usuário escolherá entre uma das 4 operações (soma, subtração, multiplicação e divisão). Lembrando que o usuário digitará apenas dois valores, e escolherá apenas uma operação matemática do menu.

```
print('CALCULADORA DE 4 OPERAÇÕES')
print('+ -> Soma')
print('- -> Subtração')
print('* -> Multiplicação')
print('/ -> Divisão')
a = float(input("Digite o primeiro número:"))
b = float(input("Digite o segundo número:"))
operacao = input("Digite a operação a realizar (+, -, * ou /):")

if operacao == "+":
 soma = a + b
 print(f'O resultado da soma é: {soma}')
elif operacao == "-":
 subtracao = a - b
 print(f'O resultado da subtração é: {subtracao}')
elif operacao == "*":
 multiplicacao = a * b
 print(f'O resultado da multiplicação é: {multiplicacao}')
elif operacao == "/":
 divisao = a / b
 print(f'O resultado da divisão é: {divisao}')
else:
 print("Operação inválida!")
```

De acordo com o enunciado da questão, inicialmente criamos um menu de opções a ser mostrado para o usuário, seguido da interação com o mesmo para que dê entrada nos valores e escolha a operação a qual realizar com tais números.

O que diferencia esse exercício de outros propostos para uma "calculadora" é o uso de estruturas condicionais onde apenas um

desfecho é permitido.

No início de nossa estrutura condicional simplesmente estamos verificando se o dado/valor de nossa variável operacao é igual a "+", caso essa condição seja verdadeira é realizada a soma dos valores atribuídos às variáveis a e b, exibindo em tela o resultado dessa soma.

Todo o resto é feito da mesma forma, apenas alterando a operação matemática em si e a mensagem de retorno, porém fazendo o uso de elif para que apenas uma dessas condições seja considerada verdadeira, de modo que quando o usuário selecionar uma, todas as outras serão descartadas, e ao término do processo da operação escolhida todo o programa é encerrado.

O retorno será:
CALCULADORA DE 4 OPERAÇÕES
+ -> Soma
- -> Subtração
* -> Multiplicação
/ -> Divisão
Digite o primeiro número: 6
Digite o segundo número: 8
Digite a operação a realizar (+, -, * ou /)
O resultado da multiplicação é 48.0

---

75 - Crie uma função que recebe um número e retorna o mesmo dividido em duas metades, sendo cada metade um elemento de uma lista:

```
def divide_pela_metade(n):
  return [n//2, n - n//2]

n = int(input('Digite um número: '))
print(divide_pela_metade(n))
```

Aqui temos um problema simples de resolver desde que entendamos a lógica a ser implementada. Note que o que é pedido na questão é que não somente o número seja dividido "ao meio", mas que cada parte seja um elemento de uma lista.

Sendo assim, criamos nossa função divide_pela_metade( ) que recebe como parâmetro um número. Dentro do corpo dessa função temos um retorno, onde já criamos a estrutura de uma lista, sendo na posição do primeiro elemento a operação de dividir o mesmo por 2, assim como na posição do segundo elemento estaremos subtraindo o valor do número pela divisão dele mesmo por 2.

Assim teremos em cada posição dessa lista um elemento representando a metade do número repassado anteriormente.

Por fim, criamos a estrutura que via input( ) interage com o usuário pedindo que o mesmo insira um número, atribuindo esse valor a variável n.

Finalizando, diretamente em nossa função print( ) podemos chamar a função divide_pela_metade( ) parametrizando a mesma com o valor de n.

Supondo que o usuário tenha digitado 26, o retorno será:
[13, 13]

---

76 - Crie um programa que gera um dicionário a partir de um valor digitado pelo usuário, de modo que serão exibidos todos valores antecessores a este número multiplicados por eles mesmos. Supondo que o usuário tenha digitado 4, a saída deve ser {1: 1, 2: 4, 3: 9, 4: 16} :

```
numero = int(input('Digite um número: '))
dicionario = dict()

for i in range(1, numero + 1):
    dicionario[i] = i * i

print(dicionario)
```

De acordo com o exercício proposto, inicialmente criamos uma variável de nome numero que recebe do usuário um número. Também criamos uma variável de nome dicionario que atribuída para si tem um gerador de dicionário, inicialmente sem dados compondo o mesmo.

Na sequência por meio de um laço for percorreremos todos elementos iniciando em 1 até chegar no número digitado pelo usuário. Lembrando de validar o número acrescentando 1 unidade ao mesmo como gatilho de encerramento do laço for.

Dentro do corpo do laço for instanciamos nosso dicionario, agora a cada laço iterando com uma posição do mesmo, inserindo nesse campo o valor de i multiplicado por ele mesmo.

Por fim, por meio da função print( ) exibimos em tela o conteúdo de nosso dicionário.

Supondo que o usuário tenha digitado 14, o retorno será:
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81, 10: 100, 11: 121, 12: 144, 13: 169, 14: 196}

---

77 - Defina uma função que pode aceitar duas strings como entrada, exibindo em tela apenas a string de maior tamanho/comprimeito. Caso as duas strings tenham mesmo tamanho, exiba em tela as duas:

```
def mostra_msg_maior(s1, s2):
    len1 = len(s1)
    len2 = len(s2)
    if len1 > len2:
        print(f'{s1} é a string de maior tamanho.')
    elif len2 > len1:
        print(f'{s2} é a string de maior tamanho.')
    else:
        print(f'{s1} e {s2} são strings de mesmo tamanho.')
        print(s1)
        print(s2)
```

Pelo enunciado da questão é possível ver de imediato que aqui temos um problema de lógica simples, bastando comparar as strings e instanciar a maior delas em uma mensagem de retorno.

Sendo assim, inicialmente criamos nossa função mostra_msg_maior( ) que recebe s1 e s2 como parâmetros. Já no corpo da função criamos as variáveis len1 e len2 que por meio do método len( ) verifica o tamanho das strings repassadas para s1 e s2.

Em seguida criamos a estrutura condicional em si, o de definimos que se o tamanho de len1 for maior que o tamanho de len2, é exibida em tela o conteúdo da variável s1.

Da mesma forma, caso o tamanho de len2 seja maior que o tamanho de len1, o conteúdo de s2 que é exibido em tela via função print( ).

Por fim, caso nenhuma das duas condições anteriores seja verdadeira, subentende-se então que as strings em s1 e s2 tem o mesmo tamanho, então exibimos as duas em tela.

No corpo geral de nosso código chamamos nossa função mostra_msg_maior( ) parametrizando a mesma com duas strings "Fernando" e "Python".

O retorno será:
Fernando é a string de maior tamanho.

---

78 - Escreva um programa que recebe um texto do usuário e o converte para código Morse, exibindo em tela o texto em formato Morse, seguindo a padronização “ . - " (ponto, traço).

```
def texto_p_morse(texto):
    d = {'A': '.-', 'B': '-...',
         'C': '-.-.', 'D': '-..', 'E': '.',
         'F': '..-.', 'G': '--.', 'H': '....',
         'I': '..', 'J': '.---', 'K': '-.-',
         'L': '.-..', 'M': '--', 'N': '-.',
         'O': '---', 'P': '.--.', 'Q': '--.-',
         'R': '.-.', 'S': '...', 'T': '-',
         'U': '..-', 'V': '...-', 'W': '.--',
         'X': '-..-', 'Y': '-.--', 'Z': '--..',
         '1': '.----', '2': '..---', '3': '...--',
         '4': '....-', '5': '.....', '6': '-....',
         '7': '--...', '8': '---..', '9': '----.',
         '0': '-----', ',': '--..--', '.': '.-.-.-',
         '?': '..--..', '/': '-..-.', '-': '-....-',
         '(': '-.--.', ')': '-.--.-', '!': '-.-.--',
         ' ': ' ', '"': '.----.', ':': '---...'}

    return ' '.join(d[i] for i in texto.upper())

texto = input('Digite o texto a ser convertido para Código Morse: ')
conversor = texto_p_morse(texto)

print(f'{texto} em Código Morse: {conversor}')
```

O primeiro ponto a ser abordado nesse exercício é o fato de que para se converter um determinado texto para código Morse devemos buscar um dicionário que equipare caracteres normais em padrão Morse (ponto, traço), o que é facilmente encontrado com uma rápida pesquisa na internet.

Dessa forma, criamos nossa função texto_p_morse( ) que recebe um texto como parâmetro. Dentro da estrutura de código

dessa função criamos nosso dicionário tradutor de Morse.

Como retorno da função em si, bastaria simplesmente associar chave e valor do dicionário, dando entrada de um caractere e recebendo seu código Morse correspondente, porém, é interessante aqui realizarmos algumas validações.

Primeira delas eliminando todo e qualquer espaço da string fornecida como dado de entrada, uma vez que para Morse não há necessidade de interpretar os espaços, assim como todo e qualquer texto inserido é convertido para maiúsculo, para ser identificado corretamente quando associado ao dicionário.

Validações feitas, agora sim podemos criar um laço for que percorre cada elemento do texto fornecido pelo usuário, retornando do dicionário seu código equivalente.

Já no escopo global de nosso código criamos uma variável de nome texto que via input( ) recebe um texto do usuário, de forma parecida criamos a variável conversor que chama nossa função texto_p_morse( ) parametrizando a mesma com o conteúdo da variável texto.

Por fim, via função print( ), fazendo uso de f'strings criamos uma mensagem interpolando em suas máscaras de substituição o texto original e seu código Morse.

Supondo que o usuário tenha digitado 'Python', o retorno será:
Python em Código Morse: .--. -.-- - .... --- -.

---

79 - Escreva um programa que recebe do usuário um número de 0 a 100 e transcreve o mesmo por extenso. Por exemplo o usuário digita 49, então é retornara a string ‘quarenta e nove’.

```
def num_p_texto(n):
    if n == 0:
        return 'zero'

    unidade = ('', 'um', 'dois', 'tres', 'quatro', 'cinco','seis', 'sete', 'oito', 'n
ove')

    dezena = ('', '', 'vinte e', 'trinta e', 'quarenta e', 'cinquenta e', 'sesse
nta e', 'setenta e', 'oitenta e', 'noventa e')

    dez_x = ('dez', 'onze', 'doze', 'treze', 'quatorze', 'quinze', 'dezesseis
', 'dezessete', 'dezoito', 'dezenove')

    h, t, u = '', '', ''

    if n == 100:
        h = unidade[0] + 'cem'
        n = n % 100
    elif n >= 20:
        t = dezena[n // 10]
        n = n % 10
    elif n >= 10:
        t = dez_x[n - 10]
        n = 0

    u = unidade[n]
    return ' '.join(filter(None, [h, t, u]))

num = int(input('Digite o número a ser convertido: '))

if num <= 100:
  print(f'{num} por extenso é: {num_p_texto(num)}')
else:
```

```
  print('Número Inválido!!!')


if num <= 100:
  print(f'{num} por extenso é: {num_p_texto(num)}')
else:
  print('Número Inválido!!!')
```

Para esse exercício podemos usar de uma abordagem um pouco parecida com a do exercício onde convertemos um certo texto para Morse, porém aqui, temos que levar em consideração que quando estamos escrevendo por extenso um determinado número, existe uma nomenclatura específica para unidades, dezenas, centenas (que não usaremos de acordo com o enunciado do exercício) e dos números intermediários entre 10 a 19.

Sendo assim, também estaremos fazendo a equivalência de um determinado número com um elemento de uma lista via índice para substituir número por texto, mas com algumas diferenças se comparado ao exercício anterior.

Inicialmente criamos nossa função num_p_texto( ) que receberá um número a ser convertido. Dentro do corpo dessa função inicialmente criamos uma estrutura condicional onde se o valor de n for igual a 0, é retornado apenas 'zero', mas caso essa condição não seja verdadeira, aí sim partimos para as conversões número -> texto.

Então criamos uma variável de nome unidade, com números descritos por extenso desde nenhum, 'um' até 'nove'. Como uma espécie de validação, vamos criar essas estruturas em formato de tupla para que não sejam modificadas.

Também criamos uma tupla atribuída a dezena com números descritos desde 'vinte e' até 'noventa e'. O mesmo é feito para a variável dez_x, onde os elementos da tupla correspondem desde 'dez' até 'dezenove'.

Finalizando essa parte criamos as variáveis h, t e u, inicialmente com strings vazias. Caso você não tenha notado, o grande

diferencial aqui, quando comparado ao exercício anterior é que para montar a nomenclatura de um determinado número por extenso, o mesmo terá de percorrer nossas variáveis unidade, dezena e dez_x para montar os blocos que formam o nome do número.

Finalizada a estrutura anterior, criamos agora nossa segunda estrutura condicional, ainda indentada como a anterior, verificando se o valor de n é igual a 100, caso sim, para retornar apenas 'cem' temos de instanciar nossa variável h, atribuindo para a mesma o conteúdo de unidade em sua posição de índice 0 concatenado com a string 'cem'. Por fim, n tem seu valor atualizado com o módulo da divisão de seu valor por 100.

De forma parecida, em nosso elif definimos como condição que se o valor de n for igual ou maior a 20, nossa variável t recebe como atributo o conteúdo de dezena, na posição de índice que corresponde a divisão exata do número por 10. Aqui, n tem seu valor atualizado para o módulo da divisão de seu valor por 10.

E finalmente em nossa terceira estrutura condicional, caso nenhuma das anteriores seja verdadeira, estipulamos que se o valor de n foi igual ou maior que 10, a variável t recebe como atributo o conteúdo de dez_x em posição de índice n subtraído de 10. Finalizando essa etapa, n tem seu valor atualizado simplesmente para 0.

É criada uma última variável dentro da função de nome u, que recebe unidade[n], em outras palavras, recebe o último valor atribuído para n.

Encerrando a função em si, para que possamos concatenar os nomes associados ao número lido simplesmente via método filter( ) juntamos as strings que formaram os últimos dados atribuídos para as variáveis h, t e u, juntando as mesmas com um espaço entre elas.

Em seguida, agora trabalhando fora da função, criamos uma variável de nome num, que via input( ) lê um número a ser convertido, digitado pelo usuário.

Criamos uma última estrutura condicional pois de acordo com o enunciado dessa questão, os números lidos não podem ser maiores que 100, então simplesmente definimos que se o valor de num for igual ou menor que 100, é exibida em tela uma mensagem onde via f'strings interpolamos o valor de num, assim como na segunda máscara de substituição chamamos nossa função num_p_texto( ) parametrizando a mesma com o valor de num.

Caso essa condição não seja válida, simplesmente é retornado ao usuário uma mensagem de número inválido pois subentende-se que o mesmo tenha digitado um número maior que 100, algo que não esperamos em nossa função.

Supondo que o usuário tenha digitado 76, o retorno será:
76 por extenso é: setenta e seis

---

80 - Crie uma função que recebe um nome e um sobrenome do usuário, retornando os mesmos no padrão americano, ou seja, sobrenome primeiro, seguido do nome:

```
def inverte_nome(n):
  nome, sobrenome = n.split()
  return ' '.join([sobrenome, nome])
```

```
pessoa = input('Digite seu nome e sobrenome: ')
pessoa = inverte_nome(pessoa)

print(pessoa)
```

Aqui temos mais um exercício que pode ser resolvido de múltiplas formas, porém, para resolver o que é pedido pela questão, vamos nos ater ao meio considerado mais fácil.

Inicialmente criamos nossa função inverte_nome( ) que receberá um nome a ser tratado. Em seguida, dentro do corpo da função, declaramos duas variáveis nome e sobrenome, que via método split( ) dividirá o conteúdo de n atribuindo cada metade a sua respectiva variável.

Encerrando a função, é retornado ao usuário por meio do método join( ) os dados de sobrenome e nome, nesta ordem, separando cada palavra com um espaço.

Fora da função, criamos uma variável de nome pessoa que via input( ) pede que o usuário digite seu nome e sobrenome.

Na sequência, a variável pessoa chama a função inverte_nome( ) parametrizando a mesma com seu próprio conteúdo, atualizando assim seu conteúdo.

Por fim é exibido em tela o conteúdo da variável pessoa.

Supondo que o usuário tenha digitado Fernando Feltrin, o retorno será:
Feltrin Fernando

---

81 - Crie um programa que gera uma senha aleatória, com um tamanho definido pelo usuário:

```
import random

tamanho = int(input("enter the length of password"))

s  =  "abcdefghijklmnopqrstuvwxyz01234567890ABCDEFGHIJKLMN
OPQRSTUVWXYZ!@#$%^&*()?"

senha =  "".join(random.sample(s, tamanho))

print(senha)
```

Para implementar certas funcionalidades aos nossos códigos, uma prática comum é buscar alguma biblioteca, módulo ou pacote que já disponha de tais funcionalidades, não sendo necessário desenvolver tudo do zero.

Em nosso exercício atual, onde temos um problema o qual se faz necessário a geração de números aleatórios, podemos perfeitamente realizar essa ação por meio de uma função já pronta da biblioteca random.

Inicialmente importamos a biblioteca random por meio do comando import random.

Na sequência criamos uma variável de nome tamanho, que recebe validado como número inteiro um número fornecido pelo usuário para ser usado como parâmetro para o tamanho da senha a ser gerada.

Criamos também uma variável de nome s que por sua vez tem uma string com todos os possíveis caracteres a serem usados para a senha gerada.

Na próxima etapa criamos uma variável de nome senha que por sua vez recebe como atributo o retorno do método random.sample(

), parametrizado com o conteúdo de s e com o tamanho definido em tamanho.

Para finalizar, é exibido em tela a senha gerada por meio da função print( ), parametrizada com o conteúdo de senha.

Em meu caso, ao digitar 14 quando questionado, o retorno foi:
sfC^1b(4?8rqDh

*Importante salientar que fazendo o uso da biblioteca random, a cada execução será gerada uma senha diferente, como é esperado.

---

82 - Crie uma função que exibe em tela tanto o conteúdo de uma variável local quanto de uma variável global, sendo as variáveis de mesmo nome, porém uma não substituindo a outra:

```
def msg():
    global mensagem
    print(mensagem)
    mensagem = 'Usufrua das funcionalidades do sistema.'
    print(mensagem)

mensagem = 'Olá, seja bem-vindo(a)!'
msg()
```

Para esse exercício, de acordo com o que é pedido em seu enunciado, usaremos de duas variáveis de mesmo nome, mas propósitos e escopos diferentes. Logo, para que não haja conflito entre as mesmas ou que essas variáveis não se sobreponham/atualizem, devemos usar de algum marcador que as separe quanto seu escopo.

Inicialmente criamos nossa função msg( ), sem parâmetros mesmo. Para instanciarmos a variável mensagem do escopo global do código (de fora do corpo da função), devemos usar o marcador global antes do nome da variável. Em seguida simplesmente exibimos em tela seu conteúdo via função print( ).

Na sequência, é criada outra variável de nome mensagem, esta sem marcador nenhum, fará parte do escopo local da função. Também exibimos em tela seu conteúdo via print( ).

Pela própria leitura léxica, na primeira linha do corpo de nossa função msg( )é lida e instanciada a variável mensagem do escopo global, graças ao seu marcador global, na quarta linha do corpo da função quando instanciamos uma variável de nome mensagem, essa terá o comportamento de outra variável, dessa vez local.

O retorno será:
Olá, seja bem-vindo(a)!

Usufrua das funcionalidades do sistema.

---

83 – Crie um programa que recebe um número digitado pelo usuário, convertendo o mesmo para algarismo de número Romano, exibindo em tela esse dado já convertido:

```
class Conversor:
    def int_para_romano(self, num):
        val_int = [
            1000, 900, 500, 400,
            100, 90, 50, 40,
            10, 9, 5, 4,
            1
            ]
        val_rom = [
            "M", "CM", "D", "CD",
            "C", "XC", "L", "XL",
            "X", "IX", "V", "IV",
            "I"
            ]
        numero_romano = ''
        i = 0
        while  num > 0:
            for _ in range(num // val_int[i]):
                numero_romano += val_rom[i]
                num -= val_int[i]
            i += 1
        return numero_romano
```

```
num = int(input('Digite um número a ser convertido: '))

print(Conversor().int_para_romano(num))
```

De modo parecido com o feito nos exercícios de conversão de número para Morse e conversão de algarismo para número por extenso, aqui temos uma aplicação interessante baseada em conversão onde iremos equiparar um número convencional para sua forma escrita em número Romano.

Inicialmente criamos nossa classe Conversor, dentro da mesma o método de classe int_para_romano( ), trabalhando dentro do escopo da classe (self), recebendo um atributo de classe, nesse caso, um número.

Dentro do corpo deste método de classe criamos uma lista de nome val_int, que por sua vez possui toda cadeia de algarismos necessários para que possamos representar qualquer número.

Da mesma forma criamos uma segunda lista, essa de nome val_rom, com os algarismos em sua representação romana, de forma que possa cobrir o mesmo número de algarismos em val_int, assim como expressar todo e qualquer número em sua forma romana.

Na sequência criamos uma variável de nome numero_romano que será atualizada com o número convertido pelo bloco de código seguinte.

É criada uma variável de controle i, inicialmente de valor 0, pois a validação que devemos fazer no processo de conversão se dá no número de elementos a serem convertidos, uma vez que o usuário irá digitar um determinado número e o processo de conversão nesse caso é feito número por número que compõe o número total digitado.

Sendo assim criamos uma estrutura de repetição while, que enquanto o valor de num for maior que 0, é aplicado um laço for que percorre toda extensão de val_int, validando esse número o dividindo por 0. A cada ciclo de repetição número_romano recebe o valor retornado equivalente em val_rom, compondo o novo valor da variável numero_romano.

Para finalizar apenas são criadas as estruturas para receber do usuário um número via função input( ), exibindo em tela o mesmo convertido.

Supondo que o usuário tenha digitado 1237, o retorno será:
MCCXXXVII

---

84 – Crie um programa que mostra a data atual, formatada para dia-mês-ano, hora:minuto:segundo.

```
import datetime

agora = datetime.datetime.now()

print ("Data e hora atual: ")
print (agora.strftime("%d-%m-%Y, %H:%M:%S"))
```

Uma das bibliotecas mais úteis que temos à disposição para situações que envolvam a manipulação de intervalos de tempo é a biblioteca datetime. Esta por sua vez não vem pré-carregada junto a inicialização padrão do Python, sendo necessário fazer a sua importação por meio do comando import datetime.

Em seguida para que mostremos em tela a hora atual de acordo com o relógio interno do sistema basta chamar a função datetime.now( ) da biblioteca datetime.

Como para nosso exercício vamos aplicar uma formatação à nossa data e hora, atribuímos tal função a nossa variável agora.

Em seguida criamos duas funções print( ), uma exibindo apenas uma mensagem e outra parametrizada com nossa variável agora, aplicando sobre a mesma o método strftime( ) que nos permite formatar manualmente como nossa data e hora serão exibidos.

Nesse caso, para exibir a data no formato como estamos habituados, sendo primeiro o dia, seguido do mês, finalizando com o ano, basta que por justaposição passemos como primeiro parâmetro em strftime( ) %s-%m-%Y, assim como para a hora em formato 24h contando até os segundos, repassamos como segundo parâmetro %H:%M:%S.

O retorno será:
Data e hora atual:
02-02-2021, 18:40:12

---

85 – Crie um programa que exibe a versão atual do Python instalada no sistema:

```
import sys

print("Versão do núcleo Python")
print (sys.version)

print("Versão detalhada:")
print (sys.version_info)
```

Importando a biblioteca sys temos acesso a uma série de funcionalidades que dizem respeito a integração do núcleo Python e sua interação com o sistema operacional, funcionalidades estas que podem ser consultadas na documentação.

Para cumprir o enunciado da questão, inicialmente importamos sys por meio do comando import sys.

Em seguida, apenas por convenção, vamos por meio de algumas funções print( ) exibir em detalhes a versão atual da linguagem Python instalada.

O retorno será:
Versão do núcleo Python
3.6.9 (default, Oct 8 2020, 12:12:24)
[GCC 8.4.0]
Versão detalhada:
Sys.version_info(major=3, minor=6, micro=9, releaselevel='final', serial=0)

---

86 – A partir de uma lista composta apenas de dados numéricos, gere outra lista usando de list comprehension usando como base a lista anterior, compondo a nova com os valores dos elementos originais elevados ao cubo:

```
lista1 = [1, 2, 3, 4, 5, 6]

lista2 = [i ** 3 for i in lista1]

print(lista1)
print(lista2)
```

A chamada compreensão em Python, comumente aplicada a listas e dicionários, nada mais é do que uma forma reduzida e otimizada de se escrever um código sobre esses tipos de dados.

Usando de compreensão podemos usar normalmente de todo e qualquer tipo de dado assim como todo e qualquer tipo de operação lógica ou aritmética.

Dessa forma, inicialmente criamos uma variável de nome lista1 que recebe uma série de números do tipo int como elementos de uma lista.

Na sequência criamos uma nova variável de nome lista2 que, conforme o enunciado, é composta por cada elemento oriundo de lista1 elevado ao cubo.

Sendo assim, o método convencional é por meio de um laço for percorrer cada elemento de lista1, a cada ciclo de repetição elevando o mesmo ao cubo e atribuindo a uma variável temporária i que alimenta cada campo da nova lista, nesse caso lista2.

*Por meio de list comprehension, em lista2 colocamos já de acordo com a sintaxe, colchetes identificando se tratar de uma lista, e dentro da mesma em seu segmento inicial a operação que será aplicada a cada elemento, nesse caso i ** 3 para elevar o mesmo a terceira potência, seguido do laço for, apenas especificando que i é um elemento que deve constar em lista1.*

Dessa forma, lista2 é gerada e tem seu conteúdo atualizado com os valores processados a partir de lista1 pela list comprehension.

O retorno será:
[1, 2, 3, 4, 5, 6]
[1, 8, 27, 64, 125, 216]

---

87 – Dada uma estrutura inicial de um carrinho de compras com 5 itens e seus respectivos valores, assim como uma função que soma os valores dos itens do carrinho, retornando um valor total. Aprimore a função deste código usando de list comprehension:

```
# Estrutura Inicial
carrinho = []
carrinho.append(('Item 1', 30))
carrinho.append(('Item 2', 45))
carrinho.append(('Item 3', 22))
carrinho.append(('Item 4', 93))
carrinho.append(('Item 5', 6))
total = 0
for produto in carrinho:
  total = total + produto[1]
print(f'O valor total é: R$ {total}')
```

```
# Código Otimizado
total2 = sum([y for x, y in carrinho])
print(f'O Valor total é: R$ {total2}')
```

Conforme o enunciado, é fornecida uma estrutura inicial onde simplesmente temos um carrinho, seguido de algumas linhas de código adicionando manualmente itens neste carrinho, finalizando com uma função que soma os valores dos itens, exibindo em tela o valor total.

Normalmente quando falamos de aprimorar/otimizar um código, estamos falando não só de o trazer para a forma sintática mais atual mas também mais otimizada e reduzida.

Em nosso exemplo, claro que por se tratar de um pequeno bloco de código, o impacto em performance na execução de nosso programa será mínimo, mas em aplicações reais blocos de código

declarados de forma não otimizada podem sim afetar o desempenho de modo que traga uma experiência ruim ao usuário.

Nesse exercício, como seria possível otimizar um simples bloco de código de uma função? A resposta mais lógica seria reescrevendo o mesmo usando de comprehension.

Uma vez que o algoritmo em questão, é percorrer os elementos de uma lista, separando-os individualmente e de acordo com sua posição de índice somar os valores dos mesmos retornando esse dado, esse processo pode muito bem ser reescrito por: total2 = sum([y for x, y in carrinho]), onde usamos a função soma do próprio sistema sum( ), por sua vez parametrizado com um laço for que percorre o carrinho, desempacotando apenas os dados de y e somando estes valores.

Por fim, por meio da função print( ) podemos exibir na composição de uma mensagem o último valor da variável total2.

O retorno será:
O valor total é: R$196
---

88 – Escreva uma função que recebe do usuário um número qualquer e retorna para o mesmo tal número elevado ao quadrado. Crie uma documentação para essa função que possa ser acessada pelo usuário diretamente via IDE.

```
def ao_quadrado(num):
    '''Esta função recebe como parâmetro um número,
       retornando ao usuário o resultado do número
       em questão elevado ao quadrado.
       Ex: ao_quadrado(16) retornará 256.
    '''
    return num ** 2

#print(ao_quadrado(18)) retornará 324
print(ao_quadrado.__doc__)
```

Quando estamos falando em documentação, toda e qualquer estrutura em Python possui tanto sua documentação formal acessível pelo site oficial da linguagem, quanto uma documentação reduzida e dinâmica que pode ser consultada diretamente durante a escrita do código.

Uma boa prática de programação é sempre que possível comentar e documentar nossos próprios códigos, principalmente se os mesmos serão distribuídos para terceiros.

Nesse processo, podemos criar um modelo de documentação dinâmica chamada docstring, que por parte de sintaxe simplesmente é escrita como um comentário de múltiplas linhas (comentário identificado por aspas triplas ‘”) logo nas primeiras linhas do corpo de uma função.

Desse modo, ao se instanciar e inicializar uma função, exibindo em tela seu __doc__ via função print( ) será retornada a docstring da mesma.

Em nosso exemplo é criada uma função de nome ao_quadrado( ) que recebe obrigatoriamente como parâmetro um dado/valor para num. Dentro do corpo da função temos uma docstring, seguido de um retorno padrão definido pela própria expressão matemática que irá pegar o valor de num e elevar ao quadrado.

Nesse caso, parametrizando nossa função print( ) com ao_quadrado.__doc__ o retorno será:

Esta função recebe como parâmetro um número,

Retornando ao usuário o resultado do número

Em questão elevado ao quadrado.

Ex: ao_quadrado(16) retornará 256.

---

89 – Escreva um programa que recebe uma frase qualquer, mapeando a mesma e retornando ao usuário cada palavra com a frequência com que a mesma aparece na frase em questão.

```
frequencia = {}

texto = 'Se alguma coisa pode dar errado, dará errado, e mais, dará
errado da pior maneira, no pior momento e de modo que cause o m
aior / pior dano possível'

for palavra in texto.split():
    frequencia[palavra] = frequencia.get(palavra, 0) + 1

palavras = frequencia.keys()

for i in palavras:
    print(f'{i} = {frequencia[i]}')
```

Entendendo inicialmente a lógica do exercício, uma vez que temos uma string composta de diversas palavras formando uma frase, será necessário desmembrar e percorrer cada uma dessas palavras para que possamos de fato iterar sobre as mesmas, nesse caso contabilizando sua frequência.

Sendo assim, inicialmente criamos uma variável de nome frequencia que recebe como atributo um dicionário vazio, nela, armazenaremos cada palavra e seu número de incidências.

Em seguida temos uma variável de nome texto que possui como atributo uma das leis de Murphy em forma de string.

Para desmembrar e percorrer cada elemento dessa string, realizamos um laço de repetição for que percorrerá cada elemento desmembrado de texto via texto.split( ), a cada laço de repetição atribuindo seu dado para a variável temporária palavra.

Dentro do corpo desse laço for instanciamos nosso dicionário frequencia, criando a chave [palavra] e atribuindo como valor para mesmo o dado oriundo de frequencia.get( ) parametrizado com

palavra, 0, tendo esse valor acrescido em uma unidade cada vez que uma determinada palavra se repetir ao longo do mapeamento da string.

Por fim é criada uma variável de nome palavras, que recebe apenas os dados de frequencia.keys, ou seja, somente as chaves de nosso dicionário.

Para exibir em tela tais palavras e sua frequência, podemos simplificar o processo com um novo laço de repetição, dessa vez percorrendo o conteúdo da variável palavras, a cada ciclo de repetição atribuindo seu dado/valor para a variável temporária i.

Como ação desse laço de repetição, por meio de nossa função print( ), por sua vez parametrizada com uma f'string, instanciamos em suas máscaras de substituição cada palavra com sua respectiva frequência equivalente.

Nesse caso o retorno será:

Se = 1

alguma = 1

coisa = 1

pode = 1

dar = 1

errado = 3

dará = 2

e = 2

mais, = 1

da = 1

pior = 3

maneira, = 1

no = 1

momento = 1

de = 1

modo = 1

que = 1

cause = 1

o = 1

maior = 1

/ = 1

dano = 1

possível = 1

---

90 – Crie um programa que recebe do usuário uma sequência de números aleatórios separados por vírgula, armazene os números um a um, em formato de texto, como elementos ordenados de uma lista. Mostre em tela a lista com seus respectivos elementos após ordenados.

```
numeros = input('Digite os números, separados por vírgula:')

lista = numeros.split(",")
lista.sort()

print(lista)
```

Aqui temos um problema de simples resolução, bastando desmembrar os números digitados pelo usuário os reordenando de forma crescente.

Inicialmente temos uma variável de nome numeros que por meio da função input( ) recebe do usuário uma sequência de números aleatórios.

Em seguida, é criada uma nova variável de nome lista que simplesmente recebe como atributo o conteúdo de numeros separados por vírgula por meio da função split( ).

Na sequência aplicamos sobre nossa variável lista o método sort( ), que neste caso, sem definirmos nenhum parâmetro específico, simplesmente reordenará os elementos em sua forma crescente.

Por fim, exibimos em tela o conteúdo da variável lista por meio de nossa função print( ).

Nesse caso, supondo que o usuário tenha digitado 9, 2, 5, 6, 3, 7 e 0, o retorno será:

[‘0’, ‘2’, ‘3’, ‘5’, ‘6’, ‘7’, ‘9’]

---

91 – Escreva um programa, da forma mais reduzida o possível, que recebe do usuário uma série de nomes, separando os mesmos e os organizando em ordem alfabética. Em seguida, exiba em tela os nomes já ordenados.

```
nomes = [x for x in input('Digite os nomes, separados por vírgula: ').s
plit(', ')]
nomes.sort()

print(', '.join(nomes))
```

Quando estamos trabalhando em certos contextos onde se faz necessário o uso de códigos reduzidos, temos várias alternativas não somente para otimizar nossos códigos em escrita, mas também no que diz respeito a sua performance.

Para isto, temos desde comprehension até expressões reduzidas que simplificam funções. Em nosso exemplo, uma vez que o usuário tenha dado entrada em vários nomes, precisamos percorrer cada um destes nomes para os separar permitindo assim a reorganização dos mesmos conforme pede o exercício.

Da forma mais reduzida o possível, então criamos uma variável de nome nomes que recebe como atributo uma lista, dentro dessa lista criamos um laço for reduzido que percorrerá cada um dos elementos digitados pelo usuário e retornados por meio da função input( ), separando cada um desses elementos por meio do método split( ) por sua vez parametrizado com ‘, ‘ para que o marcador que identificará a separação de cada um dos elementos seja uma vírgula e espaço, condizendo com a mensagem repassada ao usuário na própria função input( ).

A cada laço, um dos nomes é separado e atribuído a variável temporária x, gerando ao final do processo, cada nome como um dos elementos da lista atribuída a nomes.

Na sequência, para ordenar tais elementos em ordem alfabética basta aplicar o método sort( ), sem parâmetros mesmo, em nossa variável nomes.

Por fim, por meio da função print( ), parametrizada com ‘, ‘;joint( ) por sua vez parametrizada com o conteúdo da variável nomes, novamente estamos concatenando tais nomes para exibi-los em tela, agora reordenados conforme pedido no enunciado da questão.

Supondo que o usuário tenha digitado: Ana, Tania, Paulo, Carlos, Maria, Bibiana, o retorno será:

Ana, Bibiana, Carlos, Maria, Paulo, Tania

---

92 – Escreva um simples programa que recebe do usuário um número qualquer, retornando ao mesmo se este número digitado é um número perfeito.

```
x = int(input('Digite um número: '))

def num_perfeito(x):
    soma = 0
    for i in range(1, x):
        if x % i == 0:
            soma += i
    return soma == x

if num_perfeito(x):
    print(f'{x} é um número perfeito!!!')
else:
    print(f'{x} não é um número perfeito...')
```

Inicialmente temos de relembrar o que é um número perfeito, que na matemática nada mais é do que um número o qual pode ser dividido por todos seus divisores. Ex: O número 6 é um número perfeito, uma vez que o mesmo é divisível por 1, por 3 e por 3, de modo que a soma entre 1, 2 e 3 é igual a 6.

Sendo assim, indo para o código, de início criamos uma variável de nome x que recebe, já validando como número inteiro via int( ) um número digitado pelo usuário por meio da função input( ).

Em seguida é criada a função num_perfeito( ) que recebe o conteúdo da variável x como próprio parâmetro.

No corpo dessa função é criada uma variável de nome sima, inicialmente definida com valor 0.

Na sequência, é criado um laço for que percorre um intervalo entre 1 até o valor de x, haja visto que uma das validações que teremos de realizar para reconhecer um número como perfeito é o mesmo ser divisível por seus antecessores.

Logo, nesse laço de repetição criamos uma estrutura condicional que verifica se a divisão inteira entre o valor de x e cada elemento percorrido pelo laço for é igual a 0. Caso essa condição seja verdadeira, o calor de soma é atualizado somando seu valor atual com o valor de i. Caso contrário, soma apenas tem seu valor atualizado com o valor de x.

Dentro dessa lógica, com um laço simples e uma condicional simples conseguimos realizar todas validações necessárias para verificar um número como perfeito ou não.

Por fim, apenas criamos uma estrutura condicional onde se o retorno de num_perfeito( ) por sua vez parametrizado com o valor de x for True, é exibido em tela uma mensagem dizendo que tal valor é perfeito. Do mesmo modo, caso a condição anterior não seja verdadeira, é exibida em tela uma mensagem dizendo que o valor de x não é um número perfeito.

Supondo que o usuário tenha digitado 6, o retorno será:

6 é um número perfeito!!!

Supondo que o usuário tenha digitado 8, o retorno será:

8 não é um número perfeito...

---

93 – Escreva uma função que recebe uma lista de elementos totalmente aleatórios e os ordena de forma crescente de acordo com seu valor numérico:

```
def ordena_lista(lista):
    for i in range(1, len(lista)):
        valor = lista[i]
        indice = i -1
        while indice >= 0:
            if valor < lista[indice]:
                lista[indice + 1], lista[indice] = lista[indice], lista[indice + 1]
                indice -= 1
            else:
                break
    return lista

lista1 = ordena_lista([9, 0, 3, 5, 1, 6, 7, 2, 8, 4])

print(lista1)
```

Para certos contextos, trabalhar com listas de elementos ordenados se faz necessário, logo, é necessário escrevermos um bloco de código dedicado a este propósito, seja atualizando uma lista já existente, seja gerando uma nova lista ordenada a partir de uma lista de elementos desordenados.

Partindo para o código, já de início criamos uma função de nome ordena_lista( ) que recebe como parâmetro o conteúdo de uma lista.

No corpo dessa função inicialmente é necessário criar uma estrutura de código que irá percorrer os elementos da lista, e isso é feito por meio de um laço for, que percorrerá um intervalo entre 1 até o tamanho total da lista por meio de len( ) parametrizado com lista.

A cada ciclo de repetição de nosso laço, um elemento da lista tem seu valor lido e atribuído a variável temporária i.

Dentro do laço, é criada uma variável de nome valor que recebe lista na posição i. De forma parecida, uma variável de nome índice receberá i -1, ou seja, para cada elemento lido em i será atribuído um valor, lembrando que índices em Python por convenção são iniciados em 0.

Ainda dentro do laço for é criada uma estrutura condicional onde enquanto o valor de índice for igual ou maior que 0 (apenas validando que um índice não pode ter valores abaixo de 0 ou "negativos").

Nessa condicional é criado uma outra estrutura condicional que verifica se o valor atribuído a variável valor for menor que lista na posição índice, as variáveis responsáveis por guardar os dados da lista de acordo com seu índice são atualizadas; lista em sua posição índice + 1 recebe o último atribuído para lista em sua posição índice, da mesma forma lista em sua posição índice tem seu valor atualizado para o último valor atribuído a lista na posição índice + 1. Por fim índice tem seu valor decrementado em uma unidade e desse modo temos criado o mecanismo de ordenação dos elementos da lista, rearranjando os mesmos de forma crescente.

Quando a estrutura condicional definida anteriormente já não se fizer mais válida então o laço de repetição é interrompido, retornando o último valor atribuído a lista, que nesse caso, será a mesma com todos seus elementos agora ordenados.

Já fora da função é criada uma variável de nome lista1 que diretamente chama a função ordena_lista( ) parametrizando a mesma com a lista[9, 0, 3, 5, 1, 6, 7, 2, 8, 4].

Por meio de nossa velha conhecida função print( ) parametrizada com lista1 o que ocorre é um aninhamento de funções uma vez que a própria variável lista1 possui como atributo a função ordena_lista( ).

Nesse caso, não havendo nenhum erro de sintaxe o retorno será:

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

---

94 – Crie uma estrutura toda orientada a objetos que recebe do usuário uma string qualquer, retornando a mesma com todas suas letras convertidas para letra maiúscula. Os métodos de classe para cada funcionalidade devem ser independentes entre si, porém trabalhar no escopo geral da classe.

```
class Conversor(object):
    def __init__(self):
        self.s = ""

    def recebe_string(self):
        self.s = input('Digite uma palavra/frase: ')

    def exibe_string(self):
        print(self.s.upper())

frase1 = Conversor()
frase1.recebe_string()
frase1.exibe_string()
```

Seguindo as orientações do enunciado da questão, inicialmente criamos uma classe de nome Conversor que receberá um object.

Dentro do corpo dessa classe inicialmente é criado um método construtor/inicializador __init__( ), dentro do mesmo é criado um atributo de classe de nome s que simplesmente recebe uma string vazia.

Seguindo a indentação, é criado um método de classe de nome recebe_string( ), que pode receber um objeto qualquer, retornando o mesmo para o escopo geral da classe.

No corpo desse método de classe é instanciado o atributo de classe s, atribuindo para o mesmo via função input( ) alguma string digitada pelo usuário.

De forma parecida é criado um novo método de classe de nome exibe_string( ), que por sua vez, obedecendo o enunciado da questão, exibe em tela via print( ) o valor de s, tendo todos seus

caracteres convertidos para letra maiúscula por meio do método upper( ).

De volta ao escopo global do código, é criada uma variável de nome frase1 que instancia a classe Conversor( ). A partir desse ponto, aplicando para frase1 o método recebe_string( ) será pedido que o usuário digite uma palavra ou frase, assim como aplicando o método exibe_string( ) a palavra/frase digitada pelo usuário será exibida em letras maiúsculas.

Nesse caso, digitando Python Fernando Feltrin o retorno será:

PYTHON FERNANDO FELTRIN

---

95 – Escreva de forma reduzida um programa que recebe do usuário um nome e duas notas, salvando tais dados como um elemento de uma lista. Exiba em tela o conteúdo dessa lista.

```
from operator import itemgetter

nomes = []

while True:
    dados = input('Digite seu nome, seguido de duas notas de 0 a 10:
')
    if not dados:
        break
    nomes.append(tuple(dados.split(",")))

print(sorted(nomes, key = itemgetter(0,1,2)))
```

Como bem sabemos, em Python sempre existem muitas formas diferentes de escrever um código para um determinado propósito. De acordo com o enunciado da questão, para salvar como um elemento de lista vários dados de diferentes tipos podemos usar de diversos contêineres de dados, um dos mais eficientes para contextos onde temos poucos dados a serem armazenados são as tuplas. Trabalhando com tuplas podemos usar de um recurso interessante chamado itemgetter da biblioteca operator para instanciar e se trabalhar apenas com um dos elementos de uma lista sem alterar suas propriedades e comportamentos.

Nessa linha de raciocínio, inicialmente importamos da biblioteca operator o módulo itemgetter para fazer uso de seus recursos.

Em seguida é criada uma lista de nome nomes, que inicialmente recebe como atributo uma lista vazia.

Na sequência, para criar o mecanismo que recebe do usuário diferentes nomes com suas respectivas notas, podemos fazer o uso de um laço de repetição, onde enquanto válido, uma variável de

nome dados recebe via input( ) do usuário os respectivos dados de acordo com o pedido na questão.

Indentado dentro da estrutura de repetição é criada uma estrutura condicional if que verifica se não houverem novos dados atribuídos a dados (supondo que o usuário não tenha digitado mais nada e tenha pressionado ENTER pela última vez), se encerra o ciclo de repetição via break.

Cada um dos dados digitados pelo usuário, em cada ciclo de repetição, é adicionado via append( ) para nomes, convertendo tais dados como elementos de uma tupla, separando os mesmos via split( ) usando como marcador de referência onde houver vírgula.

De volta ao corpo geral do código, podemos diretamente por meio da função print( ), parametrizada com sorted( ), por sua vez parametrizada em justaposição com nomes, que é a origem dos dados, sendo para o parâmetro key atribuída a função itemgetter( ), com os marcadores de posição de índice 0, 1 e 2.

Dessa forma, de uma maneira simples, porém elaborada, estaremos interagindo com cada elemento da lista, desempacotando de tal elemento os itens das respectivas posições de índice definidas anteriormente.

Supondo que o usuário tenha digitado Fernando, 8, 9 e Maria, 9, 8, o retorno será:

[(‘Fernando’, ‘8’, ‘9’), (‘Maria’, ‘9’, ‘8’)]

---

96 – Crie um programa que gera o valor de salário de funcionários considerando apenas horas trabalhadas e horas extras, sendo o valor fixo da hora trabalhada R$29,11 e da hora extra R$5,00. Crie uma regra onde o funcionário só tem direito a receber horas extras a partir de 40 horas trabalhadas da forma convencional.

```
valor_hora = 29.11
valor_hora_extra = 5

horas = int(input('Digite o número de horas trabalhadas: ')) * valor_h
ora
adicional = input('Digite o número de horas extras: ') * valor_hora_ex
tra

if horas > 40:
    valor_final = horas + adicional
else:
    valor_final = horas

print(f'Salário base: R${horas}')
print(f'Adicional de horas extras: R${adicional}')
print(f'Remuneração Total: R${valor_final}')
```

De acordo com o enunciado da questão, basicamente teremos de gerar um valor em reais, simulando o salário de um determinado funcionário de acordo com algumas regras.

Em nosso código, inicialmente vamos definir as regras, sendo uma variável de nome valor_hora que recebe seu respectivo valor de 29.11, da mesma forma a variável valor_hora_extra recebe seu respectivo valor 5.

Uma vez definidas as variáveis que regrarão os cálculos, podemos trabalhar a parte lógica da questão.

Para isso criamos uma variável de nome horas que recebe do usuário via input( ), já validando como número inteiro via int( ), um

determinado número de horas trabalhadas. Da mesma forma é criada uma variável de nome adicional que recebe o número referência para as horas extras.

Na sequência é criada uma estrutura condicional que verifica se o valor de horas é superior a 40, de acordo com a regra, caso tal condição seja válida, uma nova variável de nome valor_final recebe como atributo a soma dos valores de horas e de adicional. Caso a condição imposta não seja válida, valor final recebe apenas o valor referente as horas.

Por fim, podemos via print( ) criar alguns retornos elaborados, sendo no primeiro deles, fazendo uso de f'strings, exibindo em tela o valor de horas, o segundo deles o valor de adicional e o terceiro deles o valor_final contextualizados com suas respectivas mensagens.

Supondo que o usuário tenha digitado 68 e 166, respectivamente, o retorno será:

Salário base: R$1979.48

Adicional de horas extras: R$830.0

Remuneração Total: R$2809.48

---

97 – Reescreva o código anterior adicionando um mecanismo simples de validação que verifica se os dados inseridos pelo usuário são de tipos numéricos, caso não sejam, que o processo seja encerrado.

```
valor_hora = 29.11
valor_hora_extra = 5

try:
    horas = int(input('Digite o número de horas trabalhadas: ')) * valor
_hora
    adicional = float(input('Digite o número de horas extras: ')) * valor_
hora_extra

    if horas > 40:
        valor_final = horas + adicional
    else:
        valor_final = horas

except:
    print('Digite apenas números...')
    quit()

print(f'Salário base: R${horas}')
print(f'Adicional de horas extras: R${adicional}')
print(f'Remuneração Total: R${valor_final}')
```

Uma boa prática de programação é, ao longo de nossos códigos, criar certos mecanismos de validação para evitar possíveis exceções cometidas pelo usuário.

Nessa linha de raciocínio, reaproveitando o código anterior podemos simplesmente via try e except rearranjar alguns blocos de código de modo que os mesmos somente serão executados caso nenhuma exceção seja cometida.

Sendo assim, logo em nossa quarta linha de código é inserido um try, onde dentro do corpo do mesmo constam todas as

estruturas tanto para receber os dados do usuário quanto para calcular os devidos valores do salário baseado em tais dados.

Dessa forma, logo nas primeiras linhas do bloco indentado para try, caso o tipo de dado atribuído para horas ou para adicional sejam diferentes de int e float, respectivamente, essa estrutura condicional se torna inválida, não executando nenhuma linha subsequente desse bloco de código, pulando diretamente para o except, que por sua vez exibe uma mensagem de erro via print( ) e encerra totalmente o processo via quit( ).

Caso o usuário digite os valores corretos como esperado, todos os cálculos são realizados, exibindo em tela por meio de nossas funções print( ) declaradas ao final do códigos os respectivos valores de salário.

Supondo que o usuário tenha digitado 68 e 166, o retorno será:

Salário base: R$1979.48

Adicional de horas extras: R$830.0

Remuneração Total: R$2809.48

Simulando o erro, supondo que o usuário tenha digitado 68 e h, o retorno será:

Digite apenas números...

---

98 – Crie um programa que recebe uma nota entre 0 e 1.0, classificando de acordo com a nota se um aluno fictício está aprovado ou em recuperação de acordo com sua nota. A média para aprovação deve ser 0.6 ou mais, e o programa deve realizar as devidas validações para caso o usuário digite a nota em um formato inválido.

```
def calcula_nota(nota):
    if nota < 0 or nota > 1.0:
        print('Pontuação inválida!!!')
        print('A nota deve ser um valor entre 0 e 1.0')
    elif nota == 1.0:
        print('Parabéns, você gabaritou a prova!!!')
    elif nota >= 0.6:
        print('Parabéns, você foi aprovado(a)!!!')
    elif nota < 0.6:
        print('Nota abaixo da média, você está em recuperação!!!')
    else:
        print('Não foi possível computar sua nota!!!')

try:
    nota = float(input('Digite uma nota: '))
except:
    print('Valor inválido!!!')
    print('Use somente números com casas decimais entre 0 e 1.0')
    quit()

print(calcula_nota(nota))
```

Para um exemplo como este basta que usemos da forma correta estruturas condicionais para realizar tanto a classificação por nota quanto as validações de possíveis erros a serem cometidos pelo usuário. Para isso, podemos criar um código bastante enxuto fazendo o uso de if, elif, else, try e except.

Inicialmente criamos uma função responsável por classificar a nota, função essa de nome calcula_nota( ) que recebe como

parâmetro uma nota.

Já dentro do corpo da função, logo na primeira linha já criamos nossa primeira estrutura condicional que verifica se o valor repassado como note é menor que 0 ou maior que 1.0, sendo válida essa condição, é exibido ao usuário uma mensagem de erro via print( ) orientando o mesmo a usar o programa da maneira correta.

Em seguida criamos uma estrutura elif, que verifica se o valor de nome é igual a 1.0, exibindo em tela via print( ) que o aluno gabaritou a prova.

Mais um elif é criado, dessa vez verificando se o valor de nota é igual ou maior a 0.6, caso essa afirmação seja válida, é exibido em tela via print( ) uma mensagem parabenizando o usuário por sua aprovação.

Outro elif é criado, agora verificando se o valor de nota é menor que 0.6, nesse caso exibindo em tela para o usuário uma mensagem dizendo ao mesmo que com base em sua nota está em recuperação.

Por fim é criado um else, apenas exibindo em tela uma mensagem de erro a ser exibida quando nenhuma das possibilidades anteriores forem válidas.

Dessa forma, de acordo com a nota digitada apenas um dos ifs/elifs será validado exibindo em tela sua respectiva mensagem.

De volta ao corpo geral do código, podemos via try criar uma outra estrutura de validação incorporada diretamente sobre a estrutura que receberá via input( ) alguma nota digitada pelo usuário, dessa vez simultaneamente verificando se o valor atribuído a variável nota é do tipo float.

Caso essa condição imposta não seja atendida, é exibido em tela via print( ) algumas mensagens de erro, encerrando o processo via quit( ).

Terminadas as estruturas lógicas de nosso programa, podemos diretamente via função print( ) chamar nossa função calcula_nota( ) por sua vez parametrizada com o valor atribuído a variável nota.

Supondo que o usuário tenha digitado 7, o retorno será:

Pontuação inválida!!!

A nota deve ser um valor entre 0 e 1.0

Supondo que o usuário tenha digitado 0.7, o retorno será:

Parabéns, você foi aprovado(a)!!!

Supondo que o usuário tenha digitado k, o retorno será:

Valor inválido!!!

Use somente números com casas decimais entre 0 e 1.0

---

99 – Crie uma estrutura molde (orientada a objetos) para cadastro de veículos, tendo como características que os descrevem sua marca, modelo, ano, cor e valor. Cadastre ao menos três veículos, revelando seu número identificador de objeto alocado em memória, assim como o retorno esperado pelo usuário quando o mesmo consultar tal veículo.

```
class Carro:
    def __init__(self, marca = None, modelo = None, ano = None, cor
= None, valor = None):
        self.marca = marca
        self.modelo = modelo
        self.ano = ano
        self.cor = cor
        self.valor = valor

    def descricao(self):
        descricao_carro = f'{self.marca}, {self.modelo}, {self.ano}, {self.c
or}, {self.valor}'
        return descricao_carro

carro1 = Carro('Audi', 'A3', 2006, 'Preto', 'R$ 19.900')
carro2 = Carro('Renault', 'Megane', 2010, 'Preto', 'R$ 22.500')
carro3 = Carro('Chevrolet', 'Cruise', 2018, 'Vermelho', 'R$ 49.000')

print(carro1)
print(carro2)
print(carro3)

print(carro1.descricao())
print(carro2.descricao())
print(carro3.descricao())
```

De acordo com o enunciado, para este exercício devemos desenvolver um simples programa para cadastro de veículos, tendo sua estrutura base orientada a objetos para que possamos a reutilizar como molde para cadastro de novos veículos.

Basicamente, aqui estaremos trabalhando com manipulação simples de objetos atributos de classe.

Sendo assim, já de início podemos criar tal estrutura, criando uma classe de nome Carro, onde em seu método construtor / inicializador __init__( ) podemos criar objetos referentes as características do veículo como marca, modelo, ano, cor e valor através de seu construtor self.nome_do_objeto, assim como em seu próprio campo de argumentos atribuindo um valor padrão None para validar casos onde não teremos todas as características do veículo no momento da geração de seu cadastro.

Ainda dentro da classe criamos um método de classe de nome descricao( ), que retorna uma descrição do veículo com base em suas características.

No corpo geral do código são criadas três variáveis, carro1, carro2 e carro3, que instanciam a classe Carro, repassando para a mesma por justaposição algumas características de veículos.

De acordo com o enunciado da questão, para exibirmos o identificador de cada objeto, basta parametrizar o mesmo em nossa função print( ), pois se tratando de uma estrutura orientada a objetos, o retorno padrão ao se instanciar um objeto/variável será seu número identificador e não seu conteúdo.

Para revelar o conteúdo de cada objeto, ao usar o mesmo como parâmetro de nossa função print( ) devemos instanciar também o método de classe responsável por retornar tais dados, nesse caso o método de classe descricao( ).

Não havendo nenhum erro de sintaxe, o retorno será:

<__main__.Carro object at 0x0000018EFD30D190>

<__main__.Carro object at 0x0000018EFD30DB80>

<__main__.Carro object at 0x0000018EFD4270A0>

Audi, A3, 2006, Preto, R$ 19.900

Renault, Megane, 2010, Preto, R$ 22.500

Chevrolet, Cruise, 2018, Vermelho, R$ 49.000

---

100 – Crie um programa que recebe do usuário três números diferentes, retornando para o mesmo qual destes números é o de maior valor. Usar de funções aninhadas para realizar as devidas comparações entre os valores dos números:

```
x = int(input('Digite o primeiro número: '))
y = int(input('Digite o segundo número: '))
z = int(input('Digite o terceiro número: '))

def maior_de_dois(x, y):
    if x > y:
        return x
    return y

def maior_de_tres(x, y, z):
    return maior_de_dois(x, maior_de_dois(y, z))

print(f'O maior número entre {x}, {y} e {z} é: {maior_de_tres(x, y, z)}')
```

Quando estamos trabalhando com expressões lógicas é comum sempre haver diversas formas de solucionar o mesmo problema, cabendo ao desenvolvedor adotar o método o qual julgue o mais simples e funcional.

Nessa linha de raciocínio, uma vez que temos um problema lógico de verificar valores numéricos e extrair informações a partir dos mesmos, uma boa prática de programação (por parte de eficiência) é usar de funções aninhadas, funções que interagem entre si, umas chamando as outras.

Para nosso exemplo, inicialmente declaramos três variáveis de nomes x, y e z que por meio da função input( ) pedem que o usuário digite um número que será atribuído para cada uma destas variáveis.

Na sequência, criamos uma função de nome maior_de_dois( ) que recebe dois números, repassados em forma de parâmetro, oriundos das variáveis x e y.

Dentro do corpo dessa função simplesmente criamos uma estrutura condicional que se o valor de x for maior que o valor de y, é retornado o valor de x, caso contrário, é retornado o valor de y.

Do mesmo modo, criamos uma função de nome maior_de_tres( ), que receberá três parâmetros, x, y e z.

No corpo dessa função diretamente criamos um retorno que instancia e inicializa a função maior_de_dois( ), por sua vez parametrizada em justaposição com x para seu parâmetro x, e com a função maior_de_dois( ) parametrizada com y e z no lugar do parâmetro y.

Dessa forma, de acordo com a estrutura condicional interna a maior_de_dois( ), temos as devidas interações e comparações entre os valores das três variáveis em questão.

Finalizando, diretamente em nossa função print( ), usando de uma f'string, podemos em suas máscaras de substituição instanciar as variáveis x, y e z, assim como chamar a função maior_de_tres( ), por sua vez parametrizada com os dados de x, y e z. Lembrando que caso fosse necessário guardar essa informação, deveríamos atribuir a função maior_de_tres( ) para alguma variável.

Supondo que o usuário tenha digitado 1, 8 e 5, respectivamente (embora a ordem não importe nesse caso), o retorno será:

O maior número entre 1, 8 e 5 é 8

---

101 – Crie um programa que atua como mecanismo controlador de um sistema direcional, registrando a direção e o número de passos executado. Ao final do processo, exiba em tela o número de referência para um plano cartesiano. Ex: Se o usuário digitar CIMA 5, BAIXO 3, ESQUERDA 3 e DIREITA 2, o resultado será a coordenada 2.

```
import math

posicao = [0,0]

while True:
    coordenada = input('Digite a coordenada, seguido do número de p
assos: ')
    if not coordenada:
        break

    movimento = coordenada.split(" ")
    direcao = movimento[0]
    passos = int(movimento[1])

    if direcao == 'CIMA':
        posicao[0] += passos
    elif direcao == 'BAIXO':
        posicao[0] -= passos
    elif direcao == 'ESQUERDA':
        posicao[1] -= passos
    elif direcao == 'DIREITA':
        posicao[1] += passos
    else:
        pass

print(int(round(math.sqrt(posicao[1] ** 2 + posicao[0] ** 2))))
print(posicao)
```

Para esta questão, temos um problema que a primeira vista pode parecer algo complexo, porém nada mais é do que um sistema

que receberá do usuário coordenadas digitadas, mapeando as mesmas e as transformando em um referencial de uma coordenada, de modo que podemos abstrair tal problema em blocos de estruturas condicionais simples.

Para isso, inicialmente importamos a biblioteca math para fazer uso de suas funções.

Na sequência criamos uma variável de nome posicao, que por sua vez recebe em formato de lista dois valores que podemos imaginar como coordenadas x e y, uma vez que o enunciado da questão nos pede coordenadas de um plano cartesiano.

Em seguida criamos uma estrutura while True, dentro do bloco da mesma é criada uma variável de nome coordenada, que via input( ) recebe do usuário uma coordenada e um valor para o número de passos.

Também criamos uma estrutura condicional onde quando não houverem mais dados atribuídos a coordenada, se encerra esse processo. Dessa forma, podemos receber um número ilimitado de coordenadas do usuário, encerrando o processo quando o mesmo não digitar novas coordenadas.

Dando sequência, criamos uma variável de nome movimento, que recebe os dados de coordenada separados por espaço via split( ).

É criada uma variável de nome direção que recebe o valor do elemento de posição 0 da variável movimento. Do mesmo modo é criada uma variável de nome passos, que recebe o dado/valor de posição 1 da variável movimento.

Na sequência é necessário criar a estrutura lógica que armazenará os dados de direção e passos que o usuário forneceu.

Isto é feito por uma série de if, elif e else, sendo o primeiro if verificando se o valor atribuído a variável direcao é igual a 'CIMA', nesse caso atualizando o valor da variável posicao em sua posição 0 somado com o valor de passos.

De forma parecida é criado um elif que verifica se o valor de direcao é igual a 'BAIXO', caso seja, o valor da variável posicao na posição 0 de índice tem seu valor decrementado com o valor de passos.

Mais um elif é criado, agora verificando se o valor de direcao é igual a 'ESQUERDA', caso sim, o valor da variável posicao em sua posição de índice 1 é decrementado com o valor de passos.

Por fim, um último elif que verifica se o valor de direcao é igual a 'DIREITA', caso for, a variável posicao tem seu valor na posição 1 de índice incrementada com o valor de passos.

Finalizando esta parte, é definido um else pass para que caso o usuário digite uma direção ou número de passos inválido nada aconteça.

Uma vez terminada toda a parte lógica do código podemos exibir em tela a coordenada final por meio de nossa função print( ), parametrizada com a raiz da soma dos valores da variável posicao em suas posições de índice 0 e 1 via math.sqrt( ), tendo seu valor final arredondado via round( ).

Da mesma forma, podemos exibir em tela os últimos valores atribuídos para a variável posicao usando a mesma como parâmetro de nossa função print( ).

Simulando exatamente os mesmos passos descritos no enunciado da questão, CIMA 5, BAIXO 3, ESQUERDA 3 e DIREITA 2, o resultado deverá ser a coordenada 2. Nesse caso, o retorno será:

2

[2, -1]

---

102 – Crie uma estrutura orientada a objetos híbrida, para cadastro de usuários, que permita ser instanciada por uma variável de modo que a mesma pode ou não repassar argumentos para seus objetos de classe na hora em que instancia tal classe. Para isso, a classe deve conter atributos/objetos de classe padrão e instanciáveis de fora da classe.

```
class Usuario:
    nome = 'usuário'
    senha = 'padrao'

    def __init__(self, nome = None, senha = None):
        self.nome = nome
        self.senha = senha

usuario1 = Usuario('admin', '00237')
print(f'Usuário registrado: {Usuario.nome}')
```

Aqui temos um exemplo clássico de estrutura orientada a objetos que servirá como molde para criação de novos objetos a partir da mesma, tendo em vista sua estrutura de modo a oferecer compatibilidade para diferentes formas de uso por parte do usuário.

Partindo para o código, inicialmente criamos uma classe de nome Usuario, dentro do corpo da mesma, logo nas primeiras linhas e antes mesmo do método construtor são criados os objetos nome e senha com atributos padrão em formato de string. Desse modo, temos objetos que podem tanto serem instanciáveis por métodos de classe, quanto por métodos estáticos, independentemente do escopo.

Na sequência é criado um método construtor / inicializador __init__( ) que receberá um nome e uma senha, já validando as mesmas em seus campos com valor padrão None, para caso o usuário use dos objetos nome e senha do escopo geral da classe,

os objetos nome e senha do método construtor não sejam inicializados.

De volta ao escopo global do código, é criada uma variável de nome usuario1, que instancia a classe Usuário, repassando como argumentos para a mesma em justaposição 'admin' e '00237' para nome e senha. Dessa forma, a variável usuario1 está criando seus objetos atributos de classe fazendo o uso dos objetos do método construtor da classe.

Por meio da função print( ), fazendo o uso de uma f'string, podemos em sua máscara de substituição instanciar Usuario.nome, internamente o interpretador fará referência ao atributo de classe nome situado no método construtor.

Nesse caso o retorno será:

Usuário registrado: admin

---//---

```
class Usuario:
    nome = 'usuário'
    senha = 'padrao'

    def __init__(self, nome = None, senha = None):
        self.nome = nome
        self.senha = senha
```

```
usuario2 = Usuario()
usuario2.nome = 'tecnico'
usuario2.senha = '12345'
print(f'Usuário registrado: {usuario2.nome} {usuario2.senha}')
```

Simulando a outra possível modalidade de interação, no corpo geral do código é criada uma variável de nome usuario2, que por sua vez apenas instancia a classe Usuario.

Em seguida, é definido que usuario2.nome tem seu dado/valor de atributo a string 'tecnico'. Da mesma forma é atribuído para usuario2.senha a string '12345'.

Nesse caso, toda a interação está ocorrendo instanciando as variáveis nome e senha do escopo geral da classe, e não do escopo do método inicializador.

Por meio da função print( ), exatamente da mesma forma como no exemplo anterior, é possível em uma máscara de substituição de uma f1string incorporar os dados de usuario2.nome e usuario2.senha, sendo que agora o interpretador irá inicializar as variáveis nome e senha apenas do escopo global da classe.

Nesse caso, o retorno será:

Usuário registrado: técnico 12345

---

103 – Gere uma lista com 50 elementos ordenados de forma crescente. Inicialmente usando do método convencional (e do construtor de listas padrão), posteriormente reescrevendo o código usando de list comprehension.

```
lista1 = list()

for num in range(51):
    lista1.append(num)

print(lista1)
```

Resolvendo o problema proposto pela questão em seu método convencional, inicialmente criamos uma variável de nome lista1 que simplesmente instancia o método construtor de listas list( ), sem parâmetros mesmo.

Na sequência, por meio de um laço for e do método range( ) podemos gerar os elementos ordenados para nossa lista. Sendo que a cada laço de repetição um elemento é adicionado a nossa lista1 via append( ), delimitando o número de elementos no método range( ) em 51 uma vez que o último elemento dessa função é apenas o gatilho para o encerramento do intervalo numérico, não tendo seu valor contabilizado.

Por fim, via função print( ), por sua vez parametrizada com o conteúdo da variável lista1, exibiremos em tela os elementos da mesma.

Nesse caso o retorno será:

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50]

---//---

```
lista2 = [num for num in range(51)]

print(lista2)
```

Reescrevendo o código de forma reduzida, usando de list comprehension, para nossa variável lista2 atribuímos uma lista onde é usado um laço for percorrendo cada elemento do intervalo numérico definido como parâmetro para range( ), retornando cada um desses elementos para a variável temporária num.

Lembrando que a ordem dos elementos em comprehension é lida sendo o primeiro num sendo o elemento final da lista, o segundo num a variável temporária do laço for e, nesse caso, range( ) a função a ser usada (podendo também nesse campo ser usado de alguma expressão lógica ou matemática de acordo com o propósito).

Por meio da função print( ), novamente podemos exibir o conteúdo de nossa lista, nesse caso, lista2.

Não havendo nenhum erro de sintaxe, o retorno será:

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50]

---

104 – Reescreva a estrutura de código abaixo, de um simples programa que gera uma lista com as possíveis combinações entre duas outras listas [1,2,3] e [4,5,6], reduzindo e otimizando a mesma via list comprehension:

```
lista1 = []

for x in [1, 2, 3]:
    for y in [4, 5, 6]:
        if x != y:
            lista1.append((x, y))

print(lista1)
```

Para fins de conferência, o retorno deverá ser:

[(1, 4), (1, 5), (1, 6), (2, 4), (2, 5), (2, 6), (3, 4), (3, 5), (3, 6)]

```
lista2 = [(x, y) for x in [1,2,3] for y in [4,5,6] if x != y]

print(lista2)
```

Declarada uma variável de nome lista2, podemos partir direto para o processo de otimização.

Nesse caso, criando um primeiro laço for que percorrerá cada elemento de [1,2,3], retornamos a cada ciclo de repetição um dado/valor para x. Exatamente da mesma forma um segundo laço for percorrerá cada um dos elementos de [4,5,6], retornando seus valores para variável temporária y.

Como parte final da estrutura de comprehension, criamos uma estrutura condicional que simplesmente verifica se o último valor atribuído a x é diferente de y, sendo essa condição verdadeira, retornando seus respectivos valores para (x, y) escritos como primeiro elemento da sentença.

Dessa forma, embora um pouco confusa para iniciantes, reduzimos e otimizamos o mesmo processo escrito de forma

convencional para o enunciado da questão.

Por meio de nossa função print( ), exibindo o conteúdo da variável lista2, podemos ver que de fato o processo foi realizado com sucesso, haja visto que o resultado retornado é o mesmo do resultado apresentado para conferência na própria questão.

[(1, 4), (1, 5), (1, 6), (2, 4), (2, 5), (2, 6), (3, 4), (3, 5), (3, 6)]

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