**Python Para Data Science**

**Curso 1 – Primeiros Passos**

1. **Aula 1 – Função e Linguagem de Alto Nível:**
   1. Interpolação de str com f’{}’: Podemos fazer interpolação/concatenação entre str e texto em uma única linha usando essa técnica:

nome = 'Bruno'  
idade = 21  
print(f'O nome é {nome} e a idade é {idade} anos')

* + 1. Lembrando que podemos fazer essa formatação dessa forma:

nome = 'Bruno'  
idade = 21  
print('O nome é {} e a idade é {} anos'.format(nome, idade))

* + 1. O resultado final no console é o mesmo.
  1. Input(): Recebe uma entrada do usuário e o programa só continua sua execução após sua entrada:

def saudacao():  
 nome = input('Qual o seu nome?\nR: ')  
 idade = int(input('E a sua idade, qual é?\nR: '))  
 return print(f'Olá, {nome}! Você tem a mesma idade que eu: {idade} anos')  
  
saudacao()



* + 1. Lembrando que quando criamos uma função, precisamos executar ela no final do código para que rode, por isso o saudação() lá no fim.
  1. Linguagens de alto nível são aquelas que parecem muito com a linguagem humana. Diferente daquelas que parecem mais com as de máquinas.
     1. A principal vantagem é a facilidade de entendimento.
     2. A desvantagem é que o interpreter precisa traduzir o que escrevemos para linguagem de máquina, para que o computador execute.
     3. No caso das linguagens de baixo nível, elas já estão em linguagem de máquina, não necessitando do interpreter e tendo um desempenho melhor.
  2. O que aprendemos:
     1. Aprendemos o que são variáveis;
     2. Criamos nossa primeira função;
     3. Entendemos a diferença entre linguagens de alto e baixo nível.

1. **Aula 2 – Parâmetros, Condicionais e Conversão de Tipos:**
   1. Função com parâmetro: Passamos como parâmetro para função algo que ela tenha que usar dentro do seu código para executar, mas esse parâmetro é definido fora dela:

def saudacao\_com\_parametros(nome\_da\_pessoa):  
 return print(f'Bom dia, {nome\_da\_pessoa}!')  
  
nome = 'Luiza'  
saudacao\_com\_parametros(nome)

* 1. Podemos usar condições dentro de funções:

idade = 20  
  
def verifica\_se\_pode\_dirigir(idade\_usuario):  
 if idade\_usuario >= 18:  
 return print(f'Você pode dirigir!')  
 else:  
 raise ValueError('Você não tem idade para dirigir!')  
  
verifica\_se\_pode\_dirigir(idade)

* + 1. Optei por colocar um erro, mas poderia colocar somente uma mensagem negativa.
  1. Podemos converter tipos de str para int, int para str, float para str e assim por diante.
     1. Inputs sempre recebem valores em str, mesmo que o valor passado seja um número, caso estamos esperando receber um número e utilizar esse valor em uma verificação que precise de números e não str, podemos fazer isso:
        1. Código recebendo str:

def verifica\_se\_pode\_dirigir\_sem\_parametros():  
 idade = input('Qual a sua idade?\nR: ')  
 if idade >= 18:  
 return print(f'Você tem permissão dirigir!')  
 else:  
 raise ValueError('Você não tem idade para dirigir!')  
  
verifica\_se\_pode\_dirigir\_sem\_parametros()

* + - 1. Convertendo para int, part1:

def verifica\_se\_pode\_dirigir\_sem\_parametros():  
 idade = input('Qual a sua idade?\nR: ')  
 idade = int(idade)  
 if idade >= 18:  
 return print(f'Você tem permissão dirigir!')  
 else:  
 raise ValueError('Você não tem idade para dirigir!')  
  
verifica\_se\_pode\_dirigir\_sem\_parametros()

* + - 1. Convertendo para int, part2 e como prefiro fazer:

def verifica\_se\_pode\_dirigir\_sem\_parametros():  
 idade = int(input('Qual a sua idade?\nR: '))  
 if idade >= 18:  
 return print(f'Você tem permissão dirigir!')  
 else:  
 raise ValueError('Você não tem idade para dirigir!')  
  
verifica\_se\_pode\_dirigir\_sem\_parametros()

* + 1. A mesma lógica se aplica para converter em str e float. Basta colocar str(variável)/float(variável).
  1. Tipos de operadores:



* + 1. Apostila de python e orientação a objetos da caelum: <https://www.caelum.com.br/apostila-python-orientacao-objetos/declarando-e-usando-variaveis/#entrada-do-usurio>.
  1. O que aprendemos:
     1. Criamos uma função que recebe um parâmetro;
     2. Aprendemos a usar as condicionais if e else;
     3. Convertemos a string (str) digitando através do teclado em um valor inteiro (int).

1. **Aula 3 – Listas, Repetições e Tipos Booleanos:**
   1. Quando vemos o tipo das listas recebemos: “list”
   2. A contagem dos elementos de uma lista começa no 0, portanto, se vc quiser pegar o primeiro elemento de uma lista, print o índice 0, o segundo é o índice 1 e assim por diante:

idades = [18,20,22,35,47]  
  
print(idades)  
print(idades[0])  
print(idades[1])



* + 1. Podemos pedir para o índice printar do índice x até o y, fatiando nossa lista, mas lembre-se que o fatiamento é exclusivo, portanto, se pedirmos para printar do 0 até o 2, ele irá printar somente o 18 e 20, que estão nas posições 0 e 1, excluindo o 2:

idades = [18,20,22,35,47]  
  
print(idades)  
print(idades[0])  
print(idades[1])  
print(idades[0:2])



* + 1. Dessa forma, se quisermos printar até o terceiro elemento da lista, precisamos falar para ele printar de 0 até 3:

idades = [18,20,22,35,47]  
  
print(idades)  
print(idades[0])  
print(idades[1])  
print(idades[0:2])  
print(idades[0:3])



* + 1. Se quiser printar de um determinado elemento até o final, podemos passar somente o elemento antes do “:”, ignorando o próximo:

idades = [18,20,22,35,47]  
  
print(idades)  
print(idades[0])  
print(idades[1])  
print(idades[2:])



* + 1. Se quisermos pegar o último elemento de uma lista, basta colocar -1 no índice:

idades = [18,20,22,35,47]  
  
print(idades)  
print(idades[-1])



* + 1. A ordem regressiva continua, ou seja, se quiser o penúltimo item, coloque índice -2, -3 para o anti-penúltimo e assim por diante.
  1. Podemos criar loops e laços dentro de funções, facilitando nosso processo.
     1. Ao invés de criar uma função que recebe uma idade e verifica se a pessoa pode ou não dirigir para cada uma delas em um for, printando isso:

def verifica\_se\_pode\_dirigir(idade\_usuario):  
 if idade\_usuario >= 18:  
 print(f'{idade\_usuario} anos de idade, TEM permissão dirigir!')  
 else:  
 print(f'{idade\_usuario} anos de idade, NÃO tem idade para dirigir!')  
  
idades = [18,20,12,22,15,35,47,17]  
  
for idade in idades:  
 verifica\_se\_pode\_dirigir(idade)



* + 1. Podemos criar uma função que recebe uma lista de idade e já com esse for dentro fazendo a verificação para cada uma:

def verifica\_se\_pode\_dirigir\_loop\_(idades):  
 for idade in idades:  
 if idade >= 18:  
 print(f'{idade} anos de idade, TEM permissão dirigir!')  
 else:  
 print(f'{idade} anos de idade, NÃO tem idade para dirigir!')  
  
idades = [18,20,12,22,15,35,47,17]  
verifica\_se\_pode\_dirigir\_loop\_(idades)



* + 1. Diminuindo o código e deixando mais simples e prático.
  1. Podemos criar listas com valores booleanos:

permissoes = []  
idades = [18,14,22,15,47]  
  
def verifica\_se\_pode\_dirigir(idades, permissoes):  
 for idade in idades:  
 if idade >= 18:  
 permissoes.append(True)  
 else:  
 permissoes.append(False)  
 print(permissoes)  
  
verifica\_se\_pode\_dirigir(idades, permissoes)



* + 1. Utilizamos o append() para adicionar os valores na lista.
  1. O que aprendemos:
     1. Criamos nossa primeira lista (link com operações com listas: <https://www.alura.com.br/artigos/listas-no-python>) no Python;
     2. Aprendemos a utilizar a estrutura de repetição for;
     3. Descobrimos o tipo booleano.

1. **Aula 4 – O sistema de Imports:**
   1. Podemos colocar vários tipos numa mesma lista.
   2. Biblioteca Random: Randrange() aleatoriza um número de um começo a um fim. Exclusivo, portanto se quiser aleatorizar de 0 a 10, precisamos colocar até 11:

from random import randrange  
  
aleatorio = randrange(0, 11)  
print(aleatorio)

* 1. Podemos criar uma lista e ir acrescentando cada vez mais itens aleatórios dentro dela com um for e append:

from random import randrange  
  
notas\_matematicas = []  
  
for notas in range(8):  
 notas\_matematicas.append(randrange(0, 11))  
  
print(notas\_matematicas)



* 1. Essa biblioteca diz que gera números pseudo aleatórios pois se determinarmos uma seed para ela, a “aleatoriedade” dos números será sempre a mesma, deixando de ser igual.
     1. Ou seja, se escolhermos passar a seed 10 para nosso programa, os números do randrange que pedimos para ele gerar serão sempre os mesmos, portanto, aquelas notas matemáticas que colocamos seria os mesmos não importa quantas vezes rodarmos o programa.
  2. O que aprendemos:
     1. Vimos que é possível armazenar elementos de diferentes tipos em uma lista;
     2. Aprendemos como importar o método randrange para gerar números inteiros aleatórios entre 0 e 10.

1. **Aula 5 – Plotando com Matplotlib:**
   1. Importamos a matplotlib.pyplot como plt e aprendemos vários comandos:

x = list(range(1,9))

y = notas\_matematicas

plt.plot(x, y)

plt.title('Notas de matemática')

plt.xlabel('Provas')

plt.ylabel('Notas')

plt.show()

* + 1. Plt é a biblioteca, x e y são respectivamente linha e coluna.
    2. Plt.plot(x, y): basicamente manda fazer o gráfico com as informações das variáveis x e y, onde denominamos que x era uma lista que ia de 1 até 8 (exclusivo) pois é a quantidade de provas de matemática que tiveram, e y são as notas que o aluno tirou em todas elas, também composta por 8 valores com o código seguinte:

from random import randrange

notas\_matematicas = []

for notas in range(8):

  notas\_matematicas.append(randrange(0, 11))

* + 1. Como já tínhamos feito na aula anterior. Criamos uma lista das notas e fizemos um for para dar um append de um valor aleatório de 0 a 10, mimetizando as notas do aluno, em um range de 8 vezes.
    2. Após ter as informações dos valores de x e y, e mandar plotar o gráfico com elas, definimos que o título do gráfico seria ”Notas de Matemática", a etiqueta do eixo x “provas” e a etiqueta do eixo y “notas”, dando esse resultado:



* + 1. Para remover esse “Text” da parte superior usamos o .show(), dando esse resultado final:



* + 1. Podemos melhorar esse gráfico ainda mais, colocando umas bolinhas em cada ponto de cada nota, para fazer isso, incrementamos a linha do plot com mais um parâmetro:

x = list(range(1,9))

y = notas\_matematicas

plt.plot(x, y, marker='o')

# resto do Código…

* + 1. Nos levando a este resultado:



* 1. O que aprendemos:
     1. Aprendemos como importar a "Matplotlib" conforme a convenção como "plt";
     2. Vimos que é possível criar listas a partir do código "range";
     3. Aprendemos como criar gráficos, definindo título, rótulos e marcador, além de ocultar outros detalhes indesejáveis.

**Curso 2 – Introdução à Linguagem e Numpy:**

1. **Aula 1 – Ambiente do Cientista de Dados:**
   1. Todo o conteúdo dessa aula será feito no googlo colab. Link para acessar: <https://colab.research.google.com/drive/1vSPBCjnlEjD74x-vHtJ6Bfat4WyJ9dYy>.
   2. Estamos trabalhando com a biblioteca numpy.
      1. Importamos ela como np para simplificar o uso.
      2. Subimos arquivos txt com vários dados para o notebook para podermos utilizá-los e utilizamos a função loadtxt(‘nome\_do\_arquivo.txt’) da biblioteca, responsável por carregar um arquivo txt num Array numpy:

import numpy as np

km = np.loadtxt('carros-km.txt')

km



* + 1. No notebook não precisamos dar print(km). Só o fato de digitar o km no final da célula já faz a array ser impressa, como acima.
    2. Podemos passar um parâmetro na função loadtxt() dizendo o tipo queremos que essa Array seja:

anos = np.loadtxt('carros-anos.txt', dtype = int)

anos



* + 1. Como primeira operação, fizemos a km\_média de cada carro, basicamente dividindo o km pelo ano atual menos os anos de cada carro:

km\_media = km / (2019 - anos)



* + 1. Nesse caso acabei passando 2019 para ver o mesmo exemplo que foi dado na aula e ver essa mensagem.
    2. Na mensagem basicamente diz que não foi possível fazer o cálculo com alguns carros pois houveram carros que foram feitos em 2019, nesse caso, ele faz o tratamento de dados pra gente e devolve nan (not a number), já que 0 dividido por 0, não existe:



* + 1. Essa seria a km\_media de cada carro.
    2. O tipo de variável quando usamos o Array numpy é exatamente isso:



* + 1. Documentação da função loadtxt(): <https://numpy.org/doc/1.16/reference/generated/numpy.loadtxt.html>.
  1. O que aprendemos:
     1. Os ambientes de desenvolvimento para a linguagem Python;
     2. A carregar dados externos em arrays Numpy;
     3. A trabalhar de forma básica com arrays Numpy.

1. **Aula 2 – Características do Python:**
   1. Operadores matemáticos:
      1. Divisão com //: Devolve a parte inteira da divisão, como no caso de 10 // 3, devolverá apenas 3, ao invés do 3.333333... como em uma divisão normal.



* + 1. \*\*: operador de elevado, ou seja, 23 = 8. Esse é o operador que faz essa conta.



* + 1. %: devolve o resto de uma divisão, ou seja, 10%3 vai ser igual a 1, pois é o resto da divisão. Se fosse 10%2, o resto seria 0. Muito útil para verificação de números ímpares e pares.



* 1. Expressões matemáticas:
     1. Python entende expressões, identifica os operadores e executa a conta na ordem correta, independentemente da posição dos operadores:



* + 1. Para ter certeza que uma operação vai dar certo, podemos separar as contas:





* + 1. \_: Funciona como a memória da calculadora. É uma variável que guarda o último valor de uma operação matemática:





* 1. Formas de atribuição:
     1. =: atribuição direta;



* + 1. +=: soma o valor atual da variável mais o que vier depois



* + 1. -+: subtrai o valor atual da variável mais o que vier depois;



* 1. Atribuição múltipla:
     1. Podemos criar várias variáveis em uma única linha e atribuir os valores à elas respectivamente:

ano\_atual, ano\_fabricacao, km\_total = 2021, 2003, 44410.0



* + 1. Isso nos permite simplicar ainda mais um código. Ao invés de ficar assim:

ano\_atual = 2021

ano\_fabricacao = 2003

km\_total = 44410.0

km\_media = km\_total / (ano\_atual - ano\_fabricacao)

km\_media

* + 1. Ele fica assim:

ano\_atual, ano\_fabricacao, km\_total = 2021, 2003, 44410.0

km\_media = km\_total / (ano\_atual - ano\_fabricacao)

km\_media

* 1. Em str, ao colocar “ ” dentro de ‘ ’ python entende que é uma citação e as “” aparecem no texto final:



* + 1. O contrário também se aplica, se as de fora forem duplas as de dentro devem ser simples:



* + 1. Apas simples 3 vezes serve para escrever em várias linhas:



* + 1. \n: pula linha.
    2. None: indica que a variável é nula:



* + 1. Nem mostra nada quando excuta. Esse é o tipo dela:



* 1. Int(), float(), srt(): todos convertem uns aos outros para seu tipo. Se colocar uma variável dentro de () ela se torna respectivamente um inteiro, um float e uma string. O int() não arredonda, apenas pega a parte inteira, ignorando o pós virgula.
  2. Indentação é o que define o que é um bloco de código.
     1. É a indentação que determina o que está fora ou dentro de um for, if, def etc...
     2. #: É o que define que aquela linha é um comentário. Atalho CTRL + / no pycharm e CTRL + ; no notebook google.
  3. Str.format(variável1, variável2...): formata a str:

print('Ola, {}'.format('Bruno'))



print('Ola, {}! Este é seu acesso de número {}'.format('Bruno', 32))



acessos = 32

print('Ola, {}! Este é seu acesso de número {}'.format('Bruno', acessos))



print('Ola, {nome}! Este é seu acesso de número {acessos}'.format(nome = 'Bruno', acessos = 32))



* + 1. F’str’: formata a str igual o acima:



* 1. O que aprendemos:
     1. A realizar operações matemáticas com Python;
     2. Como criar e atribuir valores a variáveis na linguagem Python;
     3. Os tipos de dados básicos em Python;
     4. A realizar transformações de tipos de dados;
     5. As regras e características básicas da linguagem Python (indentação, comentários e interpolação de strings).

1. **Aula 3 – Trabalhando Com Listas:**
   1. Listas são determinadas por variáveis que recebem [] com várias informações dentro, podendo ser do mesmo tipo ou não:

Carro\_1 = ['Jetta Variant', 'Motor 4.0 Turbo', 2003, 44410.0, False, ['Rodas de liga', 'Travas elétricas', 'Piloto automático'], 88078.64]

* + 1. Podemos inclusive guardar listas dentro das listas, como no exemplo acima, ficando assim no console:



* + 1. Nesse caso, Rodas de liga, por exemplo, não pertence a lista Carros, uma vez que está em uma lista dentro de uma lista, pertencendo a uma lista diferente da de Carros, como no exemplo abaixo.
    2. Com isso, podemos guardar listas (a variável) dentro de outras listas:

Carros = [Carro\_1, Carro\_2]

Carros



* 1. **Lembrando que todas as documentações estão no notebook com link para a documentação oficial e breve resumo delas.**
  2. Operações com listas:
     1. In: Podemos verificar se uma informação está em uma lista perguntando se o elemento x está na lista A, nos devolvendo um booleano:



* + 1. Ele é case sensitive, portando cuidado ao escrever.
    2. Not in: O contrário de in. Ele pergunta se o elemento x não está na lista A.
    3. +: Faz a concatenação de 2 listas:

A = ['Rodas de liga', 'Travas elétricas', 'Piloto automático', 'Bancos de couro']

B = ['Ar condicionado', 'Sensor de estacionamento', 'Sensor crepuscular', 'Sensor de chuva']

A + B



* + 1. Len(A): retorna o tamanho da lista:



* 1. A[i]: Listas possuem índices e podemos acessar qualquer item dela desde que saibamos qual o índice. A contagem inicia em 0:



* + 1. Índices negativos acessam a lista de trás pra frente:



* + 1. Quando temos listas dentro de listas, como no nosso exemplo de Carros, ao passar o índice 0 ele não irá mostrar o primeiro item da primeira lista dentro dele, mas sim a primeira lista toda, já que essa é o elemento 0 da lista Carros:



* + 1. Para acessar itens dentro de uma lista que está dentro de outra lista, precisamos passar dois índices: o primeiro para indicar qual lista dentro da lista queremos acessar e o segundo para indicar qual item da lista dentro da lista queremos acessar:



* + 1. No caso acima acessamos o nome do veículo da primeira lista, ou seja, do primeiro carro, dentro da lista de Carros. Se quisermos o nome do segundo carro, ou seja, primeiro elemento da segunda lista dentro da lista de Carros, precisamos passar o primeiro índice 1 e o segundo 0:



* + 1. Esse processo de acessar elementos dentro de listas dentro de listas que estão dentro de listas continuam indefinidamente:



* + 1. Como podemos ver, conseguimos acessar o primeiro elemento da lista que estava dentro da lista Carros\_1 que estava dentro da lista Carros. Se quiséssemos poderíamos acessar qual elemento de qualquer lista dentro de qualquer lista. Basta alterar o índice.
  1. A[i:j]: Slices.
     1. Conseguimos fatiar listas pegando e exibindo apenas os itens que queremos, como por exemplo, do item 2 até o 3, ou até 7, ou do 5 ao 9 de uma lista. Tudo depende da nossa necessidade e do que queremos fazer com essas listas:



* + 1. O elemento com índice i é incluído e o com índice j é excluído do resultado. Lembre-se que esse método é exclusivo. No caso acima queríamos pegar de piloto automático até Sensor de estacionamento, sendo, respectivamente, os índices 2 e 5, porém, como esse método é exclusivo, precisamos indicar o índice 6 para ele exibir até o 5.
    2. A[i:]: Fatia do índice i até o final da lista, não importa quantos itens tenham:



* + 1. A[:j]: O contrário do acima. Printa do índice 0 até o indicado:



* 1. Métodos de Listas:
     1. A.sort(): Aleatoriza os itens da lista:



* + 1. A.append(x): Adiciona um item no final da lista (Mudei o nome da lista pra ficar mais fácil digitar. Rodei o código 2 vezes, por isso a duplicata):



* + 1. A.pop(i): Remove um item da lista. Se não passar nenhum elemento, ele, por padrão, remove sempre o último item. Retorna o item que removeu:



* + - 1. Removeu o 4x4 que estava duplicado.
      2. Removendo por índice:



* + 1. A.copy(): cria uma cópia da lista A. Muitas vezes achamos que atribuir uma lista a outra lista de nome diferente, faz com que criemos uma cópia dela, quando na verdade, o que estamos fazendo é atribuir uma nova referência à mesma lista, ou seja, todas as alterações que fizermos na lista 2 (que achamos ser a cópia) também serão feitas na lista 1 (original):



* + - 1. Ao criar a copia com essa função, podemos alterar a lista 2 sem problemas de alterar a lista 1:



* + 1. A[:]: Também cria uma cópia da lista A. Igual a função acima:



* 1. O que aprendemos:
     1. Listas, que são um tipo de sequência mutável que podemos utilizar para armazenar coleções de itens;
     2. Formas de criação de listas em Python;
     3. A realizar operações básicas com listas, como a pertinência, concatenação e verificação de características;
     4. Técnicas de seleção de itens e fatiamento com listas do Python;
     5. A utilizar métodos básicos de listas.

1. **Aula 4 – Condicionais e Laços:**
   1. Podemos usar o laço for para printar ou executar qualquer função/fazer qualquer coisa com cada item de uma lista:



* + 1. Range(): passa um tamanho.
    2. List(): cria uma lista.
    3. Juntando os dois temos:



* + - 1. Uma lista com 10 itens de 0 a 9.
    1. Podemos fazer operações matemáticas no for, por exemplo:





* + 1. Podemos simplificar MUITO o código acima, deixando tudo em uma única linha:



* + - 1. Basicamente criamos a função toda dentro de uma lista, economizando tempo e código. Além disso podemos atribuir isso a uma lista da mesma forma que antes:



* 1. Podemos criar loops aninhados, como por exemplo for dentro de for, caso tenhamos listas dentro de listas e queiramos printar todos os itens de todas as listas:

dados = [

    ['Rodas de liga', 'Travas elétricas', 'Piloto automático', 'Bancos de couro', 'Ar condicionado', 'Sensor de estacionamento', 'Sensor crepuscular', 'Sensor de chuva'],

    ['Central multimídia', 'Teto panorâmico', 'Freios ABS', '4 X 4', 'Painel digital', 'Piloto automático', 'Bancos de couro', 'Câmera de estacionamento'],

    ['Piloto automático', 'Controle de estabilidade', 'Sensor crepuscular', 'Freios ABS', 'Câmbio automático', 'Bancos de couro', 'Central multimídia', 'Vidros elétricos']

]

for lista in dados:

  for item in lista:

    print(item)



* + 1. Utilizando esse recurso, podemos criar uma lista vazia e utilizar o for para jogar todos esses itens juntos dentro dela:

acessorios = []

for lista in dados:

  for item in lista:

    acessorios.append(item)

acessorios



* + 1. Set(): remove duplicatas de uma lista. Note que acima temos vários itens iguais em uma lista, então usamos o set para remover:



* + 1. Porém ele cria em type set, para alterar, basta usar o list() como já vimos antes:



* + - 1. Note que ao invés de {} como na imagem anterior, agora temos [], ou seja, lista.
    1. Todo o processo que vimos acima também pode ser resolvido em uma única linha com o list comprehensions:



* 1. Podemos criar condições utilizando listas e for:

# 1º item da lista - Nome do veículo

# 2º item da lista - Ano de fabricação

# 3º item da lista - Veículo é zero km?

dados = [

    ['Jetta Variant', 2003, False],

    ['Passat', 1991, False],

    ['Crossfox', 1990, False],

    ['DS5', 2019, True],

    ['Aston Martin DB4', 2006, False],

    ['Palio Weekend', 2012, False],

    ['A5', 2019, True],

    ['Série 3 Cabrio', 2009, False],

    ['Dodge Jorney', 2019, False],

    ['Carens', 2011, False]

]

for lista in dados:

  if lista[2]:

    print(lista)



* + 1. No código acima tínhamos várias listas dentro de dados e queríamos obter somente os que eram zero km, ou seja, precisávamos printar a lista de itens dentro de dados que possuía o índice 3 como True.
    2. Em condições, caso queiramos dizer que “se algo for verdadeiro faça tal” não precisamos colocar o “se algo == True:”, pois o python tem como padrão esse argumento em verificações if’s.
    3. Do mesmo modo que fizemos antes podemos jogar esse resultado para dentro de uma lista ao invés de printar:

zero\_km\_y = []

for lista in dados:

  if lista[2]:

    zero\_km\_y.append(lista)

zero\_km\_y



* + 1. Também podemos criar condições com else e jogar todos os que não eram zero km para outra lista:

zero\_km\_y = []

zero\_km\_n = []

for lista in dados:

  if lista[2]:

    zero\_km\_y.append(lista)

  else:

    zero\_km\_n.append(lista)

print(zero\_km\_y)

print(zero\_km\_n)



* + 1. Do mesmo modo que antes, podemos utilizar list comprehensions e colocar todo esse código em uma única linha:

[lista for lista in dados if lista[2]]

* + - 1. O resultado será o mesmo da lista zero\_km\_y. Ainda não aprendemos a usar o else nesse curso (fiz porque sou apressado), então ele não ensinou com list comprehensions.
  1. Elif: Usado após um primeiro if, mas ainda existem condições a serem testadas. Podemos terminar uma sequência de condições com um elif, não precisamos necessariamente usar um else.
     1. Python vai testando todas as condições até uma delas ser True:

a, b, c = [], [], []

for lista in dados:

  if lista[1] <= 2000:

    a.append(lista)

  elif lista[1] > 2000 and lista[1] <= 2010:

    b.append(lista)

  else:

    c.append(lista)



* + 1. Outra notação que o python aceita, mas a maioria das outras não, é que ao invés de colocar o AND na condição, podemos simplesmente dizer que a lista está entre os dois valores, assim como em notações matemáticas:

a, b, c = [], [], []

for lista in dados:

  if lista[1] <= 2000:

    a.append(lista)

  elif 2000 < lista[1] <= 2010:

    b.append(lista)

  else:

    c.append(lista)



* + 1. O resultado continua o mesmo e ainda assim economizamos código.
  1. O que aprendemos:
     1. Como utilizar estruturas de repetição e condicionais na linguagem Python;
     2. A construção de laços for;
     3. A iteração em listas do Python;
     4. Loops aninhados em listas de listas;
     5. Cláusulas if, elif e else;
     6. Operadores lógicos e de comparação;
     7. List comprehensions.

1. **Aula 5 – Conhecendo o NumPy:**
   1. Abreviação de Numerical Python.
      1. Um dos pacotes de processamento numérico em python mais importantes.
      2. Além de ser a base para a maioria dos pacotes de aplicações científicas que usam dados numéricos.
      3. A descrição completa está no notebook utilizado para este curso. Link no começo das notas.
   2. Arange(n): função do numpy que cria um Array com uma lista dentro com a quantidade de elementos do n:



* + 1. No caso acima, colocamos np pois importamos a biblioteca numpy como np:

import numpy as np

* + 1. Se tivéssemos somente importado a biblioteca, sem o as... teríamos que escrever o nome todo para executar a função arange:

numpy.arange(10)

* 1. Podemos importar somente funções específicas de uma biblioteca, como no caso do arange():

from numpy import arange

* + 1. Quando fazemos isso, não precisamos mais escrever np/numpy.arange(n) para usar a função, podemos simplesmente usá-la que o resultado é o mesmo:

arange(10)



* 1. Podemos também criar listas em Array com np:

km = np.array([1000,2300,4987,1500])

km



* + 1. A partir do momento que atribuímos uma Array à uma variável, ela altera seu tipo:

type(km)



* + 1. Apesar de ter uma lista dentro da Array, o tipo que a variável que à armazena não é list como se tivéssemos passado somente a lista para ela:

km = [1000,2300,4987,1500]

km



type(km)



* 1. Numpy trabalha com diversos tipos de dados. Para saber qual é o tipo de dados que uma Array tem podemos fazer o seguinte:

km.dtype



* + 1. Na documentação podemos procurar esse int64 e ver do que se trata. Link: <https://numpy.org/doc/1.16/user/basics.types.html>.
  1. Loadtxt(fname = ‘’, dtype = type): Função que carrega dados de um arquivo externo para uma variável, passando o parâmetro do nome e do type de arquivo.

km = np.loadtxt(fname = 'carros-km.txt', dtype = int)

km



* + 1. Se não passarmos o tipo exato, np, por padrão, transforma em float. Passando o parâmetro acima:



* 1. Arrays tem dimensões, assim como matrizes, e para saber a dimensão dela utilizamos o shape:



* + 1. No caso do km que criamos, ele está dizendo que possui 258 linhas, mas, se criarmos outro Array em acessórios utilizando os dados, já visto antes:

dados = [

    ['Rodas de liga', 'Travas elétricas', 'Piloto automático', 'Bancos de couro', 'Ar condicionado', 'Sensor de estacionamento', 'Sensor crepuscular', 'Sensor de chuva'],

    ['Central multimídia', 'Teto panorâmico', 'Freios ABS', '4 X 4', 'Painel digital', 'Piloto automático', 'Bancos de couro', 'Câmera de estacionamento'],

    ['Piloto automático', 'Controle de estabilidade', 'Sensor crepuscular', 'Freios ABS', 'Câmbio automático', 'Bancos de couro', 'Central multimídia', 'Vidros elétricos']

]

acessorios = np.array(dados)

acessorios



* + 1. E tentarmos ver o shape dela, temos que:

acessorios.shape



* + 1. Ele possui 3 linhas e 8 colunas, ou seja, 3 listas com 8 itens dentro, como podemos ver nos dados acima.
  1. Arrays tem desempenho muito melhor do que listas em operações matemáticas, por isso utilizamos arrays.
     1. Para ver se um código está performando melhor um do que outro utilizamos o %time para ver o tempo de execução do código:

np\_array = np.arange(1000000)

py\_list = list(range(1000000))

%time for \_ in range(100): np\_array \*= 2

%time for \_ in range(100): py\_list = [x \* 2 for x in py\_list]





* + - 1. Podemos utilizar o “\_” para ignorar o conteúdo de uma variável, como visto acima, além de como uma memória de calculadora.
    1. Para testar essa hipótese de que Array performa muito melhor do que lists, fizemos o código acima onde criamos uma variável recebendo um Array de tamanho 1M e outra variável recebendo uma lista padrão de mesmo tamanho.
       1. Fizemos um for com o %time antes para analisar a performance, dizendo que para cada informação por 100 vezes faça o np\_array se multiplicar por 2, ou seja, cada item da Array (1M de itens) multiplicado por 2 por 100 vezes.
       2. O mesmo código se repete para a lista do python, com a diferença de que para a lista precisamos colocar outro for para que ele acesse cada item da lista e multiplique por 2, repetindo isso 100x.
       3. Essa é outra vantagem do np. Ele não precisa que passemos essa informação, criando outro for. Ele automaticamente multiplica cada item da lista por 2 por 100x sem precisar passar outro for pra isso, como no caso das listas py.
    2. Podemos ver nos resultados que o tempo de execução de cada um deles, para fazer exatamente a mesma coisa, é absurdamente diferente, onde a array utilizou 60.6 milisegundos e a lista do py 11.9 SEGUNDOS!
    3. Concluímos que, além de ser uma escrita mais prática, evitando a necessidade de escrever 2 for’s para fazer essa execução, como no caso das listas, o tempo de resposta é extremamente mais baixo, performando inúmeras vezes melhor que listas.
    4. Um contra das arrays é que não podemos ter mais de um tipo de dados dentro dela, assim como nas listas.
       1. Ou são ints, ou floats, ou str, ou bool. Não podemos ter todos em um único Array como no caso das listas.
  1. Operações com arrays:
     1. Podemos fazer contas utilizando valores constantes e variáveis, coisa que caso estivéssemos tentando fazer com listas do py, não seria possível:
        1. Listas:

km = [44410., 5712., 37123., 0., 25757.]

anos = [2003, 1991, 1990, 2019, 2006]

idade = 2019 - anos



* + - 1. Arrays:

km = np.array([44410., 5712., 37123., 0., 25757.])

anos = np.array([2003, 1991, 1990, 2019, 2006])

idade = 2019 - anos

idade



* + - 1. Como podemos ver, queríamos saber a idade de cada veículo através da conta do ano atual (considerando o ano do curso) menos o ano de cada carro, ou seja, uma constante menos uma variável.
      2. No caso das listas ocorreu um erro, pois, sem um for, isso não é possível. Já utilizando as arrays, não precisamos do for, portanto, ele automaticamente fez cada ano da variável menos o ano atual, nos ternando o resultado de cada conta na variável idade.
      3. Isso é extremamente prático e facilita muito mais do que ter que fazer todo esse código no caso de listas:

idade = []

for ano in anos:

  idade.append(2019 - ano)

idade



* + - 1. Podemos reduzir o código com list comprehensions, tendo o mesmo resultado:

[(2019 - ano) for ano in anos]



* + 1. Também podemos fazer operações entre arrays, que, obviamente, são muito mais fáceis que entre listas, uma vez que não temos que fazer um for para cada operação:

km\_media = km / idade

km\_media





* + - 1. Já vimos esse aviso antes. Está dizendo que não existe 0 / 0 então retorna um nan.
      2. Basicamente a conta que ele está fazendo é:





* 1. Operações com arrays de 2 dimensões:
     1. Criamos a Array:

dados = np.array([km, anos])

dados

dados.shape







* + 1. Para fazer as operações vamos utilizar o mesmo esquema de seleção de dados que com listas: Índices.
    2. Utilizando o Array dados, queremos saber qual é a km\_media de cada carro, então vamos utilizar dados[0], para pegar a Array de km rodados e dividir por 2019 (ano do curso) menos dados[1], que é o array de ano de fabricação de cada carro:

km\_media = dados[0] / (2019 - dados[1])

km\_media



* + - 1. Retornando o mesmo resultado que antes.
  1. Para indexar o elemento do Array que queremos pegar, fazemos exatamente igual nas listas.
     1. Quando temos um Array com 2 linhas e várias colunas, fazemos do mesmo modo que uma lista com 2 listas dentro com vários elementos:





* + 1. Existe ainda uma forma mais simples, que é colocar tudo num único [] separado por vírgula:



* 1. Fatiamentos: A[i:j:k]
     1. I: ponto inicial;
     2. J: ponto final/parada
     3. Assim como com listas, o j é exclusivo:



* + 1. K: indicador de passo, ou seja, quantas casa/elementos irá pular até chegar do i até o j:



* + 1. Se quisermos passar apenas pelos pares (nesse caso) podemos fazer o seguinte:



* + - 1. Não passando o i e o j, o python entende que é pra ir do início da Array até o fim dela. Mesmo esquema de fatiamento das listas quando queremos pegar de um ponto i até o final da lista (2:), ou então do ponto inicial até parar num ponto j (:4).
    1. Considerando nossos dados que possuem 2 linhas, se quisermos pegar os elementos 2 ao 3 da primeira linha, podemos juntar o índex com o fatiamento, podendo ser de maneira simplificada com um único [] ou utilizando os 2:
       1. Pegando de maneira simplificada os elementos 2 ao 3 da primeira linha:



* + - 1. Pegando de maneira não simplificada os elementos 2 ao 3 da segunda linha:



* + 1. Se quisermos pegar os elementos 2 e 3 de todas as linhas:



* + 1. Sabendo disso podemos calcular a km\_media de apenas esses 2 elementos da nossa Array dados, por exemplo:



* 1. Podemos fazer verificações bool:



* + 1. E ainda utilizar isso como indexador:



* + 1. Utilizando essas verificações, podemos falar para o python pegar as 2 todas as linhas dos nossos dados e exibir a coluna somente dos veículos que vieram depois dos anos 2000:



* 1. Atributos np:
     1. Shape: retorna quantas linhas e colunas tem.
     2. Ndim: retorna quantas dimensões tem.
     3. Size: retorna número total de elementos.
     4. Dtype: retorna o tipo de dados dos elementos da Array.
     5. T: inverte linhas com colunas. Os anos e km que antes eram linhas e tinhas seus dados como colunas, agora são colunas e tem seus dados como linhas.
        1. Equivalente a transpose().
  2. Métodos np:
     1. Tolist(): converte Array em lista do py.
     2. Reshape(shape[,order]): Reorganiza arrays em novas formas, desde que permita isso.
        1. Temos um Array com 10 elementos em uma linha. Podemos passar a ter um Array com 10 elementos em 5 linhas com 2 colunas:



* + - 1. Podemos fazer com ordens diferentes, ela vem por padrão com o C, mas existem outras:



* + - 1. Um exemplo prático é que temos uma lista com 10 elementos de mesmo tipo, mas queremos dividir em 2 linhas com 5 colunas. Podemos transformar em Array e já fazer um reshape de uma vez:



* + - 1. Voltamos a ter nosso Array “dados” de antes.
      2. Supondo que ao invés de ter as informações da km e do ano de cada carro em 2 linhas separados por colunas, queremos que estejam separados por linhas a informação de cada carro. Podemos simplesmente mudar a ordem do reshape:



* + - 1. Note que virou uma bagunça de anos com km. Para consertar utilizamos o order:



* + 1. Resize((new\_shape), refcheck=True/False): Transforma um array em um novo tamanho.
       1. Por exemplo, nossos dados possuem 2 linhas e 5 colunas. Podemos mandar essa e mudar para 3 linhas e 5 colunas:



* + - 1. Ao fazer isso ele automaticamente preenche a nova linha com 0’s, caso não tenha conteúdo para colocar.
      2. Podemos atribuir novos valores à essa nova linha normalmente com o “=”, passando ela como índice:



* + - 1. Essa função vem por padrão com o refcheck=True, ou seja, ela está sempre verificando se o novo size é igual ou não à referência original (caso a Array que estamos trabalhando seja uma cópia de outra Array). Se o novo size não for igual ao da ref. Original, ele retornará um erro:



* + - 1. Erro esse que pode ser corrigido com o refcheck=False, pois assim ele para de verificar a referência e executa a função:



* 1. Estatísticas:
     1. Column\_stack((Array, Array, Array…)): Cria um array com os dados de todas essas arrays passadas como parâmetro em colunas:

anos = np.loadtxt(fname = "carros-anos.txt", dtype = int)

km = np.loadtxt(fname = "carros-km.txt")

valor = np.loadtxt(fname = "carros-valor.txt")

dataset = np.column\_stack((anos, km, valor))

dataset

dataset.shape





* + 1. Mean(Array): faz a média de todos os elementos em uma Array, mesmo que os dados não façam sentido:



* + - 1. Passando o parâmetro axis = 0, dizemos para o python que queremos que ele faça isso com cada coluna individualmente:



* + - 1. Se passar o valor 1, ele faz a média por linha ao invés de coluna.
      2. Se quisermos fazer a média de somente uma coluna específica, podemos fazer slices:



* + 1. Std(array): desvio padrão. Tudo o que foi dito acima é válido.
    2. Sum(): faz a somatória. Tudo dito acima é válido, a diferença é que o Array pode vir antes ou ser colocado dentro de ():

* 1. O que aprendemos:
     1. A importação de pacotes em Python;
     2. Técnicas para criação de arrays Numpy;
     3. Arrays de mais de uma dimensão;
     4. Comparações de desempenho entre arrays Numpy e listas do Python;
     5. Operações aritméticas com arrays Numpy;
     6. Seleções de itens e fatiamentos em arrays;
     7. Indexação com arrays booleanos;
     8. Atributos e métodos de arrays no pacote Numpy;
     9. A geração de estatísticas descritivas e sumarizações com arrays.

**Curso 3 – Funções, Pacotes e Pandas Básico:**

**Link do colab para usar durante o curso:** <https://colab.research.google.com/drive/1OWQDcsQqYZ9Tkz51Hu_CL2ZTth4QMBcb>

1. **Aula 1 – Apresentação do Curso:**
   1. Importando a biblioteca pandas:

import pandas as pd

* + 1. Geralmente nomeada de pd pela comunidade, facilita a vida. Assim como np para numpy.
    2. Pandas usa arquivos csv para trabalhar com dados. Abre pelo excel.
    3. Para importar os dados para uma variável usamos o read\_csv(‘nome\_do\_arquiv.csv’, sep=’;’):

dataset = pd.read\_csv('db.csv', sep = ';')

* + 1. Precisamos colocar o sep (separador) como ‘;’, caso contrário ocorrerá um erro. O resultado dessa leitura com a separação é essa:



* + 1. Em baixo de tudo ele mostra a quantidade de linhas e de colunas que esse arquivo/dados possuem.
    2. Note que ele exibe apenas 10 linhas, quando na verdade possui 258. Para fazer com que mostre tudo utilizamos o seguinte código:

pd.set\_option('display.max\_rows', 1000)

* + 1. Podendo ser colocado logo abaixo do import. Basicamente seta a opção de exibir o máximo de linhas da biblioteca pandas como 1000, podendo ser alterado pro valor que desejar.
    2. Ao rerodar o código ele exibe todas as linhas.
    3. A indexação no pandas também se inicia em 0.
    4. O mesmo pode acontecer com colunas, para consertar basta alterar o rows por columns:

pd.set\_option('display.max\_columns', 1000)

* 1. Ao usar o dtype para ver qual o tipo de dados que temos, ele mostra a informação de cada uma das nossas colunas:



* + 1. O pandas define str como object.
    2. Diferente das arrays de np, o pd permite que coloquemos várias informações de tipos diferentes, como vemos acima.
  1. Com uma única linha de código podemos fazer com que o pd nos devolva todas as análises estatísticas descritivas de uma vez, apenas utilizando a função describe():

dataset[['Quilometragem', 'Valor']].describe()



* + 1. Na indexação pegamos a coluna de Quilometragem e a de valor da nossa variável dataset, e utilizamos a função describe(). O python nos retorna quantas observações, ou seja, contagem, a média, desvio padrão, o valor mínimo, todos os quartis e o valor máximo de todos os 258 dados dessas 2 colunas.
    2. Abrimos 2 colchetes na indexação pois queríamos pegar mais de 1 informação/coluna, se quiséssemos pegar 3, 4 ou 5 informações, manteríamos os [[]].
  1. Info(): função que exibe descrição básica de todos os dados da variável:



* + 1. Existe uma pequena diferença dos valores do teste descritivo pois ele conta os nulos e missing.
  1. O que aprendemos:
     1. Os ambientes de desenvolvimento para a linguagem Python;
     2. A criar DataFrames com o pacote pandas, a partir de dados externos;
     3. Como obter informações básicas de um DataFrame;
     4. Como obter estatísticas descritivas dos dados de um DataFrame.

1. **Aula 2 – Listas Estatísticas – Tuplas:**
   1. Tuplas são sequências imutáveis que armazenam coleções de itens. Semelhante as listas, mas diferentemente delas, tuplas não podem ser alteradas, ou seja, ter elementos adicionados ou removidos dela.
      1. Como vimos anteriormente, a definição de listas é por [], já a de tuplas é de ().
      2. Built-in functions geralmente retornam tuplas.
      3. Podemos criar tuplas apenas colocando (), ou então diretamente com números separados por ‘,’ como 1,2,3..., utilizando built-in tuple():

* 1. A indexação de tuplas funciona exatamente igual às listas:



* + 1. O slice também funciona da mesma maneira. Também é exclusivo:



* + 1. Também podem existir tuplas dentro de tuplas. Sua indexação e a indexação para selecionar elementos dentro dela é exatamente igual às listas:



* 1. O for funciona da mesma forma que com listas.
  2. Podemos fazer o desempacotamento das tuplas de várias formas, uma delas é criando variáveis e atribuir a tupla à elas:

carro\_1, carro\_2, carro\_3, carro\_4 = nomes\_carros

* + 1. Ao printar um por vez vemos que cada carro\_ possuí um nome dos que estavam na tupla, mas sempre seguindo a sequência em que foram criadas:



* + 1. Podemos usar isso para pegar somente o elemento específico que queremos de dentro da tupla, colocando ‘\_’ nos lugar dos elementos que não nos interessam:

\_, a, \_, b = nomes\_carros



* + 1. Se quisermos pegar somente o segundo, por exemplo, e em uma tupla de 60 itens o resto não me interessa, eu não preciso ficar colocando o ‘\_’ nos próximos 57 itens, basta colocar \*\_ que o python entende que o resto é ‘\_’:

\_, c, \*\_ = nomes\_carros



* 1. Zip(): builtin Function responsável por pegar os elementos de mais de uma lista e juntar eles em uma tupla diferente juntos:

carros = ['Jetta Variant', 'Passat', 'Crossfox', 'DS5']

valores = [88078.64, 106161.94, 72832.16, 124549.07]



* + 1. O python está pegando o primeiro nome da lista carros e o primeiro valor da lista valores e criando uma tupla com os dois e assim subsequentemente até o final.
    2. Precisamos colocar o zip em uma list pois ele não é exibido caso não esteja. Apenas guardado na memória:



* + 1. Podemos juntar o for com o zip e o desempacotamento de tuplas, colocando cada item da tupla em uma variável no for e pedir para ele imprimir todas elas:

for item in zip(carros, valores):

  print(item)



for carro, valor in zip(carros, valores):

  print(carro, valor)



* + 1. Também podemos usar tudo isso mais o if:

for carro, valor in zip(carros, valores):

  if valor > 100000:

    print(carro)



* 1. O que aprendemos:
     1. O que são as tuplas;
     2. Formas de criação de uma tupla;
     3. Técnicas de seleção de itens e fatiamento com tuplas do Python;
     4. Formas de iterar por uma tupla;
     5. A técnica conhecida como desempacotamento de tuplas;
     6. A utilizar a built-in function zip().

1. **Aula 3 – Mapeando Dados – Dicionários:**
   1. Dicionários são coleções que possuem mapeamento, sendo associações de pares entre chaves e valores, onde o primeiro elemento é conhecido como chave e o segundo como valor.
      1. Em listas podemos usar a função index(‘nome\_do\_elemento’) para ele nos retornar o index desse elemento:

carros = ['Jetta Variant', 'Passat', 'Crossfox']

carros.index('Passat')



* + 1. Com duas listas, onde a segunda possuí os valores do carro, precisamos passar o índice de valor, o código acima, para que o python nos devolva o valor do carro que queremos:

valores = [88078.64, 106161.94, 72832.16]

valores[carros.index('Passat')]



* + 1. Para juntar os valores com os carros utilizamos a segunda estrutura:

dados = {'Jetta Variant': 88078.64, 'Passat': 106161.94, 'Crossfox': 72832.16}

dados





* + 1. Note que dicionários são definidos por {}, diferentemente das tuplas () e das listas [].
    2. Caso já tenhamos duas listas determinadas, como nesse caso, não precisamos reescrever todos os valores das duas em um dicionário. Podemos simplesmente usar a função zip() já vista antes para juntar em tuplas, mas dentro da built-in Function dict(). O python automaticamente identifica e cria os dicionários:

dados = dict(zip(carros, valores))

dados



* 1. Para indexar em dicionários podemos simplesmente colocar a chave do valor que desejamos encontrar, dessa forma ficando muito mais fácil de achar os itens, uma vez que não temos que saber qual o número do índice dele.



* + 1. Podemos usar o operador in para saber se alguma chave está ou não em um dicionário, sendo retornado True se sim e False se não:

'Passat' in dados



'Fusca' in dados



* + 1. Len(dict): retorna o número de pares (chave e valor) de um dicionário:

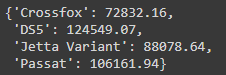
len(dados)



* + 1. Dict[key] = value: Adiciona um novo item a um dicionário:

dados['DS5'] = 124549.07

dados



* + 1. Del dict[key]: Deleta um par de um dicionário:

del dados['Passat']

dados



* + 1. Dentro de um dict podemos passar outro dict com várias chaves e valores para uma chave, assim como em listas:

dados = {

'Passat': {

'ano': 2012,

'km': 50000,

'valor': 75000,

'acessorios': ['Airbag', 'ABS']

},

'Crossfox': {

'ano': 2015,

'km': 35000,

'valor': 25000

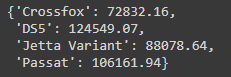
}

}

* + 1. A indexação para acessar os valores de dicts em dicts é igual dict e listas.
  1. Métodos com dicts:
     1. Dict.update({chave:valor}): Adiciona elementos em um dicionário:

dados.update({'Passat': 106161.94})

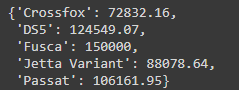
dados



* + - 1. Além disso também podemos atualizar valores e passar mais de uma chave:valor ao mesmo tempo com ele:

dados.update({'Passat': 106161.95, 'Fusca': 150000})

dados



* + - 1. Alteramos valor do Passat e colocamos o fusca.
    1. Dict.copy(): cria uma cópia de um dicionário. Como em listas.
    2. Dict.pop(key): igual ao list. Deleta o último elemento do dict ou o par da chave selecionada:

dadosCopy.pop('Passat')

dadosCopy



* + - 1. Se tentarmos dar pop em algo que já não está na lista, ele gera um erro, mas existe um parâmetro que podemos passar para que ele deixe apenas um aviso:

dadosCopy.pop('Passat', 'Chave não encontrada')



* + - 1. Mas se colocarmos uma chave válida, que ainda não foi removida, ele retorna o valor dela removendo a chave:

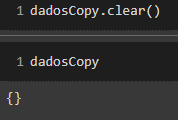
dadosCopy.pop('DS5', 'Chave não encontrada')



* + 1. Dict.clear(): Limpa o dicionário todo, eliminando todas as chaves e valores:

dadosCopy.clear()

dadosCopy



* 1. Iterando dicts:
     1. Dict.keys(): Retorna uma com todas as chaves de um dicionário:

dados.keys()



* + - 1. Podemos usar esse método junto com o for para printar todos os valores:

for key in dados.keys():

  print(dados[key])



* + 1. Dict.values(): Retorna uma lista com todos os valores de um dict:

dados.values()



* + 1. Dict.items(): retorna uma lista contendo uma tupla para cada chave:valor:

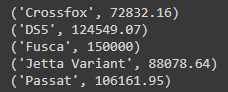
dados.items()



* + - 1. Podemos usar isso junto com for, desempacotamente e ifs juntos, assim como em tuplas:

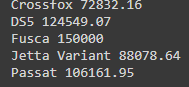
for item in dados.items():

  print(item)



for key, value in dados.items():

  print(key, value)



for key, value in dados.items():

  if value > 100000:

    print(key)



* 1. O que aprendemos:
     1. Estruturas de dados, que representam um tipo de mapeamento, conhecidas como dicionário, na linguagem Python;
     2. Formas de criação de dicionários em Python;
     3. Operações básicas com dicionários (pertinência, acesso, atribuição, etc);
     4. A utilizar os métodos mais importantes de dicionários (update(), pop(), clear(), etc);
     5. Técnicas de iteração por dicionários.

1. **Aula 4 – Organização do Código – Funções e Pacotes:**
   1. Funções são unidades de códigos reutilizáveis que realizam uma tarefa específica, podendo receber entradas e retornar resultados.
      1. Built-in functions: Funções integradas no python, ou seja, que não precisamos fazer nada para utilizar, tais como print(), type(), len(), set(), zip(), tuple()... dentre muitas outras.
      2. List(): Cria uma lista através de dados em uma variável. Consideremos que temos um dict e queremos fazer uma lista somente com os valores. Existe o jeito fácil e o jeito difícil:
         1. Difícil:

valores = []

for valor in dados.values():

  valores.append(valor)

valores



* + - 1. Fácil:

list(dados.values())



* + - 1. Por isso é importante conhecer as built-in functions e consultar documentação. Não tem porque dificultar algo escrevendo linhas e linhas de códigos se podemos simplesmente utilizar uma built-in que faz o que precisamos em uma única linha. O mesmo vale para somar valores.
    1. Sum(iterável, valor(opt)): Utilizado para realizar somas de iteráveis, podendo colocar um valor em seguida sugerindo que ele irá começar com aquele valor e somar com o iterável na resposta final:

sum(dados.values(), 1000000)



* + - 1. Novamente existe o jeito fácil e o difícil de fazer:
         1. Difícil:

soma = 0

for valor in dados.values():

  soma += valor

soma



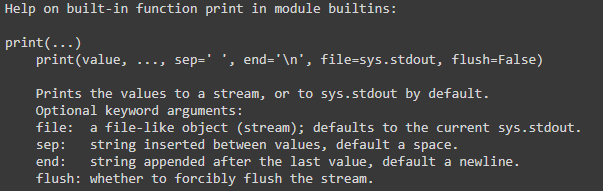
* + - * 1. Fácil:

sum(dados.values())



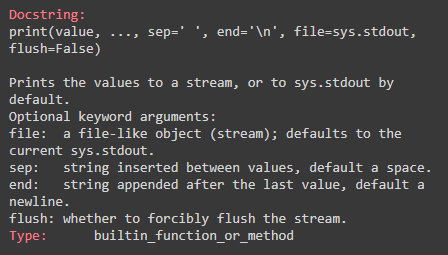
* + 1. Help(function): Exibe um breve resumo de uma função escolhida, dizendo o que faz, e como usar:

help(print)



* + - 1. No caso do colab podemos simplesmente colocar uma ‘?’ na frente da função e ele abre uma janela lateral com a explicação:

print?



* + 1. Nós só criamos uma função caso ela ainda não exista, caso contrário, não tem porquê reinventar a roda. Sempre consultar documentação para ver se uma função já não existe.
  1. Funções sem parâmetros:
     1. São aquelas que não necessita passar nenhuma informação para seu funcionamento, apenas nome\_da\_função() faz com que ela seja executada:

def media():

  valor = (1 + 2 + 3) / 3

  print(valor)

media()



* + - 1. Essa é uma função pouco útil pois é estática, não tem porque ter uma função que retorne sempre a mesma média de valores fixos. Considerando isso é que vemos a necessidade de criar def’s com parâmetros, que recebam inputs para sempre trazer respostas diferentes.
  1. Funções com parâmetros:
     1. São aquelas que precisamos passar alguma informação no () para que ela execute:

def media(number\_1, number\_2, number3):

  valor = (number\_1 + number\_2 + number3) / 3

  print(valor)

media(1,2,3)



media(23, 45, 67)



* + - 1. Agora Podemos passar qualquer valor para nossa função que ela nos retorna a média deles.
    1. Para melhorar ainda mais, podemos passar uma lista como parâmetro, dessa forma não precisamos trabalhar com um número limitado de valores.
       1. Além disso, para fazer a soma, utilizamos a built-in sum() e dividimos pelo len() da nossa lista do parâmetro, desse modo, não importa a quantidade e quais valores forem passados, a média sempre será retornada de maneira correta:

def media(lista):

  valor = sum(lista) / len(lista)

  print(valor)

media(list(range(11)))



* + 1. Como vimos acima, também Podemos utilizar outras built-in dentro dos parâmetros de uma função, desse modo, ao invés de criar manualmente uma lista com 11 caracteres de 0 a 11, simplesmente converti um range(11) em list(), desse modo posso alterar o range e testar novos valores sempre que quiser. Isso poupa a necessidade de criar outra variável com o mesmo código dele (list(range(11))) pra depois passar ela como parâmetro.
    2. Esses valores que estamos recebendo são apenas prints, pois não mandamos nossas def’s retornarem elas, ou seja, não conseguimos atribuir a nenhuma variável e nem fazer nada com esse resultado, apenas visualizar. Para receber um valor que possamos utilizar como retorno, precisamos pedir para nossa def retornar o valor...
  1. Para retornar o valor que uma def te dá, basta utilizar o return:

def media(lista):

  valor = sum(lista) / len(lista)

  return valor

media(list(range(11)))



* + 1. Ele não altera o retorno simples de visualização que temos no colab, mas a diferença do print é que agora podemos atribuir essa função a uma variável e ela receberá esse valor de retorno:

resultado = media(list(range(11)))

resultado



* + 1. Podemos ainda pedir para retornar uma tupla, com os valores do resultado e do tamanho da lista, por exemplo:

def media(lista):

  valor = sum(lista) / len(lista)

  return (valor, len(lista))

media(list(range(11)))



* + 1. Também podemos atribuir esse retorno a variáveis diferentes, assim como fizemos em desempacotamento de tuplas:

resultado, n = media(list(range(11)))

resultado

n





* 1. O que aprendemos:
     1. A utilizar built-in functions no Python;
     2. A consultar a documentação do Python;
     3. Como definir funções com e sem o uso de argumentos;
     4. Como definir funções que retornam valores.

1. **Aula 5 – Biblioteca dos Cientistas – Pandas:**
   1. É uma ferramenta de manipulação de dados de alto nível, construída com base no pacote Numpy. O pacote pandas possui estruturas de dados bastante interessantes para manipulação de dados e por isso é muito utilizado por cientistas de dados.
   2. Series: Arrays unidimensionais rotuladas onde é possível armazenar qualquer tipo de dados. Os rótulos são chamados de index.
      1. Forma básica de criação de uma series:

s = pd.Series(dados, index = index)

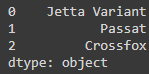
* + 1. O argumento \*dados\* acima pode ser qualquer coisa: Array np, lista, dict, ou uma constante.
  1. DataFrame: Estrutura de dados tabular, bidimensional, com rótulos tanto nas linhas quanto nas colunas. Como a series, os dataframes armazenam qualquer tipo de dados.
     1. Forma básica de criação:

df = pd.DataFrame(dados, index = index, columns = columns)

* + 1. O argumento \*dados\* pode ser um dicionário, uma lista, um array Numpy, uma Series e outro DataFrame.
  1. Criando uma series:
     1. Utilizamos a estrutura acima mencionada com uma lista de carro:

carros = ['Jetta Variant', 'Passat', 'Crossfox']

pd.Series(carros)



* 1. Criando um DataFrame:
     1. Seguimos o exemplo acima utilizando uma lista com dicionários:

dados = [

    {'Nome': 'Jetta Variant', 'Motor': 'Motor 4.0 Turbo', 'Ano': 2003, 'Quilometragem': 44410.0, 'Zero\_km': False, 'Valor': 88078.64},

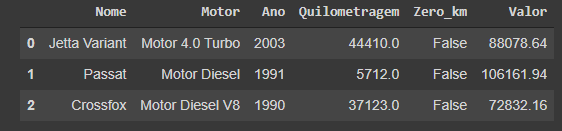
    {'Nome': 'Passat', 'Motor': 'Motor Diesel', 'Ano': 1991, 'Quilometragem': 5712.0, 'Zero\_km': False, 'Valor': 106161.94},

    {'Nome': 'Crossfox', 'Motor': 'Motor Diesel V8', 'Ano': 1990, 'Quilometragem': 37123.0, 'Zero\_km': False, 'Valor': 72832.16}

]

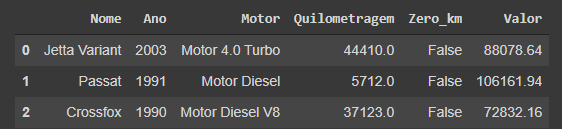
dataset = pd.DataFrame(dados)

dataset



* + 1. Note que a indexação das colunas foi feita a partir das chaves do dicionário, colocando os valores como parte da tabela.
    2. Caso queira alterar a ordem de exibição do dataframe, basta colocar as colunas na ordem que desejar.
    3. A indexação das colunas vem logo após [[]]:

dataset[['Nome',  'Ano', 'Motor', 'Quilometragem', 'Zero\_km', 'Valor']]



* + 1. Podemos criar um dataframe de outra forma, com dicionários contendo listas como valor e as chaves sendo a indexação das colunas:

dados = {

    'Nome': ['Jetta Variant', 'Passat', 'Crossfox'],

    'Motor': ['Motor 4.0 Turbo', 'Motor Diesel', 'Motor Diesel V8'],

    'Ano': [2003, 1991, 1990],

    'Quilometragem': [44410.0, 5712.0, 37123.0],

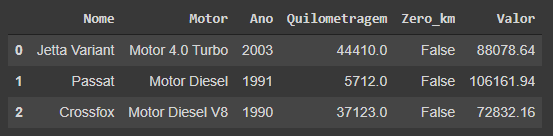
    'Zero\_km': [False, False, False],

    'Valor': [88078.64, 106161.94, 72832.16]

}

dataset = pd.DataFrame(dados)

dataset



* 1. Criando DataFrame de arquivos externos:
     1. Do mesmo jeito que fizemos até agora desde o começo, utilizando o read\_csv():

dataset = pd.read\_csv('db.csv', sep=';')

dataset



* + - 1. O separador que colocamos como ‘;’ só existe e serve nesse caso pois todas as informações desse arquivo csv se encontram separados por ‘;’.
      2. Se fosse um arquivo excel normal, onde a separação foi feita por células e tudo mais, bastava colocar o nome do arquivo, sem a necessidade de colocar o argumento sep.
      3. Podemos ainda passar como parâmetro o index\_col=0. Fazendo isso, definimos que o índex do nosso DataFrame se torne a coluna de índice 0, ou seja, a primeira coluna, ou seja, nesse caso, a coluna de \*nome\*:

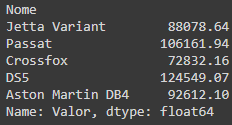
dataset = pd.read\_csv('db.csv', sep=';', index\_col=0)

dataset



* + - 1. Deixando de ser a numérica como na imagem anterior.
  1. Selecionando colunas:
     1. Se quisermos receber um series de uma coluna específica, basta colocar o nome do DataFrame[‘coluna’] (o head() está presente para não mostrar todas as 258 linhas):

dataset['Valor'].head()

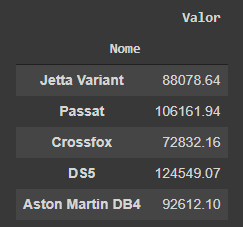


type(dataset['Valor'])



* + - 1. Podemos ver que o tipo que nos retorna são series. DataFrames não são nada mais que um conjunto de series.
    1. Mas se quiser receber um DataFrame dessa seleção, precisamos colocar DataFrame[[‘coluna’]]:

dataset[['Valor']].head()

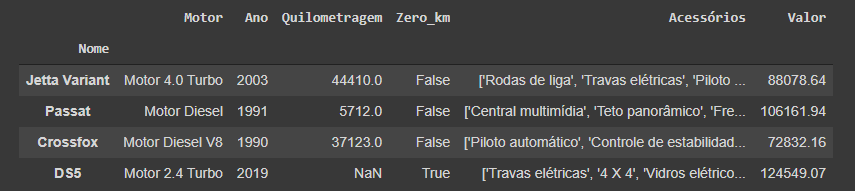


type(dataset[['Valor']])



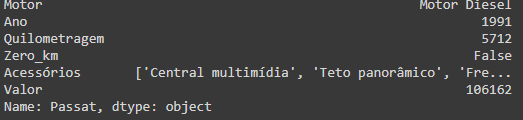
* + - 1. Notamos que o tipo quando colocamos [[]] muda.
  1. Selecionando linhas:
     1. Funciona exatamente igual as listas, tuplas, dicts e tudo mais, incluindo o processo de slices:

dataset[0:4]



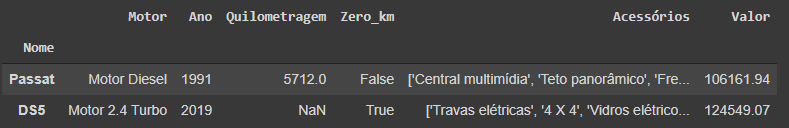
* + 1. DataFrame.loc: seleciona um grupo de linhas e colunas segundo os rótulos ou uma matriz booleana. Um jeito mais elegante de seleção:

dataset.loc['Passat']



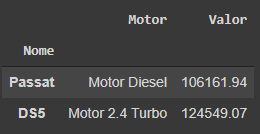
* + - 1. Ele nos retorna uma series com as informações da linha do passat.
      2. Se quiser que ele retorne em formato de DataFrame, coloque [[]]. Também podemos solicitar mais de uma linha nesse formato:

dataset.loc[['Passat', 'DS5']]



* + - 1. Se tentar pedir mais de uma linha no formato series ele dará um erro.
      2. Lembrando que só estamos conseguindo utilizar o nome dos carros como índice pois definimos lá em cima que a primeira coluna seria usada como rótulo, caso contrário estaríamos usando números para determinar cada uma.
      3. Se quisermos selecionar mais de uma linha e mais de uma coluna, mas não todas, precisamos que seja em formato DataFrame[[]], pois em series [] ocorre um erro. Separamos a indexação da seguinte forma [[‘Linha\_1’, ‘Linha\_2’...], [‘Coluna\_1’, ‘Coluna\_2’...]]. Esse é o melhor método para se utilizar:

dataset.loc[['Passat', 'DS5'], ['Motor', 'Valor']]

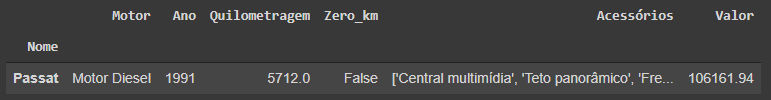


* + - 1. Se quiser imprimir todas as linhas ao invés de apenas algumas, fazemos isso:

dataset.loc[:, ['Motor', 'Valor']]

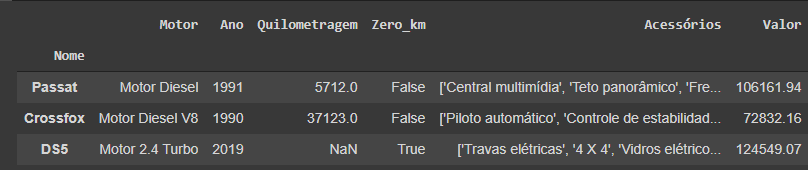
* + - 1. O .loc se baseia nos rótulos.
    1. DataFrame.iloc: Seleciona com base nos índices, ou seja, se baseia na posição das informações:

dataset.iloc[[1]]



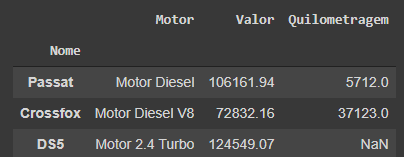
* + - 1. Pegamos a informação do passat, mas diferentemente de antes, utilizamos o índice dele e não o rótulo. Continua assim indefinidamente, igual lists, dicts, tuples....:

dataset.iloc[1:4]



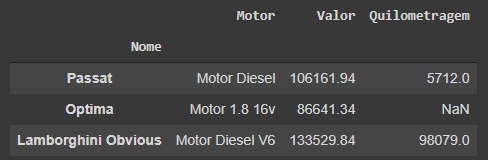
* + - 1. Para selecionar várias linhas e colunas, fazemos como .loc, mas com números ao invés de rótulos:

dataset.iloc[1:4, [0, 5, 2]]



* + - 1. Caso queiramos pegar linhas aleatórias e não em sequência como acima (1:4), fazemos do mesmo jeito que fizemos com as colunas:

dataset.iloc[[1, 42, 22], [0, 5, 2]]



* + - 1. Para pegar todas as linhas, mas colunas específicas fazemos como no .loc:

dataset.iloc[:, [0, 5, 2]]