**Python para Data Science: Introdução à linguagem e Numpy**

Material da aula

02. Ambiente do cientista de dados:

1. Introdução:

Olá! Meu nome é Rodrigo Dias e serei seu instrutor nesse curso de **Python para Data Science**, criado especialmente para quem está começando na carreira de cientista de dados, mas ainda não teve contato com a linguagem Python, tão relevante para essa área.

O treinamento é dividido em duas partes. A primeira delas tratará sobre o básico de Python, como tipos de dados, variáveis, listas, cláusulas if e for, e assim por diante. Ao final, conheceremos um pouco a biblioteca **Numpy**, muito relevantes em ciência de dados. Já no final do próximo treinamento, conheceremos ainda outra biblioteca, a **Pandas**.

Esperamos que o curso seja bastante útil para sua carreira. Bons estudos!

# 02Projeto inicial do treinamento

Nesse curso usaremos o **Google Colab** para escrever o código Python e executar os experimentos. Para tal, você precisa de uma conta no Google e acessar [esse link](https://colab.research.google.com/).

Além disso, baixe [aqui](https://caelum-online-public.s3.amazonaws.com/1410-pythondatascience/01/python-data-science-aula-1-inicio.zip) o ZIP do projeto inicial deste treinamento, necessário para a continuidade do mesmo.

# 03Instalação do ambiente

Antes de começarmos nosso treinamento, vamos conhecer um pouco o notebook Python e o ambiente que utilizaremos para nossos estudos: o [Google Colaboratory](https://colab.research.google.com/). Na atividade [Projeto inicial do treinamento](https://cursos.alura.com.br/course/python-tipos-listas-numpy/task/65463) você encontrará o link para download dos materiais do curso, incluindo a pasta "Numpy". Esta, por sua vez, inclui um arquivo Python\_Data\_Science\_Numpy.ipynb, cuja extensão representa um notebook Python, e outra pasta "data" contendo alguns dados que utilizaremos já a partir do próximo vídeo.

No Google Colab, clicaremos na aba "Upload > Escolher arquivo" e subiremos o arquivo Python\_Data\_Science\_Numpy.ipynb que acabamos de baixar. Esse é um notebook pré-preparado e documentado para te auxiliar nos estudos, entendendo tudo que será feito ao longo do curso. Temos, por exemplo, uma introdução sobre a linguagem Python.

Note que o texto define Python como uma linguagem de programação de alto nível. Isso significa que a sua linguagem se aproxima mais da linguagem humana do que da linguagem das máquinas. Ela dá suporte a múltiplos paradigmas de programação, é um projeto open-source, e vem se tornando uma das linguagens de programação interpretadas mais relevantes. Nos últimos anos ela vem desenvolvendo uma comunidade ativa em processamento científico e análise de dados, e vem se destacando como uma das linguagens mais importantes em data science e machine learning, tanto no meio acadêmico como no mercado.

Como usaremos o Google Colab, não será necessário instalar nada. Note que você precisará de uma conta Google para fazer alguns procedimentos, como o upload de arquivos. Na página <https://www.python.org/> você encontrará algumas referências sobre a linguagem, como a sua documentação e o download do próprio Python caso queira realizar a instalação local. Já em <https://www.anaconda.com/> você poderá baixar a Anaconda Distribution, um conjunto amplo de ferramentas para data science.

Voltando ao Colab, verificaremos a versão do Python que estamos utilizando digitando, na primeira célula de código desse notebook, o comando !python -V ou !python --version. As células funcionam como um terminal, prompt ou shell de programação. Como essa é a nossa primeira execução, o Colab levará algum tempo para se conectar e carregar as informações, retornando, enfim, a versão **Python 3.6.8** (na época de gravação do treinamento). Caso você tenha algum problema com versões futuras da linguagem, poderá forçar o notebook a utilizar esta.

No próximo vídeo começaremos a trabalhar efetivamente com o Python!

# 04Verificando a versão do Python

Vimos que é possível utilizar o **Colab** como *shell*/*prompt*.

Assinale as alternativas que indicam a forma correta de verificar a versão do Python utilizada no **Colab**.

Parte superior do formulário

* Alternativa correta



!python --version

Alternativa correta! Esta é uma outra maneira de verificar a versão do Python.

Parte inferior do formulário

!python -V

Alternativa correta! Lembre-se que, para acessar o prompt ou shell do sistema via Colab, precisamos iniciar a linha de comando com o caractere "!".

# 05O que são arrays Numpy?

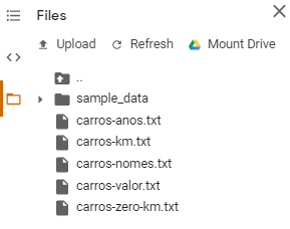
Para iniciarmos nosso curso, abriremos, no Colab, o notebook Python que baixamos anteriormente, mais especificamente na seção **1.3 Trabalhando com arrays Numpy**. O **Numpy** é um pacote para cálculos numéricos, e é muito eficiente realizando essa tarefa - inclusive faremos uma comparação de desempenho entre listas do Python e arrays Numpy. Outra informação interessante sobre o Numpy é que ele serve de base para a construção de pacotes interessantes para Data Science, como o Pandas, que estudaremos na segunda parte desse treinamento.

Quando queremos utilizar uma biblioteca no Python, precisamos importá-la de modo a tornar disponíveis os seus métodos e funções. Não se preocupe se você não entender todos os conceitos e termos citados nesses primeiros vídeos, pois eles serão explicados com calma ao longo do curso.

Na primeira célula da seção **1.3**, escreveremos o código import numpy as np. Na verdade o import numpy já seria suficiente, mas a instrução as np, muito comum em comunidades de data science, atribui um apelido à biblioteca. Assim, poderemos acessá-la por meio do apelido, ou seja, com a estrutura np.método(). Executaremos a célula com o atalho "Shift + Enter".

Dentre os materiais de aula baixados anteriormente, você encontrará cinco arquivos de texto (carros-anos.txt, carros-km.txt, carros-nomes.txt, carros-valor.txt e carros-zero-km.txt) contendo informações sobre veículos anunciados na internet. A ideia é trabalharmos sobre esses dados, tratando-os, obtendo estatísticas e assim por diante.

Vamos supor que desejamos, a partir desse conjunto de dados e das características desses veículos, criar um modelo de previsão do preço dos automóveis. O primeiro passo para isso será obtermos e tratarmos tais dados. No Colab, abriremos o menu na lateral esquerda, clicaremos no ícone que representa uma pasta e em seguida em "Upload". Selecionaremos os arquivos TXT listados acima, clicaremos em "Abrir" e então em "Ok" na mensagem que é exibida na tela. Os arquivos passarão a figurar no menu localizado à esquerda.



Nosso objetivo agora é carregarmos esses dados em um objeto que o Python seja capaz de entender e trabalhar. De modo a entender como é simples realizar cálculos com o Numpy, queremos descobrir a quilometragem média anual dos veículos. Como temos arquivos contendo a quilometragem total e o ano de fabricação de cada carro, será bem simples obtermos essa informação.

Começaremos criando, na segunda célula dessa seção, uma variável km que receberá um array numpy. Isso será feito a partir da chamada da função np.loadtxt(), que carrega o arquivo TXT passado por parâmetro dentro de um array numpy. Nesse caso, passaremos carros-km.txt e executaremos a célula com "Shift + Enter".

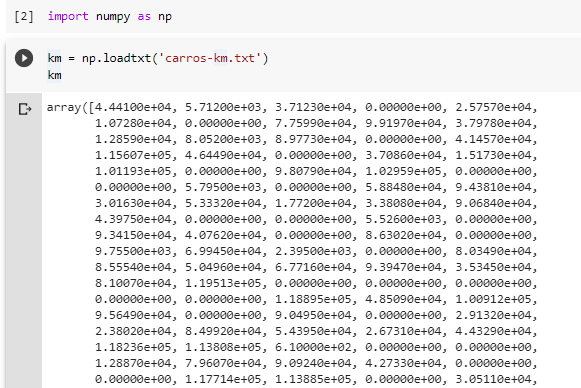
km = np.loadtxt('carros-km.txt')

No Colab, quando desejamos visualizar o conteúdo de uma variável, basta digitarmos o seu nome e executarmos o código.

km = np.loadtxt('carros-km.txt')

km

Isso fará com que o array que acabamos de carregar seja mostrado na tela.



Feito isso, fecharemos a saída no console para mantermos a tela menos poluída. Outra informação que precisamos para conseguirmos calcular a quilometragem média por ano é o ano de fabricação dos veículos. Sendo assim, criaremos a variável anos recebendo também a chamada de np.loadtxt(), dessa vez passando o arquivo carros-anos.txt.

Também podemos definir o tipo de dado que está contido nessa variável. Por exemplo, anos são números inteiros, e não pontos flutuantes quando a quilometragem. Conseguiremos isso passando, como segundo argumento, a instrução dtype = int.

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

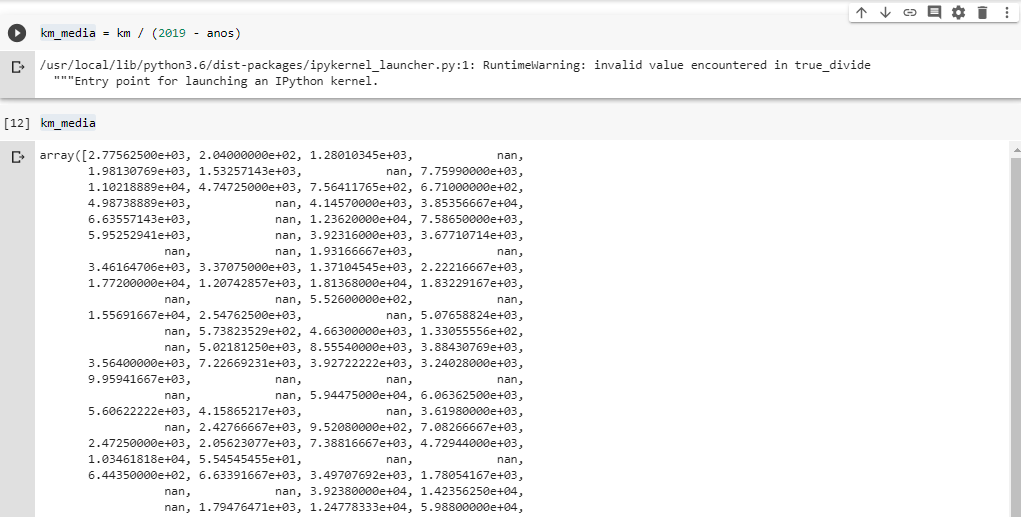
anos

Usando "Shift + Enter", receberemos como retorno a relação de anos contida no array que criamos. Agora queremos calcular a quilometragem anual média desses veículo. Se você lidou com linguagem de programação, sabe que, a partir do array, teríamos que criar um laço for de modo a varrer cada item e operando o cálculo, armazenar o resultado em um novo array e então terminar a operação. Entretanto, isso não é necessário com os arrays numpy, e conseguiremos tal resultado como se estivéssemos fazendo uma operação simples,tal qual "A + B = C".

Na célula seguinte, criaremos a variável km\_media recebendo a divisão de km por 2019 - anos - ou seja, o ano de gravação desse curso menos a variável anos. Ao executarmos, teremos o seguinte retorno:

/usr/local/lib/python3.6/dist-packages/ipykernel\_launcher.py:1: RuntimeWarning: invalid value encountered in true\_divide """Entry point for launching an IPython kernel.

Essa mensagem indica que existem divisões por zero no conjunto, pois existem veículos dentro do array anos que foram produzidos em 2019. Se executarmos a variável km\_media, veremos que, nesses casos, o numpy inclui um nan (not a number) e realiza o cálculo normalmente nos demais pontos em que ele é possível.



Dessa forma, obtivemos o resultado do nosso cálculo com uma conta extremamente simples, sem precisarmos de um laço. Interessante, não? Antes de finalizarmos, vamos executar a função embutida type() do Python passando a variável km\_media como parâmetro. Essa função nos retornará o tipo de dado com que estamos trabalhando - nesse caso, numpy.ndarray, um array do numpy que conheceremos mais profundamente ao final do curso.

Nosso objetivo nesse capítulo era demonstrar quão simples é carregar dados e fazer cálculos sobre eles utilizando a biblioteca Numpy. No próximo começaremos a conversar um pouco mais sobre Python.

# 06Numpy com dados externos

Tivemos o nosso primeiro contato com a biblioteca **Numpy**, do Python. Numpy é a abreviação de Numerical Python e é um dos pacotes mais importantes para processamento numérico em Python.

O pacote Numpy possui um poderoso objeto array multidimensional, que nos permite realizar um conjunto bastante amplo de operações numéricas, sem a necessidade de utilização de laços for.

No último vídeo, utilizamos uma das funções da biblioteca Numpy para ler um conjunto de dados de um arquivo externo (txt) e transformar estas informações em arrays Numpy. Assinale a alternativa que mostra esta função.

loadtxt()

Alternativa correta! A documentação desta função pode ser consultada em: <https://numpy.org/doc/1.16/reference/generated/numpy.loadtxt.html>.

# 07Consolidando o seu conhecimento

Chegou a hora de você seguir todos os passos realizados por mim durante esta aula. Caso já tenha feito, excelente. Se ainda não, é importante que você execute o que foi visto nos vídeos para poder continuar com a próxima aula.

Continue com os seus estudos, e se houver dúvidas, não hesite em recorrer ao nosso fórum!

# 08O que aprendemos?

Nesta aula, aprendemos:

* Os ambientes de desenvolvimento para a linguagem Python
* A carregar dados externos em *arrays* **Numpy**
* A trabalhar de forma básica com *arrays* **Numpy**

02. Características do Python:

# 01Projeto da aula anterior

Caso queira, você pode baixar [aqui](https://caelum-online-public.s3.amazonaws.com/1410-pythondatascience/01/python-data-science-aula-1-completa.zip) o projeto do curso no ponto em que paramos na aula anterior.

# 02Operações matemáticas

Nesse capítulos começaremos a discutir as características básicas do Python, ainda que a ênfase do curso seja sua aplicação em *data science*. Nessa seção conversaremos sobre operações matemáticas, variáveis e tipos de variáveis, como alterar esses tipos, e assim por diante. Dentre as operações matemáticas, além de soma, subtração, divisão e multiplicação, temos também algumas outras que talvez você não conheça.

Para que nossos estudos fiquem documentados, faremos um passo a passo com cada operação na seção

**2.1 Operações matemáticas** do nosso notebook, a começar pela adição, que é bastante simples. Adicionaremos na primeira célula a operação 2 + 2. Note que o interpretador do Python identifica automaticamente o tipo de dado que estamos enviando, além da operação que queremos realizar entre esses dados. Ao executarmos, teremos como retorno o valor 4.

Na subtração, digitaremos a instrução 2 - 2 e executaremos, obtendo o retorno 0. Para indicarmos uma multiplicação utilizamos o caractere asterisco (\*), que testaremos com a operação 2 \* 3, cujo retorno é 6. Já para divisão, usaremos o caractere barra (/), como em 10 / 3. Nesse caso, o retorno não é um valor exato, mas sim 3.333333, algo que chamamos de *número de ponto flutuante*.

Em programação também temos acesso ao operador //, que divide dois números mas retorna somente a parte inteira do seu quociente - no caso de 10 // 3, o retorno será apenas 3. A operação de exponenciação é efetuada por meio do operador \*\*. No exemplo 2 \*\* 3 (dois ao cubo), o resultado será 8.

O operador % é conhecido como *resto da divisão*, pois divide dois números e nos retorna o resto da divisão. Quando a divisão for exata, por exemplo 10 % 2, esse retorno será 0. Se fizermos 10 % 3, o resultado será 1, já que a divisão inteira de 10 por 3 é 3 e o seu resto é 1.

O Python também nos permite trabalhar com expressões matemáticas, respeitando a já conhecida ordem das operações (multiplicação e divisão primeiro, depois soma e subtração). Nesse exemplo usaremos a expressão 5 \* 2 + 3 \* 2. O interpretador do Python identificará que existem múltiplas operações e verificará quais têm preferência, executando-as na ordem correta - ou seja, primeiro 5 \* 2, depois 3 \* 2 e então a soma desses produtos (10 + 6). Como resultado, teremos 16.

Para tornarmos a expressão mais clara ou alterarmos a ordem padrão das operações, basta usarmos os parênteses (). Por exemplo, na expressão 5 \* (2 + 3) \* 2 a primeira operação executada será aquela entre parênteses, 2 + 3. Em seguida, as multiplicações serão executadas da esquerda para a direita (5 \* 5 \* 2), resultando em 50.

Quando acessamos informações com o Python, é possível nos depararmos com a variável \_ (*underscore*), que no modo interativo funciona como a memória de uma calculadora. Por exemplo, se digitarmos \_ e executarmos a célula, receberemos 50, que foi o último valor executado pelo interpretador. No próximo treinamento, no qual falaremos de Pandas, conheceremos outra funcionalidade dessa variável.

Para deixarmos mais evidente e documentado o uso dessa variável, realizaremos a operação 5 \* 2 e, em seguidas, somaremos ao resultado a multiplicação 3 \* 2, ou seja, \_ + 3 \* 2. Com isso, teremos o retorno 16. Se em seguida executarmos \_ / 2, estaremos realizando a operação 16 / 2, cujo resultado é 8.0.

Note que quando fazemos uma divisão com o Python, o retorno sempre é um ponto flutuante, representado por .0 quando essa divisão é exata. Quando existem casas decimais, elas são apresentadas após o ponto.

No próximo vídeo conversaremos mais sobre variáveis.

# 03Desmistificando expressões

Considere as expressões abaixo:

**1)** 10 % 2 + 3 // 10

**2)** 5 \* (2 + 3) / 2

**3)** 2 \*\* 3 \* 4

Selecione a alternativa que apresenta os resultados corretos de cada uma das expressões acima, utilizando a linguagem Python.

**1)** 0

**2)** 12.5

**3)** 32

Alternativa correta! Sempre utilize um notebook para testar os seus códigos.

# 04Entendendo variáveis

Vimos como realizar operações em Python e que elas retornam resultados. Mas e se quisermos pegar esses resultados e utilizá-los em outra parte do código ou programa? Com o passar do tempo, esses programas vão ficando cada vez mais complexos, com diversas operações e resultados, e seria interessante termos uma forma mais inteligente de armazenar essas informações para utilizá-las em outros pontos do processo.

As **variáveis** servem exatamente para isso, e devem ser encaradas como partes da memória do computador nas quais podemos armazenar uma informação temporariamente, rotulando-a com algum nome. Dessa forma, conseguiremos acessar as informações de maneira mais simples enquanto o nosso código estiver rodando.

Na seção **2.2 Variáveis** do nosso notebook Python você encontrará uma breve documentação detalhando as regras para nomear as variáveis. Em Python, as variáveis podem começar com letras de A-Z, tanto maiúsculas quanto minúsculas, ou com o caractere *underscore* (\_).

Altura

\_peso

O resto do nome da variável também pode ser preenchido com caracteres de A-Z, *underscore* ou números.

nome\_da\_variavel

\_valor

dia\_28\_11\_

Note que as variáveis em Python são *case sensitive*, ou seja, fazem diferenciação entre caracteres maiúsculos ou minúsculos.

Nome\_Da\_Variável

nome\_da\_variavel

NOME\_DA\_VARIAVELCOPIAR CÓDIGO

Como toda linguagem de programação, o Python também tem palavras reservadas que não podem ser usadas como variáveis, como and, assert, for, dentre outras. A lista completa pode ser encontrada na documentação.

Os operadores de atribuição são:

* = (igual)
* += (mais igual)
* −= (menos igual)
* ∗= (multiplicação igual)
* /= (divisão igual)
* ∗∗= (exponenciação igual)
* %= (resto da divisão igual)
* //= (divisão inteira igual)

Ao longo desse capítulo veremos alguns exemplos de utilização desses operadores. Começaremos então a trabalhar com as variáveis. Com elas, podemos armazenar o resultado de operações ou mesmo atribuir valores diretamente. No capítulo anterior fizemos um cálculo da quilometragem média dos carros contidos nos nossos dados, e agora escreveremos um pequeno programa que realiza esse cálculo utilizando variáveis.

Primeiro criaremos a variável ano\_atual que receberá o valor 2019, o ano de gravação desse curso.

ano\_atual = 2019

Aqui estamos definindo o nome da variável e utilizando o operador = para atribuir um valor a ela. Com "Shift + Enter", essa instrução será executada e o valor 2019 poderá ser acessado por meio da referência ano\_atual enquanto o programa estiver rodando.

Prosseguiremos criando também as variáveis ano\_fabricacao, que receberá o valor 2003, e km\_total, que receberá 44410.0. Aqui devemos prestar atenção em dois fatores: primeiro, note que não estamos utilizado acentos ou caracteres como ç nos nomes das variáveis, uma boa prática para evitar conflitos no nosso código; e segundo, o valor 44410.0 recebe um .0 ao final, indicando para o interpretador que estamos lidando com um valor de ponto flutuante.

ano\_atual = 2019

ano\_fabricacao = 2003

km\_total = 44410.0

Para conseguirmos a quilometragem média, bastará executarmos a seguinte expressão:

kmmédia=kmtotal/(Anoatual−Anofabricação)

No código, bastará escrevermos essa expressão utilizando as variáveis que acabamos de definir. Note que colocamos a subtração no denominador entre parêntese para indicarmos que ela deverá ser executada primeiro.

km\_total / (ano\_atual - ano\_fabricacao)COPIAR CÓDIGO

Como retorno, teremos 2775.625, a quilometragem média que nosso carro rodou por ano dada a sua idade (16 anos). Podemos colocar essa operação dentro de outra variável, que chamaremos de km\_media).

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

km\_mediaCOPIAR CÓDIGO

A execução desse código nos trará o mesmo retorno, 2775.625. Perceba que, criando variáveis e associando valores a elas, conseguimos utilizá-los facilmente ao longo do nosso programa. Se colocarmos as variáveis e operações em uma única célula, teremos algo que se assemelha mais à ideia que temos de um programa.

ano\_atual = 2019

ano\_fabricacao = 2003

km\_total = 44410.0

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

km\_mediaCOPIAR CÓDIGO

2775.625

Agora se quisermos alterar o ano de fabricação desse veículo, por exemplo, bastará alterarmos o valor sendo atribuído à variável ano\_atual.

ano\_atual = 2019

ano\_fabricacao = 2005

km\_total = 44410.0

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

km\_mediaCOPIAR CÓDIGO

3172.1428571428573

Depois desse teste, mudaremos o valor de ano\_fabricacao de volta para 2003. Para testarmos outros operadores de atribuição, faremos uma previsão bem rudimentar a respeito desse veículo. Imagine que o dono mantenha o mesmo padrão de comportamento e, no próximo ano, rode mais 2775.625 quilômetros. Com esse valor, faremos uma estimativa simples de quanto o veículo terá rodado em 2020.

Isso significa que queremos adicionar o valor da km\_media ao valor de km\_total.

ano\_atual = 2019

ano\_fabricacao = 2003

km\_total = 44410.0

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

km\_total = km\_total + km\_media

km\_totalCOPIAR CÓDIGO

47185.625

Essa é uma forma já conhecida de atribuição, mas existe uma mais simples:

ano\_atual = 2019

ano\_fabricacao = 2003

km\_total = 44410.0

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

km\_total += km\_media

km\_totalCOPIAR CÓDIGO

A linha km\_total += km\_media é exatamente igual à linha km\_total = km\_total + km\_media, e tem o mesmo resultado.

47185.625

Conclusão: "valor = valor + 1" é equivalente a "valor += 1"

Também é possível fazer declarações múltiplas de variáveis em uma única linha.

ano\_atual, ano\_fabricacao, km\_total = 2019, 2003, 44410.0COPIAR CÓDIGO

Ou seja, basta passarmos os nomes das variáveis e os seus respectivos valores na mesma ordem para realizarmos essa atribuição. Isso torna possível simplificarmos o código do nosso programa, reduzindo as linhas de código:

ano\_atual, ano\_fabricacao, km\_total = 2019, 2003, 44410.0

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

km\_mediaCOPIAR CÓDIGO

Trabalhando com *data science* a criação de variáveis é rotineira, portanto é importante fixar esse conteúdo. No próximo vídeo conversaremos sobre tipos de variáveis, um tópico bastante importante!

# 05Nomes de variáveis

Sobre o nome de variáveis, avalie as afirmativas abaixo:

**1)** Nomes de variáveis pode começar com letras (a - z, A - Z) ou o caractere *underscore* (\_)

**2)** O restante do nome de uma variável só pode conter letras e números

**3)** Os nomes de variáveis são *case sensitive*, ou seja, letras maiúsculas e minúsculas são consideradas

Quais estão corretas?

As afirmativas 1 e 3 estão corretas

Alternativa correta! A afirmativa 2 deixa de citar que o caractere "\_" também pode ser utilizado para construir os nomes de variáveis.

# 06Operadores de atribuição

Avalie os itens abaixo:

**1)**

A = 1

B = 2

C = 3COPIAR CÓDIGO

É equivalente a:

A, B, C = 1, 2, 3COPIAR CÓDIGO

**2)**

soma = soma + valorCOPIAR CÓDIGO

É equivalente a:

soma += valorCOPIAR CÓDIGO

**3)**

quadrado = valor \*\* 2COPIAR CÓDIGO

É equivalente a:

quadrado \*\*= 2COPIAR CÓDIGO

Quais estão corretos?

*Selecione uma alternativa*

Os itens 1 e 2 estão corretos

Alternativa correta! No item 1, temos uma declaração múltipla. Veremos mais sobre isso no próximo treinamento, quando conheceremos as tuplas.

# 07Tipos de dados

Já conhecemos as variáveis, e agora conversaremos sobre os **tipos de dados**. São eles que determinam como números e caracteres são armazenados e manipulados dentro de um programa Python. A linguagem Python possui os tipos básicos, que são so numéricos int e float (inteiro e ponto flutuante, respectivamente), além dos bool (*booleanos*, true para verdeiro e false para falso), as strings (sequências de caracteres que representam textos, como números, letras e caracteres especiais) e None, que representa a ausência de valor.

Dependendo do tipo de dados que estamos trabalhando, o programa se comportará de formas diferentes e específicas. Por exemplo, se estamos trabalhando com dados numéricos e utilizamos o operador de soma (+), simplesmente somaremos esses valores. Porém, trabalhando com strings, o operador de soma fará a concatenação dessas strings - ou seja, um comportamento completamente diferente. No futuro conheceremos também outros tipos mais específicos dentro do Python.

Já vinhamos trabalhando com dados numéricos, como quando especificamos a variável ano\_atual.

ano\_atual = 2019COPIAR CÓDIGO

Podemos verificar o tipo dessa variável com a função type(), uma das funções embutidas do Python (algo que chamamos de *built-in function*, assunto que será abordado na continuação deste treinamento).

type(ano\_atual)COPIAR CÓDIGO

int

Como retorno, obtivemos o tipo "int", que representa um numérico inteiro. Outra variável que utilizamos anteriormente foi a km\_total de um veículo.

km\_total = 44410.0COPIAR CÓDIGO

Quando utilizamos no valor um . seguido de algum número, estamos definindo que tal valor possui também uma parte decimal. Sendo assim, o Python armazenará essa informação como outro tipo de dado.

type(km\_total)COPIAR CÓDIGO

float

Para entendermos os valores booleanos, definiremos uma variável zero\_km que receberá o valor True, com a letra inicial maiúscula.

zero\_km = True

type(zero\_km)COPIAR CÓDIGO

bool

Se atribuíssemos False, o retorno da função type() seria o mesmo, já que ambos os valores são do tipo bool. Agora começaremos a trabalhar também com strings. Observe o que acontece quando atribuímos à variável nome um texto rodeado por aspas simples:

nome = 'Jetta Variant'

nomeCOPIAR CÓDIGO

'Jetta Variant'

Ao executarmos, recebemos como retorno o texto atribuído. Isso também funciona com aspas duplas.

nome = "Jetta Variant"

nomeCOPIAR CÓDIGO

"Jetta Variant"

Ou seja, tanto aspas simples quanto aspas duplas são utilizadas para definir valores do tipo string. Mas e se tivermos aspas dentro do nosso texto, como uma citação? Para isso usamos aspas de tipos diferentes. Por exemplo, se precisamos das aspas duplas dentro do texto, definiremos a string usando aspas simples.

nome = 'Jetta "Variant"'

nomeCOPIAR CÓDIGO

'Jetta "Variant"'

O contrário também se aplica:

nome = "Jetta 'Variant'"

nomeCOPIAR CÓDIGO

"Jetta 'Variant'"

Também é possível armazenar strings com mais de uma linha, algo que é feito utilizando três aspas simples ou duplas.

carro = '''

Nome

Idade

Nota

'''COPIAR CÓDIGO

'\n Nome\n Idade\n Nota\n'

Note que o \n nesse retorno significa que uma linha está sendo pulada. Todas essas variáveis que acabamos de criar são do tipo "str", ou seja, strings. O último tipo que queremos apresentar nesse vídeo é o None, a ausência de um valor. Para isso, criaremos uma variável quilometragem que receberá o valor None, com a qual estamos representando um veículo que nunca rodou.

quilometragem = None

quilometragemCOPIAR CÓDIGO

Essa execução não trará nenhum retorno, afinal o valor da variável quilometragem é vazio. Entretanto, ainda é possível verificar o seu tipo:

type(quilometragem)COPIAR CÓDIGO

NoneType

No próximo vídeo entenderemos como e quando é possível converter tipos de valores. Por exemplo, se usarmos o símbolo + para concatenar uma string e um valor numérico, podemos obter um erro. Sendo assim, tentaremos fazer transformações desses tipos de dados para trabalharmos com eles.

# 08Descobrindo o tipo

Os tipos de dados especificam como números e caracteres serão armazenados e manipulados dentro de um programa. Em Python, os tipos de dados básicos são os seguintes:

1. Números
   1. int - Inteiros
   2. float - Ponto flutuante
2. *Booleanos* (bool) - Assume os valores True ou False. Essencial quando começarmos a trabalhar com declarações condicionais
3. *Strings* (str) - Sequência de um ou mais caracteres, que pode incluir letras, números e outros tipos de caracteres. Representa um texto
4. *None* (NoneType) - Representa a ausência de valor

Em nossa última aula, utilizamos um recurso do Python (*Built-in Function*) para identificar o tipo de dado de uma variável. Assinale a alternativa que apresenta esta função.

*Selecione uma alternativa*

type()

Alternativa correta! A *Built-in Function* type() possibilita que identifiquemos o tipo de dado de uma variável.

# 09Conversão de tipos

Vimos no vídeo anterior que nossos programas se comportação de maneiras distintas a depender do tipo de dado com que estivermos lidando. Em ciência de dados, sempre trabalharemos não só com dados de fontes diversas, mas também de tipos diversos - a grande maioria em formato de strings, mas sendo possível identificar, dentro delas, informações distintas.

Sendo assim, será necessário separar tais informações e, provavelmente, convertê-las para formatos que façam mais sentido. Por exemplo, se tivermos dados numéricos em uma string, é interessante convertê-los para tipos numéricos de modo a permitir a realização de operações matemáticas. É justamente sobre a **conversão de tipos de dados** que falaremos nesse vídeo.

Antes de prosseguirmos, vamos definir algumas variáveis que nos auxiliarão na explicação, as duas primeiras representando inteiros e as duas últimas representando strings (algo que podemos conferir com a função type()).

a = 10

b = 20

c = 'Python e '

d = 'legal'COPIAR CÓDIGO

Quando fazemos uma soma com as variáveis a e b, obtemos o seu resultado normalmente:

a + bCOPIAR CÓDIGO

30

Mas o que acontece se fizermos uma operação de soma com as variáveis c e d? Vamos testar:

c + dCOPIAR CÓDIGO

'Python é legal'

Perceba que não foi feita a soma, mas sim a concatenação das duas strings. Segundo essa lógica, poderíamos gerar a string 'Python é 10' usando a concatenação das variáveis c e a, certo?°

c + aCOPIAR CÓDIGO

Entretanto, o resultado dessa execução será um erro, pois o Python não entende como somar ou concatenar informações de tipos diferentes. Sendo assim, para realizarmos essa junção, precisaremos converter uma delas. Claro, devemos levar em conta o que faz mais sentido: não é possível transformarmos "Python" é em um valor numérico, mas é possível transformar 10 em string.

Para convertermos um valor numérico em string, é possível usar a função str() passando como parâmetro a variável ou valor que desejamos converter - por exemplo, a variável a.

str(a)COPIAR CÓDIGO

Se fizermos a chamada de type(str(a)), teremos como retorno um "str", indicando que estamos trabalhando com uma string. Sendo assim, agora poderemos concatenar as duas variáveis:

c + str(a)COPIAR CÓDIGO

'Python é 10'

Além da str(), outras funções básicas de conversão disponíveis no Python são int(), para obtermos valores inteiros, e float(), que transforma valores numéricos em tipo float. Como exemplo, faremos a conversão da variável a para o tipo float.

float(a)COPIAR CÓDIGO

10.0

Como é possível perceber, o valor 10 agora é representado como 10.0, indicando um número de ponto flutuante. Da mesma forma, podemos transformar um ponto flutuante em um tipo inteiro, mas temos que tomar alguns cuidados nesse ponto. Para exemplificar, criaremos uma variável var com o valor aproximado de PI, ou seja, 3.141592. Em seguida, converteremos esse valor para um inteiro.

var = 3.141592

int(var)COPIAR CÓDIGO

Como retorno, teremos:

3

Ou seja, a parte decimal foi ignorada. Não devemos confundir essa operação com um arredondamento. Por exemplo, mesmo que tenhamos um valor 3.99, o valor retornado pela função int() será 3.

Essas são as conversões básicas do Python. Com elas, poderemos transformar strings que contenham dados numéricos em tipos numéricos, ou dados numéricos em strings. No próximo vídeo conheceremos mais alguns pontos essenciais da linguagem Python.

# 10Concatenando strings

Considere as variáveis abaixo:

text = 'A quilometragem média do veículo é '

Km = 100000

Ano\_atual = 2019

Ano\_fabricacao = 1999COPIAR CÓDIGO

Assinale a alternativa que produz o seguinte *output* no console do Colab:

'A quilometragem média do veículo é 5000 km'

text + str( int( Km / (Ano\_atual - Ano\_fabricacao) ) ) + ' km'

Alternativa correta! Note que foram realizadas duas conversões. Na primeira, o resultado da operação (float) é convertido em um inteiro (int) e depois, o dado do tipo inteiro é convertido em uma *string* (str).

# 11Indentação, comentários e formatação

Para encerrarmos essa introdução ao básico de Python, conversaremos um pouco sobre indentação, comentários e formatação de strings. A **indentação**, que é o recuo que utilizamos no código, é uma boa prática em qualquer linguagem de programação, já que facilita a leitura e a manutenção do código, dentre em coisas. Além disso, no Python ela é obrigatória, e sua ausência incorre em comportamentos indesejados ou até mesmo em erros.

Na seção **2.5 Indentação, comentários e formatação de strings**, encontraremos alguns códigos previamente escritos que nos ajudarão a entender esses conceitos,a começar pelo trecho abaixo:

ano\_atual = 2019

ano\_fabricacao = 2019

if (ano\_atual == ano\_fabricacao):

print('Verdadeiro')

else:

print('Falso')COPIAR CÓDIGO

Aqui estamos criando duas variáveis, ano\_atual e ano\_fabricacao, e o valor 2019 está sendo atribuído às duas. Logo após, temos uma cláusula if que verifica se ano\_atual é igual a ano\_fabricacao. Em caso positivo, a mensagem "Verdadeiro" é impressa no console; caso contrário (else), a mensagem "Falso" será exibida. Ao executarmos o código, como esperado, a mensagem "Verdadeiro" é mostrada na tela.

A linha recuada print('Verdadeiro') indica que essa linha de código está dentro da cláusula if acima dela. Mas o que acontece se removermos esse recuo?

ano\_atual = 2019

ano\_fabricacao = 2019

if (ano\_atual == ano\_fabricacao):

print('Verdadeiro')

else:

print('Falso')COPIAR CÓDIGO

Ao executarmos, teremos um *IndentationError*, ou seja, justamente um erro de indentação, pois o Python não consegue interpretar que a linha print('Verdadeiro') está dentro da cláusula if. Ao incluirmos o recuo antes do código, tudo volta a funcionar normalmente.

Outra boa prática em programação é a inclusão de **comentários** no código. O interpretador do Python ignora os comentários, portanto é permitir escrever qualquer coisa que você desejar, como o que aquele código está fazendo, se ele afeta outra parte do programa, se precisa dos dados de uma fonte específica e assim por diante. Para comentários em Python, usamos o símbolo #. No Google Colab, o atalho "Ctrl + ;" permite comentar ou descomentar uma linha de código.

# Isto

# é um

# comentário

ano\_atual = 2019

ano\_atualCOPIAR CÓDIGO

Existe ainda mais uma forma de incluir comentários no Python, que são as aspas simples triplas.

'''Isto é um

comentário'''

ano\_atual = 2019

ano\_atualCOPIAR CÓDIGO

A seguir temos um exemplo do código que rodamos acima, agora com alguns comentários:

# Definindo variáveis

ano\_atual = 2019

ano\_fabricacao = 2019

'''

Estrutura condicional que vamos

aprender na próxima aula

'''

if (ano\_atual == ano\_fabricacao): # Testando se condição é verdadeira

print('Verdadeiro')

else: # Testando se condição é falsa

print('Falso')COPIAR CÓDIGO

A execução desse código ocorrerá normalmente, como se os comentários não existissem. Agora conversaremos um pouco sobre a **formatação de strings**, muitas vezes necessária para melhorar a leitura das saídas do nosso código. Trabalharemos mais especificamente com a funçãostr.format() e o formato f-Strings, mais recente e que vem sido bastante utilizado.

O primeiro passo será incluirmos no código a chamada de print(), uma função embutida do Python que recebe parâmetros e os imprime no console. Fora do Colab, ela é necessária para visualizar os retornos dos códigos que estamos executando. Passaremos para a função o parâmetro 'Olá, {}!' seguido de .format(), um método que pode ser executado a partir de um objeto do tipo string. Esse método receberá "Rodrigo", o nome do nosso instrutor.

print('Olá, {}!'.format('Rodrigo'))COPIAR CÓDIGO

O parâmetro passado para o format() será colocado dentro das chaves ({}) da string anterior. Sendo assim, nosso retorno será:

Olá, Rodrigo!

Podemos ter mais de um valor desse tipo. Como exemplo, vamos supor que estamos trabalhando em um sistema de autenticação e queremos exibir uma mensagem com o nome do usuário e o seu número de acessos. Para isso, passaremos ao print() a string 'Olá, {}! Este é seu acesso de número {}' seguida da chamada de format() contendo os parâmetros Rodrigo e 32. Repare que agora temos duas chaves e elas são posicionais - a primeira delas receberá o primeiro parâmetro, e a segunda o segundo parâmetro.

print('Olá, {}! Este é seu acesso de número {}'.format('Rodrigo', 32))COPIAR CÓDIGO

Olá, Rodrigo! Este é seu acesso de número 32

Também podemos fazer isso nomeando as variáveis. Nesse caso, criaremos, dentro do próprio método format(), a variável nome recebendo 'Rodrigo' e a variável acessos recebendo 32. Dessa forma, não precisaremos mais trabalhar de maneira posicional, podendo passar dentro das chaves as variáveis que acabamos de criar.

print('Olá, {nome}! Este é seu acesso de número {acesso}'.format(nome = 'Rodrigo', acesso = 32))COPIAR CÓDIGO

Olá, Rodrigo! Este é seu acesso de número 32

O formato mais novo, f-Strings, é bem mais simples de entender e aplicar. Por exemplo,

nome = 'Rodrigo'

acessos = 32COPIAR CÓDIGO

Após rodarmos essas variáveis, executaremos o mesmo print() que criamos acima, dessa vez adicionando o caractere f antes da string.

print(f'Olá, {nome}! Este é seu acesso de número {acessos}')COPIAR CÓDIGO

Como retorno, teremos:

Olá, Rodrigo! Este é seu acesso de número 32

Quando adicionamos o f, o Python pega as variáveis armazenadas na memória e passa para a string nos pontos em que determinamos, de um jeito mais elegante e simples que a função format(). No próximo capítulo começaremos a trabalhar com as listas em Python e aprenderemos alguns parâmetros de comparação entre listas.

# 12Características da linguagem

Sobre algumas características da linguagem Python, avalie as alternativas abaixo e marque somente as corretas.

 Uma das formas de se escrever um comentário no nosso código Python é iniciando uma linha com o caractere '#'. Os comentários são ignorados pelo interpretador Python

Alternativa correta! Também é possível escrever blocos de comentários (múltiplas linhas) utilizando aspas triplas antes e depois do bloco.

 Alternativa correta



Em Python, a indentação não é somente uma questão de organização e estilo, mas sim um requisito da linguagem

Alternativa correta! Na linguagem Python, os programas são estruturados por meio de indentação.

# 13Formatação de strings

Assinale as alternativas que apresentam o resultado esperado.

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

Alternativa correta! O método .format() permite a atribuição de nomes para os argumentos.

 nome = 'Rodrigo'

acessos = 32

print(f'Olá, {nome}! Este é seu acesso de número {acessos}')

Alternativa correta! Assim como fizemos em nossa última aula.

 Alternativa correta



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

Alternativa correta! Lembrando que .format() é um método do objeto string e preenche os campos delimitados por {}, com seus argumentos na ordem em que estão posicionados.

# 14Consolidando o seu conhecimento

Chegou a hora de você seguir todos os passos realizados por mim durante esta aula. Caso já tenha feito, excelente. Se ainda não, é importante que você execute o que foi visto nos vídeos para poder continuar com a próxima aula.

# 15O que aprendemos?

Nesta aula, aprendemos:

* A realizar operações matemáticas com Python
* Como criar e atribuir valores a variáveis na linguagem Python
* Os tipos de dados básicos em Python
* A realizar transformações de tipos de dados
* As regras e características básicas da linguagem Python (indentação, comentários e interpolação de *strings*)

03. Trabalhando com listas:

# 01Projeto da aula anterior

Caso queira, você pode baixar [aqui](https://caelum-online-public.s3.amazonaws.com/1410-pythondatascience/02/python-data-science-aula-2-completa.zip) o projeto do curso no ponto em que paramos na aula anterior.

# 02Criando listas

Já aprendemos a criar variáveis e com quais tipos de dados o Python trabalha, como os numéricos int e float (inteiros e pontos flutuantes, que representam números reais), o str (abreviação de string e que lida com textos) e bool (abreviação de booleano, representando true ou false). Nas aulas anteriores, criamos variáveis atribuindo a elas valores únicos, como nos exemplos abaixo:

ano atual = 2019

km = 4000

Em data science, costumamos trabalhar com grandes conjuntos de informações. Nesse curso, por exemplo, temos um dataset com diversos carros, cada um com seu ano de fabricação, quilometragem, valor e assim por diante. Se criássemos uma variável para cada um desses dados, teríamos um programa muito extenso e inviável, sem contar com a possibilidade de erros.

A solução para esse problema é a utilização das **listas** do Python, sequências mutáveis que servem para armazenar coleções de itens normalmente homogêneos, o que faz mais sentido, mas não necessariamente homogêneos. Na seção **3. TRABALHANDO COM LISTAS** do nosso notebook, temos um breve texto mostrando as formas de criação de uma lista, incluindo algumas células já preenchidas para não tomarmos muito tempo digitando esses dados.

Nosso primeiro exemplo é uma lista com diversos acessórios de um veículo. Para criá-la, abrimos e fechamos colchetes ([]) e passamos os itens separados por vírgulas.

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

Acessorios

Ao executarmos, recebemos a lista, também indicada por colchetes, contendo os itens dentro dela.

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

Ao usarmos a função type() para verificar o tipo da variável Acessorios, receberemos como retorno list, que é o tipo de lista do Python. Como citado anteriormente, as listas são preferencialmente homogêneas de modo a termos uma organização, mas podem conter valores de tipos variados, como strings, numéricos ou booleanos. Observe o exemplo abaixo:

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

Carro\_2 = ['Passat', 'Motor Diesel', 1991, 5712.0, False, ['Central multimídia', 'Teto panorâmico', 'Freios ABS'], 106161.94]

Aqui temos uma lista com todas as informações do Carro\_1, como nome, motorização, ano de fabricação, quilometragem e assim por diante. Repare que essa lista contém também outra lista com os acessórios do veículo, demonstrando quão versátil é essa funcionalidade. NO futuro aprenderemos a acessar os itens de uma lista, inclusive aqueles contidos em uma sublista.

Ao executarmos esse código, poderemos visualizar as informações de cada um dos carros:

Carro\_1

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

Carro\_2

['Passat', 'Motor Diesel', 1991, 5712.0, False, ['Central multimídia', 'Teto panorâmico', 'Freios ABS'], 106161.94]

Assim como é possível termos uma lista dentro de outra lista, também podemos criar uma lista contendo variáveis que armazenam outras listas, como no exemplo:

Carros = [Carro\_1, Carro\_2]

Carros

[['Jetta Variant', 'Motor 4.0 Turbo', 2003, 44410.0, False, ['Rodas de liga', 'Travas elétricas', 'Piloto automático'], 88078.64], ['Passat', 'Motor Diesel', 1991, 5712.0, False, ['Central multimídia', 'Teto panorâmico', 'Freios ABS'], 106161.94]]

Veja que logo de cara abrimos dois colchetes, já que a informação de cada carro é também uma lista em si. Ou seja, estamos trabalhando com uma lista de listas (listas estas que também possuem outras listas!). No próximo vídeo conheceremos algumas operações e métodos que nos permitem modificar as listas.

# 03Um pouco de teoria

Sobre as **listas** do Python, avalie as afirmativas:

**1)** Listas são sequências mutáveis, que são utilizadas para armazenar coleções de itens

**2)** É possível armazenar tipos variados de dados em uma mesma lista do Python

**3)** É possível criar listas dentro de outras listas

Quais estão corretas?

Todas as alternativas estão corretas

Alternativa correta! Utilize sempre o nosso notebook de estudo para ajudar na solução dos exercícios.

# 04Operações com listas

Agora que conhecemos as listas do Python, é interessante aprendermos a manipular as informações dentro delas. Como foi dito anteriormente, as listas são mutáveis, ou seja, é possível realizar operações sobre os seus dados, por exemplo adicionando ou retirando itens. Quanto mais recursos um cientista de dados tem para manipular as informações, melhores e mais eficientes serão os seus programas.

Na seção **3.2 Operações com listas** temos, inicialmente, um [link para a documentação do Python](https://docs.python.org/3.6/library/stdtypes.html#common-sequence-operations) onde você encontrará todas operações possíveis com listas. Aqui abordaremos algumas das principais operações utilizadas no dia-a-dia de um cientista de dados.

Primeiro aprenderemos a identificar se um item está dentro de uma lista. Mas qual a utilidade disso? Imagine que temos a seguinte lista de Acessorios de um veículo:

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

Acessorios

Estamos interessados em rodar um modelo de previsão de preços somente para carros que tenham determinada característica, por exemplo as "Rodas de liga". Se identificarmos que a lista contém determinado item, poderemos selecionar aquele veículo. Observe que nessa seção encontramos a instrução **x in A**, onde x representa aquilo que estamos buscando, A representa a lista onde buscaremos e in é o comando Python que fará essa execução. Veja o exemplo:

'Rodas de liga' in Acessorios

Aqui é como se estivéssemos perguntando "o item Rodas de liga está presente na lista Acessorios?". Ao executarmos esse código, o Python nos retorna um booleano True - ou seja, o item 'Rodas de liga' está sim presente em Acessorios. Vamos refazer o teste, agora para o item "4 x 4", que não está contido na lista Acessorios.

'4 x 4' in Acessorios

Dessa vez o retorno será um booleano False, algo que já era esperado. Também podemos fazer essa pergunta "ao contrário" usando a instrução not:]

'Rodas de liga' not in Acessorios

O retorno desse código será False, já que "Rodas de liga" está presente em acessórios. Se fizermos o mesmo com "4 x 4", o retorno será verdadeiro.

'4 x 4' not in Acessorios

True

Prosseguindo, já aprendemos que a funcionalidade do sinal + pode variar de acordo com o tipo de dado com que estamos trabalhando - quando passamos valores numéricos, fazemos uma soma; quando passamos strings, fazemos a concatenação. No caso das listas também é feita a concatenação, juntando os elementos das duas listas.

No nosso exemplo, a lista Acessorios foi divida em duas listas chamadas A e B, cada uma contendo quatro itens.

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

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

Ao executarmos A + B, será feita a concatenação das listas, resultando em uma nova lista de oito itens.

A +

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

Outra informação interessante de descobrirmos é o tamanho de uma lista. No momento estamos lidando com poucos itens, sendo possível contá-los "a olho". Entretanto, em data science acabamos trabalhando com conjuntos de inúmeros dados. A função len(), intrínseca do Python, nos permite descobrir facilmente o tamanho de uma lista:

len(Acessorios)

8

Essas são algumas das operações mais importantes e recorrentes no trabalho com data science, e no próximo vídeo aprenderemos a fazer seleções nas listas.

# 05Trabalhando com listas

Considere a lista abaixo:

carro = [

'Jetta Variant',

'Motor 4.0 Turbo',

2003,

44410.0,

False,

['Rodas de liga', 'Travas elétricas', 'Piloto automático'],

88078.64

]COPIAR CÓDIGO

Considere também as seguintes operações:

**1)** '2003' in carro

**2)** 'Rodas de liga' in carro

**3)** 'False' not in carro

Assinale a alternativa que indica os resultados de cada operação acima:

**1)** False

**2)** False

**3)** True

Alternativa correta! Sempre utilize um notebook para treinar e entender melhor o funcionamento da linguagem.

# 06Seleções em listas

Nosso objetivo agora é aprender a acessar individualmente os itens de uma lista, algo que chamamos de **seleção**. A primeira forma é por meio da sintaxe **A[i]**, onde A representa uma lista e i o índice do elemento a ser acessado. É interessante notarmos que listas são coleções ordenadas de itens nas quais a ordem é determinada por um índice de números inteiros que se inicia no 0.

Se você fechou o notebook do curso anteriormente, será necessário executar novamente os códigos da seção **3**, já que utilizaremos as listas Acessorios e Carros nos nossos exemplos. Começaremos com a lista Acessorios.

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

AcessoriosCOPIAR CÓDIGO

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

Entendemos que para acessarmos um item, precisaremos passar o nome da lista, Acessorios, e a posição de tal item. Além disso, como a ordenação se inicia no 0, o primeiro item da lista será o de índice 0; o segundo, índice 1; e assim por diante. Pensando nisso, se quisermos acessar o item "Rodas de liga", faremos:

Acessorios[0]COPIAR CÓDIGO

Como retorno, teremos:

'Rodas de liga'

Já se quisermos o item "Travas elétricas", usaremos o índice 1:

Acessorios[1]COPIAR CÓDIGO

'Travas elétricas'

E se quisermos acessar o último da lista? Poderíamos, é claro, simplesmente contar os itens, chegando ao índice 7. Mas também existe outro recurso: o índice negativo -1.

Acessorios[-1]COPIAR CÓDIGO

'Sensor de chuva'

Seguindo essa lógica, o índice -2 acessará o penúltimo item da lista; o -3 o antepenúltimo; e assim sucessivamente.

Anteriormente vimos que listas também podem conter outras listas, como é o caso da lista Carros.

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

Carro\_2 = ['Passat', 'Motor Diesel', 1991, 5712.0, False, ['Central multimídia', 'Teto panorâmico', 'Freios ABS'], 106161.94]

Carros = [Carro\_1, Carro\_2]

CarrosCOPIAR CÓDIGO

[['Jetta Variant', 'Motor 4.0 Turbo', 2003, 44410.0, False, ['Rodas de liga', 'Travas elétricas', 'Piloto automático'], 88078.64], ['Passat', 'Motor Diesel', 1991, 5712.0, False, ['Central multimídia', 'Teto panorâmico', 'Freios ABS'], 106161.94]]

O primeiro item dessa lista é também uma lista que por sua vez contém as informações do Carro\_1, da mesma forma que o segundo item representa uma lista com as informações do Carro\_2. Ao acessarmos o primeiro item/índice de Carros, receberemos a primeira lista:

Carros[0]COPIAR CÓDIGO

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

Mas e se quisermos acessar informações dentro desta lista, como o nome do carro? Para isso, acessaremos o primeiro item por meio do seu índice - ou seja, Carros[0] - e em seguida passaremos o índice do item desejado nessa sublista, que também é 0.

Carros[0][0]COPIAR CÓDIGO

'Jetta Variant'

Repare que a nossa sublista Carros\_1 também possui uma lista em seu anterior, o que significa que estamos lidando com três listas em sequências. Agora queremos acessar o item "Travas elétricas", que faz parte dos acessórios contidos nas características desse carro. Para isso, primeiramente teremos que acessar o penúltimo item da lista Carros\_1, algo que conseguiremos com o índice -2.

Carros[0][-2]COPIAR CÓDIGO

['Rodas de liga', 'Travas elétricas', 'Piloto automático']

Como "Travas elétricas" é o segundo item dessa lista, conseguiremos acessá-lo por meio do índice 1.

Carros[0][-2][1]COPIAR CÓDIGO

'Travas elétricas'

É interessante explorar essas possibilidades, testando e fazendo exercícios, para entender bem a lógica das listas em Python e das operações com elas. Outra forma de fazer seleções são os **fatiamentos**, também conhecidos como *slices*, com os quais conseguimos pegar parte de uma lista. Isso é feito com a notação **A[i:j]**, onde A representa uma lista, i o índice inicial e j o índice final do fatiamento.

Isso significa que A[i:j] recorta a lista A do índice i até o j, incluindo o item com índice i, mas excluindo o item com índice j. Para exemplificarmos, queremos acessar os itens que vão de "Piloto automático" até "Sensor de estacionamento" - ou seja, os índices 2 a 5.

Acessorios[2:5]COPIAR CÓDIGO

Entretanto, observe o retorno recebido:

['Piloto automático', 'Bancos de couro', 'Ar condicionado']

Como o fatiamento é não-inclusivo, somente os itens de índice 2 a 4 foram listados. Se quisermos que "Sensor de estacionamento" seja incluído, devemos fazer a seleção até o índice 6.

Acessorios[2:6]COPIAR CÓDIGO

['Piloto automático', 'Bancos de couro', 'Ar condicionado', 'Sensor de estacionamento']

Agora queremos todos os itens a partir do índice 2 até o final. Para isso, basta não passarmos nenhum índice após os dois pontos (:).]

Acessorios[2:]COPIAR CÓDIGO

['Piloto automático', 'Bancos de couro', 'Ar condicionado', 'Sensor de estacionamento', 'Sensor crepuscular', 'Sensor de chuva']

O contrário também funciona. No exemplo, queremos todos os itens anteriores ao índice 5 - lembrando que o último item não é incluído.

['Rodas de liga',

'Travas elétricas',

'Piloto automático',

'Bancos de couro',

'Ar condicionado']COPIAR CÓDIGO

Como dito anteriormente, é interessante testar e explorar possibilidades até entender a lógica desse tipo de seleção. Claro, fazer exercícios ajuda bastante a fixar esses conhecimentos. No próximo vídeo falaremos sobre alguns métodos importantes para o trabalho com listas.

# 07Seleções e fatiamentos

Considere a seguinte lista:

letras = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H']COPIAR CÓDIGO

Considere também as seguintes seleções na lista letras:

**1)** ['A', 'B']

**2)** ['C', 'D', 'E']

**3)** ['F', 'G', 'H']

Assinale a alternativa que apresenta os códigos que têm como resultado as seleções acima.

**1)** letras[:2]

**2)** letras[2:5]

**3)** letras[-3:]

Alternativa correta! Lembrando que é possível navegar pelos itens de uma lista com índices negativos. Esta técnica permite a navegação pela lista no sentido contrário, isto é, do último item (índice -1) até o primeiro.

# 08Listas dentro de listas

Utilize a lista carros abaixo para responder a questão:

carros = [

[

'Jetta Variant',

'Motor 4.0 Turbo',

2003,

False,

['Rodas de liga', 'Travas elétricas', 'Piloto automático']

],

[

'Passat',

'Motor Diesel',

1991,

True,

['Central multimídia', 'Teto panorâmico', 'Freios ABS']

]

]COPIAR CÓDIGO

Considere também os seguintes itens:

**1)** True

**2)** 'Teto panorâmico'

**3)** ['Rodas de liga', 'Travas elétricas', 'Piloto automático']

Assinale a alternativa que mostra o código utilizado para selecionar os itens acima:

*Selecione uma alternativa*

**1)** carros[1][3]

**2)** carros[1][-1][-2]

**3)** carros[0][-1]

Alternativa correta! Em alguns tipos de seleção, a utilização de índices negativos ajuda bastante. Com eles, é possível iniciar a varredura dos itens de uma lista do último (índice -1) até o primeiro.

# 09Métodos de listas

Conheceremos agora alguns métodos que nos permitem fazer modificações nas listas que estamos criando, como a adição de itens, a ordenar da lista, a remoção de itens e a cópia. Na seção **3.4 Métodos de listas** de nosso notebook, deixamos um link para a documentação do Python sobre esse assunto. É importante ter o hábito de consultar a documentação, pois podem existir pontos específicos do seu trabalho que não sejam resolvidos com os métodos apresentados aqui no curso, mas talvez existam outros métodos que possam auxiliar. Também é possível criar novas funções, mas esse é um assunto para o próximo curso.

Aqui trabalharemos com os métodos **sort()**, **append()**, **pop()** e **copy()**. Para facilitarmos os estudos, rodaremos novamente a definição da variável Acessorios.

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

AcessoriosCOPIAR CÓDIGO

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

O método sort() funciona por meio da sintaxe A.sort(), onde A representa uma lista genérica. Note que a nossa lista não está ordenada seguindo alguma lógica, como a ordem alfabética. O sort() nos permite fazer isso, dentre outras possibilidades, como ordenar os valores dos carros. Esse tipo de ordenação nos permite fazer algumas análises preliminares.

Acessorios.sort()

AcessoriosCOPIAR CÓDIGO

O método sort() modifica a lista por meio da qual ele foi chamado, resultando, nesse caso, em uma lista de acessórios em ordem alfabética:

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

O método append() adiciona um elemento passado como parâmetro ao final da lista a partir do qual foi chamado.

Acessorios.append('4 x 4')

AcessoriosCOPIAR CÓDIGO

['Ar condicionado', 'Bancos de couro', 'Piloto automático', 'Rodas de liga', 'Sensor crepuscular', 'Sensor de chuva', 'Sensor de estacionamento', 'Travas elétricas', '4 x 4']

Já o pop() faz exatamente o contrário. Se não passarmos nenhum parâmetro, ele removerá o último elemento da lista, retornando o elemento removido.

Acessorios.pop()COPIAR CÓDIGO

'4 x 4'

Se visualizarmos novamente a lista, perceberemos que o item 4 x 4 não será mais exibido.

AcessoriosCOPIAR CÓDIGO

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

Também podemos remover um elemento específico da lista utilizando a indexação. Por exemplo, se quisermos remover Rodas de liga, teremos que passar como parâmetro o índice desse elemento, que é 3.

Acessorios.pop(3)COPIAR CÓDIGO

'Rodas de liga'

Com isso, nossa lista deixará de conter esse item.

AcessoriosCOPIAR CÓDIGO

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

Imagine que desejamos fazer alterações drásticas em uma lista para realizarmos testes, mas ainda mantendo o conteúdo da original. O método copy() pode ser usado para fazer uma cópia de uma lista. Porém, você deve imaginar que também é possível fazer isso criando uma nova variável e atribuindo a ela o conteúdo da primeira, como no exemplo abaixo:

Acessorios\_2 = Acessorios

Acessorios\_2COPIAR CÓDIGO

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

Feito isso, adicionaremos o elemento "4 x 4" à lista Acessorios\_2 usando o método append().

Acessorios\_2.append('4 x 4')

Acessorios\_2COPIAR CÓDIGO

['Rodas de liga', 'Travas elétricas', 'Piloto automático', 'Bancos de couro', 'Ar condicionado', 'Sensor de estacionamento', 'Sensor crepuscular', 'Sensor de chuva', '4 x 4']

A ideia é que a lista Acessorios mantenha-se a mesma, sem a adição do novo item, certo? Porém, observe o que acontece quando exibimos o seu conteúdo:

AcessoriosCOPIAR CÓDIGO

['Rodas de liga', 'Travas elétricas', 'Piloto automático', 'Bancos de couro', 'Ar condicionado', 'Sensor de estacionamento', 'Sensor crepuscular', 'Sensor de chuva', '4 x 4']

Repare que a lista Acessorios também foi modificada. Isso acontece porque a instrução Acessorios\_2 = Acessorios apenas dá um novo rótulo a uma referência armazenada na memória. Sendo assim, tudo alteração feita a essa referência usando Acessorios\_2 modificará também a lista Acessorios. Há o método copy() não copia somente a referência, criando uma nova área na memória para a lista a partir do qual foi chamado.

Para testarmos, chamaremos Acessorios.pop() de modo a removermos o item "4 x 4". Obviamente, se chamarmos Acessorios\_2, o tem também terá sido removido. Agora criaremos uma nova lista Acessorios\_2 que receberá Acessorios.copy().

Acessorios\_2 = Acessorios.copy()

Acessorios\_2COPIAR CÓDIGO

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

A partir dessa lista, chamaremos novamente o método append() passando como parâmetro o item "4 x 4", o que fará com que esse elemento seja listado.

Acessorios\_2.append('4 x 4')

Acessorios\_2COPIAR CÓDIGO

['Rodas de liga', 'Travas elétricas', 'Piloto automático', 'Bancos de couro', 'Ar condicionado', 'Sensor de estacionamento', 'Sensor crepuscular', 'Sensor de chuva', '4 x 4']

Entretanto, ele não será exibido na lista Acessorios, já que elas são independentes.

AcessoriosCOPIAR CÓDIGO

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

Outra maneira de realizarmos uma cópia atribuindo um novo lugar na memória é por meio da sintaxe **A[:]**, onde A representa uma lista qualquer. Observe o exemplo:

Acessorios\_2 = Acessorios[:]COPIAR CÓDIGO

Essa sintaxe é equivalente ao método copy() - ou seja, ambos têm a mesma funcionalidade. No próximo capítulo conversaremos sobre estruturas de repetição e condicionais.

# 10Elaborações com listas

Considere a seguinte lista do Python e o resultado obtido após um conjunto de manipulações:

**In [1]:**

Acessorios = [

'Rodas de liga',

'Travas elétricas',

'Piloto automático',

'Bancos de couro',

'Ar condicionado'

]COPIAR CÓDIGO

**In [2]:** Qual código?

**Out [2]:**

['Airbag',

'Ar condicionado',

'Bancos de couro',

'Piloto automático',

'Rodas de liga',

'Vidros elétricos']COPIAR CÓDIGO

Assinale a alternativa que apresenta o código executado na célula In [2].

*Selecione uma alternativa*

Acessorios.append('Airbag')

Acessorios.sort()

Acessorios.pop()

Acessorios.append('Vidros elétricos')

Acessorios

Alternativa correta! Quando estiver utilizando um notebook para ajudar na solução de um exercício, procure executar cada comando em células distintas, para observar o resultado de cada operação e assim entender melhor como cada método funciona.

# 11Copiando listas

Considere o código abaixo:

dados = [1, 2, 3, 4]

dados\_copia = dados

dados\_copia.append(5)COPIAR CÓDIGO

A respeito do código acima, assinale as alternativas corretas.

*Selecione 2 alternativas*

Qualquer alteração em dados\_copia será refletida em dados

Alternativa correta! O procedimento acima não cria uma cópia de dados. Apenas atribui um novo rótulo para um mesmo endereço na memória.

Após a execução do código, as listas dados e dados\_copia continuam iguais

Alternativa correta! Neste procedimento, nós apenas acrescentamos um outro rótulo a um mesmo espaço na memória. Qualquer alteração em dados será refletida em dados\_copia e vice-versa.

# 12Consolidando o seu conhecimento

Chegou a hora de você seguir todos os passos realizados por mim durante esta aula. Caso já tenha feito, excelente. Se ainda não, é importante que você execute o que foi visto nos vídeos para poder continuar com a próxima aula.

# 13O que aprendemos?

Nesta aula, aprendemos:

* **Listas**, que são um tipo de sequência mutável que podemos utilizar para armazenar coleções de itens
* Formas de criação de listas em Python
* A realizar operações básicas com listas, como a pertinência, concatenação e verificação de características
* Técnicas de seleção de itens e fatiamento com listas do Python
* A utilizar métodos básicos de listas

# 01Projeto da aula anterior

Caso queira, você pode baixar [aqui](https://caelum-online-public.s3.amazonaws.com/1410-pythondatascience/03/python-data-science-aula-3-completa.zip) o projeto do curso no ponto em que paramos na aula anterior.

# 02Instrução for

Nosso assunto agora são as **estruturas de repetição** e **condicionais** em Python, começando pela instrução **for**. Ela permite que façamos iterações dentro de coleções do Python, que são as listas . Também existem outras formas de criar iteradores para utilizar em estruturas de repetição, como o range que conheceremos mais adiante.

Abaixo temos o formato padrão da estrutura for. Nela temos a instrução for em si seguida de uma variável que representará cada item sendo varrido na coleção, uma instrução in que indica onde queremos iterar e uma coleção - no caso, a nossa lista. Note que a linha termina com dois pontos (:) e que as instruções a serem executadas no laço são indentadas em relação a ela, indicando o seu pertencimento.

for <variável> in <coleção>:

<instruções>COPIAR CÓDIGO

No nosso exemplo continuaremos trabalhando com a lista Acessorios.

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

AcessoriosCOPIAR CÓDIGO

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

Nosso objetivo por enquanto é fazermos uma varredura e acessarmos cada item dessa lista. A ideia também é que seja possível executarmos algum tipo de teste ou condição para esses itens, como o preenchimento de uma lista, mas não trabalharemos nisso especificamente no momento.

Sendo assim, faremos um for e chamaremos a variável a ser utilizada de item. Em seguida, indicaremos que a varredura deverá ocorrer na lista Acessorios usando a palavra reservada in, terminando a linha com dois pontos (:). Pressionando "Enter", o Colab identificará que estamos entrando em um laço for e adicionará a indentação automaticamente. Se estiver usando uma IDE que não faz isso, bastará pressionar "Tab" na linha seguinte.

for item in Acessorios:COPIAR CÓDIGO

Por enquanto simplesmente imprimiremos cada item da lista passando essa variável para a função intrínseca print().

for item in Acessorios:

print(item)COPIAR CÓDIGO

Com isso, acessaremos cada item da lista e os imprimiremos no console do Colab:

Rodas de liga Travas elétricas Piloto automático Bancos de couro Ar condicionado Sensor de estacionamento Sensor crepuscular Sensor de chuva

No futuro falaremos também de loops alinhados e trabalharemos com as instruções if e else. No momento conversaremos sobre uma construção de listas conhecida como *list comprehensions*, além de conhecermos outra *built-in function* do Python, a range(). As referências para a documentação sobre cada um desses assuntos está no nosso notebook. As *list comprehensions* podem parecer confusas no começo, mas com o tempo você pegará o jeito.

O range() é um método que gera um iterador do tamanho passado por parâmetro. Ou seja, se passarmos o valor 10, geraremos uma "lista" iterável com dez elementos.

range(10)COPIAR CÓDIGO

Ao executarmos, receberemos:

range(0, 10)

Isso significa que temos um *range* que se inicia no índice 0 e tem o tamanho 10 - ou seja, dez elementos. Para analisarmos isso, usaremos outra forma de criar listam em Python, a função list(). Ao passarmos o range(10) como parâmetro dessa função, nos será devolvida uma lista com dez elementos.

list(range(10))COPIAR CÓDIGO

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

É basicamente esse tipo de lista que é iterada quando usamos o range() dentro de um for. Em nosso teste, chamaremos cada item encontrado pelo iterador de i e imprimiremos esse valor ao quadrado com print(i \*\* 2).

for i in range(10):

print(i \*\* 2)COPIAR CÓDIGO

Com isso, faremos a varredura, executaremos a operação e imprimiremos o seu resultado:

0 1 4 9 16 25 36 49 64 81

Preste atenção nesses pontos pois, nos exercícios, começaremos a misturar diferentes conceitos já abordados ao longo do curso - algo que continuará a ser feito no próximo treinamento. Ou seja, devemos entender esses conhecimentos como cumulativos e estudá-los com afinco. Se você tiver alguma dificuldade, pode criar um novo notebook Python e realizar alguns testes ou procurar o fórum do curso para obter auxílio.

Agora queremos armazenar as informações que acabamos de criar dentro de uma lista, algo que é possível por meio do método append(). Para isso, iniciaremos uma lista vazia quadrado e em seguida usaremos o for para iterar em um range(10) chamando cada valor encontrado de i. Dentro do iterador, obteremos o quadrado de i com i \*\* 2 e adicionaremos o resultado à lista quadrado usando o método append(). Por fim, sairemos do *loop* e executaremos somente quadrado para obtermos o seu valor.

quadrado = []

for i in range(10):

quadrado.append(i \*\* 2)

quadradoCOPIAR CÓDIGO

Como retorno, teremos uma lista de todos os valores elevados ao quadrado:

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

Voltando ao assunto das *list comprehensions*, sabemos que elas são uma forma concisa de criar listas no Python. Por exemplo, a lista que acabamos de obter pode ser criada usando uma única linha de código:

[i \*\* 2 for i in range(10)]COPIAR CÓDIGO

Aqui estamos abrindo colchetes e, dentro deles, passando a instrução que deve ser executada - nesse caso, i \*\* 2. Em seguida, usaremos o iterador for indicando o nome da variável (i) e onde desejamos iterar (range(10)). O resultado é uma lista idêntica à obtida anteriormente:

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

Com o tempo esse tipo de estrutura se tornará mais natural, portanto não se preocupe se parecer confuso por enquanto. No próximo vídeo conversaremos sobre *loops* alinhados.

# 03Formato padrão

Assinale a alternativa que mostra o formato padrão correto para se utilizar a instrução for na linguagem Python.

for <variável> in <coleção>:

<instruções>

Alternativa correta! Lembrando que no campo <variável>, podemos ter qualquer nome válido em Python, no campo <coleção> podemos ter qualquer objeto iterável (listas, dicionários, tuplas etc) e para iniciar as <instruções>, precisamos antes indentar o código.

# 04List comprehensions

Observe o código abaixo:

**In [1]:**

quadrado = []

for i in range(10):

quadrado.append(i \*\* 2)

quadradoCOPIAR CÓDIGO

**Out [1]:**

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]COPIAR CÓDIGO

Selecione a alternativa que apresenta a forma correta de se criar, de forma concisa (*list comprehensions*), a mesma lista do código acima.

[i \*\* 2 for i in range(10)]

Alternativa correta! Nos próximos vídeos, vamos ver algumas formas mais avançadas de se trabalhas com *list comprehensions*.

# 05Loops aninhados

Nosso assunto agora serão os *loops alinhados*, que são instruções for dentro de outras instruções for, algo que pode ser feito sucessivas vezes. Para esse exemplo criaremos uma lista dados que conterá outras listas dentro dela, que são os acessórios de três veículos diferentes. O objetivo é identificarmos quais acessórios temos disponíveis dentro de todo o *dataset*. Por exemplo, todos os carros possuem "Bancos de couro".

Ao final, queremos uma lista que identifique a existência desse acessório uma única vez. Depois, com as informações de todos os acessórios, poderemos tomar algumas decisões, como agrupar/desagrupar veículos por conta de determinado acessório ou outros tipos de modelagens mais robustas.

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']

]

dadosCOPIAR CÓDIGO

Começaremos criando um for que itera pelos itens de dados chamando cada elemento encontrado (ou seja, cada uma das três sublistas) de lista. Em seguida, imprimiremos cada lista.

for lista in dados:

print(lista)COPIAR CÓDIGO

['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']

Como cada elemento lista é também uma lista, podemos iterar pelos seus itens usando outro for. Dessa vez varreremos a lista chamando cada elemento encontrado de item e imprimiremos esses item no console.

for lista in dados:

for item in lista:

print(item)COPIAR CÓDIGO

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

Assim conseguiremos visualizar todos os acessórios separadamente. Repare que alguns itens inclusive aparecem repetidamente, afinal constam em mais de uma lista. A ideia é pegá-los individualmente, sem repetições.

Para ilustrar, criaremos uma lista vazia Acessorios e usaremos os *loops* alinhados e o método append() para adicionarmos a ela cada item da lista. Ao final, imprimiremos Acessorios.

Acessorios = []

for lista in dados:

for item in lista:

Acessorios.append(item)

print(Acessorios)COPIAR CÓDIGO

['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']

É sobre essa lista que queremos trabalhar. Repare que estamos mostrando algumas funções internas do Python que podem ser úteis no dia-a-dia de um cientista de dados, e trabalharemos ainda mais com elas no próximo curso. Claro que você deve adquirir o hábito de buscar essas funções na documentação, algo que facilitará muito o seu desenvolvimento.

Um exemplo de função interna é a set(), cuja página na documentação também se encontrada linkada no notebook. Ela é muito utilizada para realizar operações em conjuntos matemáticos, como interseção e união. Outro uso é justamente a remoção de duplicatas em uma sequência, o problema que estamos enfrentando agora.

Para isso, simplesmente passaremos a lista Acessorios como parâmetro da função set().

{'4 X 4', 'Ar condicionado', 'Bancos de couro', 'Central multimídia', 'Controle de estabilidade', 'Câmbio automático', 'Câmera de estacionamento', 'Freios ABS', 'Painel digital', 'Piloto automático', 'Rodas de liga', 'Sensor crepuscular', 'Sensor de chuva', 'Sensor de estacionamento', 'Teto panorâmico', 'Travas elétricas', 'Vidros elétricos'}

Entretanto, esse é um retorno do tipo set. Para mostrarmos e trabalharmos esse recurso como uma lista, usaremos a função list().

list(set(Acessorios))COPIAR CÓDIGO

['Freios ABS', 'Teto panorâmico', '4 X 4', 'Ar condicionado', 'Sensor de chuva', 'Sensor de estacionamento', 'Central multimídia', 'Vidros elétricos', 'Sensor crepuscular', 'Travas elétricas', 'Câmbio automático', 'Painel digital', 'Controle de estabilidade', 'Piloto automático', 'Rodas de liga', 'Câmera de estacionamento', 'Bancos de couro']

E se quiser refazer todo esse processo usando *list comprehensions*, reduzindo o código a uma única linha? Para isso, primeiramente passaremos, entre colchetes, o resultado "final" da nossa iteração, que é a variável item. Em seguida, como estamos operando varreduras sequenciais, escreveremos as duas execuções de for uma seguida da outra - ou seja, for lista in dados for item in lista.

Como queremos transformar o retorno em uma lista sem repetições, usaremos os métodos set() e list().

list(set([item for lista in dados for item in lista]))COPIAR CÓDIGO

Como retorno, teremos o mesmo resultado acima:

['Freios ABS', 'Teto panorâmico', '4 X 4', 'Ar condicionado', 'Sensor de chuva', 'Sensor de estacionamento', 'Central multimídia', 'Vidros elétricos', 'Sensor crepuscular', 'Travas elétricas', 'Câmbio automático', 'Painel digital', 'Controle de estabilidade', 'Piloto automático', 'Rodas de liga', 'Câmera de estacionamento', 'Bancos de couro']

Esse tipo de exercício para treinar a construção de *list comprehensions*. No próximo vídeo começaremos a falar da instrução if.

# 06Iterando em listas de listas

Considerando a lista dados abaixo:

dados = [

['A', 'B', 'C'],

['D', 'E', 'F'],

['G', 'H', 'I']

]COPIAR CÓDIGO

Avalie os itens abaixo:

**1)** **In [1]:**

result = []

for lista in dados:

for item in lista:

result.append(item)

resultCOPIAR CÓDIGO

**Out [1]:**

['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I']COPIAR CÓDIGO

**2)** **In [2]:**

result\_2 = []

for lista in dados:

result\_2 += lista

result\_2COPIAR CÓDIGO

**Out [2]:**

['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I']COPIAR CÓDIGO

**3)** **In [3]:**

[item for lista in dados for item in lista]COPIAR CÓDIGO

**Out [3]:**

['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I']COPIAR CÓDIGO

Quais estão corretos?

*Selecione uma alternativa*

Todos os itens estão corretos

Alternativa correta! O item 2 pode parecer uma pegadinha, mas funciona neste caso específico. Lembre-se que o operador de soma (+), quando aplicado em duas listas, retorna a concatenação destas listas.

# 07Instrução if

Nosso assunto agora é a instrução **if**, com a qual conseguimos testar uma condição e executar um código caso ela seja verdadeira, seguindo o formato padrão abaixo:

if <condição>:

<instruções caso a condição seja verdadeira>COPIAR CÓDIGO

Ou seja, se (if) uma <condição> for verdadeira, executaremos as <instruções caso a condição seja verdadeira>. Note que a linha em que usamos o if deve ser terminada com dois pontos (:), e a instrução a ser executada deve ser precedida por uma indentação.

Outro recurso interessante de ser abordado nesse momento são os **operadores de comparação**. Já trabalhamos com operadores de atribuição, por exemplo, quando atribuímos um valor a uma variável (x = 2), rotulando um espaço na memória onde aquele valor é armazenado.

Nesse caso, queremos realizar comparações, como testar de x é igual a 2. Nesse caso, usaríamos o operador de comparação ==.

== igual a

!= diferente de

> maior que

< menor que

>= maior igual a

<= menor igual a COPIAR CÓDIGO

Existem também os **operadores lógicos** **and**, **or** e **not**, com os quais trabalharemos no futuro. Para trabalharmos com o if, deixamos preparada uma lista dados contendo sublistas com as informações de alguns veículos - nome, ano de fabricação e se o veículo é ou não zero km.

# 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]

]

dadosCOPIAR CÓDIGO

Primeiramente usaremos o for para iterarmos por dados chamando cada elemento de lista. Se fizermos um print(list), teremos como retorno a varredura de cada uma das sublistas.

for lista in dados:

print(lista)COPIAR CÓDIGO

['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]

A ideia agora é imprimirmos somente os veículos marcados como True, ou seja, os que são zero km. Para isso, dentro do for, incluiremos um if e abriremos um par de parênteses (()). Podemos acessar o último valor de cada índice por meio do seu índice - como se trata do terceiro valor, o índice é 2. Como exemplo, podemos simplesmente imprimir todos os valores com esse índice em lista.

for lista in dados:

print(lista[2])COPIAR CÓDIGO

False False False True False False True False False False

Com isso, conseguimos acessar o valor que indica se um carro é zero km ou não. Agora incluiremos, no laço for, um if que verifica se a posição list[2] é igual (==) a True. Em caso positivo, imprimiremos a lista.

for lista in dados:

if (lista[2] == True):

print(lista)COPIAR CÓDIGO

Como retorno teremos somente os veículos zero km desse conjunto.

['DS5', 2019, True] ['A5', 2019, True]

Agora temos a possibilidade de criar um laço e, dentro dele, verificar uma condição, executando determinada instrução caso a condição passe. Para tornarmos nosso código um pouco mais elegante, criaremos uma lista vazia zero\_km\_Y. Caso a verificação retorne verdadeiro, adicionaremos o item lista à nova variável. Por fim, imprimiremos o seu conteúdo.

zero\_km\_Y = []

for lista in dados:

if (lista[2] == True):

zero\_km\_Y.append(lista)

zero\_km\_YCOPIAR CÓDIGO

[['DS5', 2019, True], ['A5', 2019, True]]

Como ainda não conhecemos as instruções else e elif, podemos construir outro código para a situação contrária. No caso, teremos uma variável zero\_km\_N e verificaremos de lista[2] retorna False.

zero\_km\_N = []

for lista in dados:

if (lista[2] == False):

zero\_km\_N.append(lista)

zero\_km\_NCOPIAR CÓDIGO

[['Jetta Variant', 2003, False], ['Passat', 1991, False], ['Crossfox', 1990, False], ['Aston Martin DB4', 2006, False], ['Palio Weekend', 2012, False], ['Série 3 Cabrio', 2009, False], ['Dodge Jorney', 2019, False], ['Carens', 2011, False]]

Assim conseguimos criar agrupamentos diferentes, o que nos permitirá testar coisas diferentes - por exemplo, um modelo somente para veículos 0 km ou veículos que não são 0 km.

Também é possível executarmos esse código por meio das *list comprehensions*. Como o resultado final é uma lista, esse será o primeiro parâmetro que passaremos entre colchetes. Em seguida, incluiremos o iterador for, varrendo a lista dados, e então a instrução if.

[lista for lista in dados if lista[2] == True]COPIAR CÓDIGO

[['DS5', 2019, True], ['A5', 2019, True]]

Com isso replicamos o código anterior, agora usando uma única linha. No próximo vídeo usaremos o else para executarmos código também quando uma condição não é verdadeira.

# 08Estruturas condicionais

Sobre **estruturas condicionais**, avalie as alternativas abaixo e assinale a verdadeira.

*Selecione uma alternativa*

**In [2]:**

ano = 2003

if (ano == 2003):

print('OK')

**Out [2]:** OK

Alternativa correta! Sempre lembrando de manter a atenção para os tipos de operadores que estamos utilizando. Na primeira linha, estamos utilizando um operador de atribuição (=) e na segunda, um de comparação (==).

# 09Instruções if-else, if, elif e else

Nesse vídeo falaremos sobre as instruções **if-else**, e **if-elif-else**. Em nossos exemplos usaremos os mesmos dados do vídeo anterior, portanto pode ser possível executá-los novamente.

# 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]COPIAR CÓDIGO

Estávamos construindo duas listas que separavam os dados entre veículos que são ou não são 0 km. Para isso, usando somente a condição if, precisávamos realizar a mesma operação duas vezes, ainda que realizando testes diferentes. Entretanto, com o if/else é possível realizarmos somente uma operação, seguindo o formato padrão abaixo:

if <condição>:

<instruções caso a condição seja verdadeira>

else:

<instruções caso a condição não seja verdadeira>COPIAR CÓDIGO

Para testarmos, começamos com a estrutura de condicional que utilizamos para criar a lista zero\_km\_Y.

zero\_km\_Y = []

for lista in dados:

if (lista[2] == True):

zero\_km\_Y.append(lista)

zero\_km\_YCOPIAR CÓDIGO

Aqui, quando a condição lista[2] é verdadeira (True), executamos o código que adiciona os itens à lista zero\_km\_Y. Para esse tipo de situação, na qual estamos testando uma variável booleana que só assume os valores True ou False, o uso do if/else se encaixa perfeitamente. Primeiro declararemos, na mesma linha de zero\_km\_Y, a variável zero\_km\_N (que também será uma lista do Python.

Manteremos o for varrendo todas as listas em dados, além da condição if preenchendo a lista zero\_km\_Y caso o retorno da verificação seja verdadeiro. Para incluirmos a cláusula contrária, adicionaremos um else seguido de dois pontos (:) no mesmo nível de indentação do if. Na linha seguinte, usaremos o método append() para adicionarmos os demais itens de lista à variável zero\_km\_N.

zero\_km\_N, zero\_km\_Y = [], []

for lista in dados:

if (lista[2] == True):

zero\_km\_Y.append(lista)

else:

zero\_km\_N.append(lista)COPIAR CÓDIGO

Com isso conseguiremos resolver o nosso problema utilizando uma única estrutura.

print(zero\_km\_Y)COPIAR CÓDIGO

[['DS5', 2019, True], ['A5', 2019, True]]

print(zero\_km\_N)COPIAR CÓDIGO

[['Jetta Variant', 2003, False], ['Passat', 1991, False], ['Crossfox', 1990, False], ['Aston Martin DB4', 2006, False], ['Palio Weekend', 2012, False], ['Série 3 Cabrio', 2009, False], ['Dodge Jorney', 2019, False], ['Carens', 2011, False], ['Jetta Variant', 2003, False], ['Passat', 1991, False], ['Crossfox', 1990, False], ['Aston Martin DB4', 2006, False], ['Palio Weekend', 2012, False], ['Série 3 Cabrio', 2009, False], ['Dodge Jorney', 2019, False], ['Carens', 2011, False], ['Jetta Variant', 2003, False], ['Passat', 1991, False], ['Crossfox', 1990, False], ['Aston Martin DB4', 2006, False], ['Palio Weekend', 2012, False], ['Série 3 Cabrio', 2009, False], ['Dodge Jorney', 2019, False], ['Carens', 2011, False], ['Jetta Variant', 2003, False], ['Passat', 1991, False], ['Crossfox', 1990, False], ['Aston Martin DB4', 2006, False], ['Palio Weekend', 2012, False], ['Série 3 Cabrio', 2009, False], ['Dodge Jorney', 2019, False], ['Carens', 2011, False]]

Agora partiremos para o **elif**, que basicamente consiste na junção de um else e um if. A ideia é que testamos uma primeira condição com o if e executamos determinada instrução caso ela seja verdadeira. Caso contrário, se tivermos outra condição para testar, usamos o elif. Se a condição for verdadeira, executamos um trecho de código; do contrário, podemos partir para outra condição com o elif, e assim sucessivamente. Por fim, encerramos a sequência com um else que executará outro código caso nenhuma das condições anteriores sejam verdadeiras.

if <condição 1>:

<instruções caso a condição 1 seja verdadeira>

elif <condição 2>:

<instruções caso a condição 2 seja verdadeira>

elif <condição 3>:

<instruções caso a condição 3 seja verdadeira>

.

.

.

else:

<instruções caso as condições anteriores não sejam verdadeiras>COPIAR CÓDIGO

Na verdade o else não é estritamente necessário, sendo possível terminar as instruções com o próprio elif. Em nosso próximo exemplo, queremos separar os veículos do conjunto dados de acordo com o ano em que foram fabricados, agrupando-os em três grupos diferentes. Essa informação pode ser encontrada no índice 1 das nossas listas.

Antes de fazermos isso, vamos explorar um pouco os códigos preparados no notebook a respeito de dois operadores lógicos: **and** e **or**.

Quando testamos duas condições e usamos o operador and, se uma delas é verdadeira e a outra também é, o resultado é verdadeiro; se uma é verdadeira e a outra é falsa, o resultado e´falso; e se ambas forem falsas, o resultado também é falso.

print('AND')

print(f'(True and True) o resultado é: {True and True}')

print(f'(True and False) o resultado é: {True and False}')

print(f'(False and True) o resultado é: {False and True}')

print(f'(False and False) o resultado é: {False and False}')COPIAR CÓDIGO

AND (True and True) o resultado é: True (True and False) o resultado é: False (False and True) o resultado é: False (False and False) o resultado é: False

No caso do or, essa lógica é um pouco diferente. Se ambas as condições forem verdadeiras, o resultado é verdadeiro; se uma for verdadeira e outra não, o resultado também é verdadeiro; e a única situação em que o resultado é falso é quando as duas condições forem falsas.

print('OR')

print(f'(True or True) o resultado é: {True or True}')

print(f'(True or False) o resultado é: {True or False}')

print(f'(False or True) o resultado é: {False or True}')

print(f'(False or False) o resultado é: {False or False}')COPIAR CÓDIGO

OR (True or True) o resultado é: True (True or False) o resultado é: True (False or True) o resultado é: True (False or False) o resultado é: False

Para o próximo teste, criaremos as listas vazias A, B e C. Em seguida, usaremos o for para iterar por dados chamando cada elemento encontrado de lista. Incluiremos então a primeira verificação: se lista[1] for menor ou igual a 2000 (ou seja, se o veículo foi fabricado antes do ano 2000 ou neste ano), adicionaremos o elemento em A usando o método append().

A, B, C = [], [], []

for lista in dados:

if lista[1] <= 2000:

A.append(lista)COPIAR CÓDIGO

Com o elif, criaremos uma segunda condição, dessa vez verificando se lista[1] é maior que 2000 (ou seja, se o veículo foi fabricado após o ano 2000) e (and) menor ou igual a 2010 (fabricação antes de 2010). Em caso positivo, adicionaremos o elemento na lista B usando o append().

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)COPIAR CÓDIGO

Por fim, adicionaremos uma última condição com o else: caso nenhuma das condições anteriores sejam preenchidas, adicionaremos o carro à lista C.

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)COPIAR CÓDIGO

Após executarmos esse código, verificaremos o conteúdo das listas:

ACOPIAR CÓDIGO

[['Passat', 1991, False], ['Crossfox', 1990, False]]

BCOPIAR CÓDIGO

[['Jetta Variant', 2003, False], ['Aston Martin DB4', 2006, False], ['Série 3 Cabrio', 2009, False]]

CCOPIAR CÓDIGO

[['DS5', 2019, True], ['Palio Weekend', 2012, False], ['A5', 2019, True], ['Dodge Jorney', 2019, False], ['Carens', 2011, False]]

Perceba que A só possui veículos cuja fabricação é anterior ou ano 2000; B compreende veículos com fabricação entre 2001 e 2010; e C contém somente veículos fabricados após 2010.

O Python nos permite utilizar uma notação bastante conhecida em matemática, na qual colocamos o elemento comparado entre dois valores. No nosso exemplo, lista[1] deve estar entre 2000 e 2010:

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)COPIAR CÓDIGO

Esse código funcionará da mesma forma que o anterior, Lembre-se que essa notação não é válida na maioria das linguagens de programação, mas é bem aceita no Python e pode ser encontrada em pacotes voltados à estatística ou matemática.

Encerramos então o conteúdo básico de Python apresentado neste treinamento. Mas fique tranquilo(a), pois iremos nos aprofundar ainda mais na linguagem durante a continuação, "Python para Data Science: Funções, Pacotes e Pandas básico". Na próxima seção, que encerra a primeira parte do curso, conheceremos a biblioteca **Numpy**.

# 10Operadores lógicos

Em nosso último vídeo, vimos como utilizar os operadores lógicos and e or. Visto isso, verifique as duas tabelas abaixo:

**1)**

| **X** | **Y** | **X and Y** |
| --- | --- | --- |
| False | False | False |
| False | True | False |
| True | False | False |
| True | True | True |

**2)**

| **X** | **Y** | **X or Y** |
| --- | --- | --- |
| False | False | False |
| False | True | True |
| True | False | True |
| True | True | False |

Selecione a alternativa que apresenta a avaliação correta.

*Selecione uma alternativa*

Apenas a tabela 1 está correta

Alternativa correta! A expressão X or Y será False somente quando X e Y forem False, ou seja, na tabela 2, a última linha da tabela está incorreta.

# 11Consolidando o seu conhecimento

Chegou a hora de você seguir todos os passos realizados por mim durante esta aula. Caso já tenha feito, excelente. Se ainda não, é importante que você execute o que foi visto nos vídeos para poder continuar com a próxima aula.

# 12O que aprendemos?

Nesta aula, aprendemos:

* Como utilizar estruturas de repetição e condicionais na linguagem Python
* A construção de laços for
* A iteração em listas do Python
* *Loops* aninhados em listas de listas
* Cláusulas if, elif e else
* Operadores lógicos e de comparação
* *List comprehensions*

**05. Conhecendo o Numpy:**

# 01Projeto da aula anterior

Caso queira, você pode baixar [aqui](https://caelum-online-public.s3.amazonaws.com/1410-pythondatascience/04/python-data-science-aula-4-completa.zip) o projeto do curso no ponto em que paramos na aula anterior.

# 02Arrays NumPy

Nesta última seção do curso conversaremos sobre o pacote **Numpy**, uma abreviação de Numerical Python. Este é um dos pacotes mais importantes para processamento numérico no Python, além de ser a base para a maioria dos pacotes científicos que utilizam dados numéricos. No notebook deixamos inclusive um texto explicativo que introduz a esse pacote.

Alguns pontos importantes são a disponibilidade de um poderoso objeto array multidimensional, as funções matemáticas sofisticadas para operações com arrays sem a necessidade de utilização de laços for, e os recursos de algebra linear e geração de números aleatórios. O Numpy também serve como um contêiner eficiente para transporte de dados multidimensionais entre algoritmos e pacotes com os quais trabalhamos, além de servir de base para construção de outros pacotes.

Existem diversos pacotes do Python que não são distribuídos na versão padrão (default) do seu interpretador, como o próprio Numpy, o Pandas (assunto do próximo curso), o Scikit-learn, o Matplotlib e o Seaborn. O Google Colab disponibiliza esses pacotes previamente, eliminando a necessidade de sua instalação. Caso você esteja usando outra distribuição, a instalação pode ser necessária.

Para utilizarmos os métodos e funções de interesse em cada pacote, é necessário importá-los. No caso do Numpy, basta usarmos a palavra-chave import (reservada do Python) seguida do nome dessa biblioteca.

import numpyCOPIAR CÓDIGO

Na [documentação do Numpy](https://numpy.org/doc/1.16/reference/generated/numpy.arange.html), encontramos algumas explicações sobre o arange(), que basicamente funciona como a função range() - ou seja, ela cria uma lista (nesse caso um array numpy) que se inicia no 0 e tem o tamanho indicado por parâmetro.

Para chamarmos essa função, usaremos numpy.arange(), passando como valor o número 10. Como retorno, teremos um array de tamanho 10:

numpy.arange(10)COPIAR CÓDIGO

array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

Existe também outra forma de fazer essa importação, que é atribuindo um apelido à biblioteca, algo que fizemos no início desse curso.

import numpy as npCOPIAR CÓDIGO

Esse tipo de importação é muito comum em livros e textos sobre data science, e se estende também a outras bibliotecas. A Pandas, por exemplo, costuma ser apelidada de pd. Com a atribuição do apelido, poderemos chamar a função arange() da seguinte forma:

np.arange(10)COPIAR CÓDIGO

array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

Se conhecemos bem a biblioteca e não queremos encher a memória do computador com informações que não serão utilizadas, podemos importar somente as funções que iremos utilizar. Para isso, basta usarmos a instrução from seguida do nome da biblioteca, a palavra-chave import e a função desejada - nesse caso, arange().

from numpy import arangeCOPIAR CÓDIGO

Com isso teremos acesso à função arange\*() diretamente.

arange(10)COPIAR CÓDIGO

array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

Agora começaremos a criar propriamente os famosos **arrays Numpy**. Uma das maneiras de fazermos isso é a partir de uma lista. Para isso, criaremos uma variável km que receberá a chamada de np.array(). Passaremos como parâmetro dessa função uma lista contendo números aleatórios.

km = np.array([1000, 2300, 4987, 1500])

kmCOPIAR CÓDIGO

array([1000, 2300, 4987, 1500])

Também é possível criar arrays numpy a partir de dados externos, algo que já fizemos no início desse curso. Para isso, faremos novamente o upload dos arquivos disponibilizados (carros-aos.txt, carros-km.txt e assim por diante). No código, criaremos novamente a variável km, agora recebendo a chamada de np.loadtxt().

O primeiro parâmetro que passaremos a essa função é o fname =, que receve o nome do arquivo cujo conteúdo queremos armazenar - nesse caso, carros-km.txt. Outro parâmetro possível é o dtype, que define o tipo de dado armazenado. O padrão na criação do array é float, e usaremos a instrução dtype = int para forçarmos esse tipo.

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

kmCOPIAR CÓDIGO

A execução nos devolverá um retorno como esse:

array([ 44410, 5712, 37123, 0, 25757, 10728, 0, 77599, 99197, 37978, 12859, 8052, 89773, 0, 41457, 115607, 46449, ...])

Se quisermos, podemos verificar o tipo de dado armazenado com km.dtype.

dtype('int64')

Existem também arrays com mais de uma dimensão. Nesse caso, trabalharemos apenas com arrays de duas dimensões, deixando as outras possibilidades para cursos mais avançados. Esse tipo de array funciona como uma matriz ou um dataset. De modo a facilitarmos nosso aprendizado, usaremos a seguinte lista pré-preparada.

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']

]

dadosCOPIAR CÓDIGO

Criaremos então uma variável Acessorios à qual atribuiremos a chamada de np.array(). Esta, por sua vez, receberá por parâmetro a lista dados.

Acessorios = np.array(dados)

AcessoriosCOPIAR CÓDIGO

Com isso, teremos um array numpy contendo toda a nossa lista.

array([['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']],

dtype='<U24')COPIAR CÓDIGO

É possível verificarmos a dimensão de cada um dos arrays que acabamos de criar utilizando a instrução .shape. Começaremos com km.shape:

km.shapeCOPIAR CÓDIGO

(258, )

Isso significa que o nosso array possui 258 "linhas" e uma única dimensão. Mas observe o que acontece quando fazemos o mesmo para Acessorios:

Acessorios.shapeCOPIAR CÓDIGO

(3, 8)

Esse retorno é uma dupla indicando que o array possui **3 linhas** e **8 colunas**.

Agora que conhecemos um pouco do funcionamento desses arrays, ainda falta entendermos o motivo para utilizá-los ao invés das listas. Primeiramente, os arrays têm um desempenho muito melhor que o das listas quando fazemos operações matemáticas com eles. Para demonstrarmos isso, criaremos uma variável np\_array à qual atribuiremos a chamada de np.arange() recebendo 1000000 (um milhão) como parâmetro.

np\_array = np.arange(1000000)COPIAR CÓDIGO

Em seguida, criaremos também uma variável py\_list que usaremos para armazenar uma lista (list()) também contendo números até um milhão (range(1000000)).

py\_list = list(range(1000000))COPIAR CÓDIGO

A ideia agora é fazermos um for que multiplicará cada elemento do array np\_array por 2, e repetirá esse processo 100 vezes. Para isso, usaremos a sintaxe for \_ in range(100): np\_array \*= 2, onde o underscore (\_) indica que queremos ignorar aquela variável. Antes dessa operação, adicionaremos a instrução %time, com a qual conseguiremos verificar o tempo de execução desse código. Note que, nesse caso, conseguiremos multiplicar os valores por 2 diretamente, sem a necessidade de outro for.

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

Repetiremos essa construção, dessa vez passando a lista do Python (py\_list) na operação de multiplicação. Entretanto, dessa vez será necessária a inclusão de outro for, que resumiremos com o auxilio das list comprehensions.

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

Agora compararemos os tempos de execução de cada um deles, começando pelo array numpy:

CPU times: user 65.2 ms, sys: 0 ns, total: 65.2 ms Wall time: 73.7 ms

Note que foram 73 millisegundos para terminar o processamento da operação. Já com as listas do Python:

CPU times: user 8.17 s, sys: 1.79 s, total: 9.96 s Wall time: 9.98 s

Aqui temos um tempo muito superior, de aproximadamente 10 segundos. Ou seja, a diferença de desempenho é enorme e representa uma vantagem de se utilizar o Numpy. No próximo vídeo conversaremos sobre operações aritméticas com arrays Numpy.

# 03Sobre os arrays

Sobre os *arrays* **Numpy**, avalie as alternativas abaixo e selecione as corretas.

*Selecione 2 alternativas*

 *Arrays* **Numpy** somente podem ter uma dimensão

Alternativa errada! *Arrays* **Numpy** podem ter várias dimensões. Em nossa última aula, criamos um *array* com duas dimensões.

 Alternativa correta



Não é possível ter mais de um tipo de dado em um único *array* **Numpy**

Alternativa correta! Diferente das listas do Python, que podem conter tipos variados em uma mesma sequência, *arrays* **Numpy** suportam somente um tipo de dado por vez.

 Alternativa correta



É possível criar *arrays* **Numpy** bidimensionais, com tipos diferentes de dados em cada dimensão

Alternativa errada! Isto não é possível. Em um mesmo *array* só pode haver dados de um mesmo tipo.

 Alternativa correta



É possível criar *arrays* **Numpy** a partir de dados externos, no formato TXT, utilizando a função loadtxt() do **Numpy**

Alternativa correta! Como vimos em nossas aulas, basta fornecer o endereço do arquivo TXT que contém as informações, para carregar um *array* **Numpy** com os dados.

# 04Operações com arrays

Já conhecemos os arrays Numpy, aprendemos a criá-los de diferentes maneiras e testamos a sua performance em comparação às listas do Python. Agora aprenderemos a realizar operações aritméticas com esses arrays.

O conjunto de dados do nosso projeto possui informações de veículos, como ano de fabricação, quilometragem e valor. No início do curso fizemos um cálculo para descobrir a quilometragem média dos carros do conjunto, e para isso utilizamos a idade total e a quilometragem dos veículos.

Na seção **5.2 Operações aritméticas com arrays Numpy** do notebook deixamos duas células previamente preenchidas, uma delas com listas e outra com arrays Numpy, mas ambas com os mesmos valores. Começaremos trabalhando com as listas.

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

anos = [2003, 1991, 1990, 2019, 2006]

Essas listas são ordenadas, ou seja, o primeiro valor de cada uma corresponde ao primeiro carro; o segundo, ao segundo carro; e assim sucessivamente. Queremos fazer um cálculo simples para descobrir a idade do veículo, algo que conseguiremos subtraindo o ano atual (2019 na gravação deste curso) pelo ano de fabricação de cada automóvel.

Criaremos entõa uma variável idade que receberá a subtração de 2019 pela lista anos.

idade = 2019 - anos

A execução desse código nos retornará um erro, já que não é possível fazer a subtração de um inteiro por uma lista. Entretanto, isso é possível com os arrays Numpy.

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

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

Ao subtrairmos 2019 pelo array anos, o retorno será outro array Numpy que poderá ser exibido na tela, este contendo as idades de cada um dos veículos.

idade = 2019 - anos

idade

array([16, 28, 29, 0, 13])

Também podemos realizar operações usando dois ou mais arrays. Por exemplo, se quisermos a quilometragem média (km\_media) podemos simplesmente dividir o array km pela variável idade que acabamos de criar.

km\_media = km / idadeCOPIAR CÓDIGO

Nesse caso teremos uma saída que indica um erro, já que nosso conjunto contém um valor 0, incorrendo em uma divisão por zero. Entretanto, o próprio Numpy preenche esse valor com nan (*not a number*) e continua a operação.

array([2775.625 , 204. , 1280.10344828, nan, 1981.30769231])

Perceba que não tivemos que utilizar nenhum tipo de laço ou criar outros arrays para realizarmos essas contas, que afinal são bem simples. Nesse momento estamos trabalhando apenas com uma dezena de dados, mas imagine se tivéssemos milhões? Levaria muito tempo para percorrermos todas as informações e realizarmos as operações usando laços, e isso custaria muito desempenho.

No vídeo anterior conhecemos os arrays de duas dimensões, e é possível realizar operações com eles. Para criarmos um array desse tipo, basta chamarmos a função np.array() passando como argumento uma lista contendo outros dois arrays. Nesse caso, criaremos o novo array dados usando como base os arrays km e anos que criamos anteriormente.

dados = np.array([km, anos])

dadosCOPIAR CÓDIGO

Como retorno, temos um array Numpy com duas linhas e 5 colunas (2, 5), que representando uma matriz.

array([[44410., 5712., 37123., 0., 25757.], [ 2003., 1991., 1990., 2019., 2006.]])

Isso é ilustrado pelo desenho abaixo, lembrando que a indexação se inicia no 0:



Agora queremos acessar esses dados de modo a calcularmos novamente a quilometragem média. Da mesma forma que fizemos com as listas, o índice 0 indica que estamos acessando o primeiro item - ou seja, o primeiro array/linha contendo 5 dados. Já com dados[1], acessaremos o segundo índice/linha. Ou seja, para obtermos km\_media, podemos simplesmente dividir dados[0], que representa a quilometragem, por 2019 - dados[1], que é a idade dos veículos.

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

km\_media

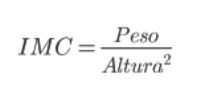
Receberemos um aviso indicando que existe uma divisão por zero, mas a lista com os resultados será criada normalmente:

array([2775.625 , 204. , 1280.10344828, nan, 1981.30769231])

Aprenderemos mais sobre as seleções em arrays no próximo vídeo.

# 05Contas com arrays NumPy

O **IMC** (índice de massa corporal) é um indicador utilizado para detectar situações de obesidade ou desnutrição. Para obter o IMC, basta calcular a razão entre o peso de um indivíduo (kg) e sua altura elevada ao quadrado, conforme mostra a fórmula abaixo:



Considerando o enunciado acima, verifique cada alternativa e assinale as que calculam corretamente o IMC, não aprensentando erro(s) durante o seu cálculo.

*Selecione 2 alternativas*

Parte superior do formulário

 peso = np.array([106.0, 68.5])

altura = np.array([1.9, 1.53, 1.75])

IMC = peso / altura \*\* 2

IMC

Alternativa errada! Para conseguirmos realizar os cálculos, os dois *arrays* devem ter a mesma dimensão.

 Alternativa correta



peso = np.array([106.0, 68.5, 75.0])

altura = np.array([1.9, 1.53, 1.75])

IMC = peso / (altura \* altura)

IMC

Alternativa correta! Esta é uma forma correta para obter o valor do IMC de cada indivíduo, sem a necessidade de laços for.

 Alternativa correta



peso = np.array([106.0, 68.5, 75.0])

altura = np.array([1.9, 1.53, 1.75])

IMC = peso / altura \*\* 2

IMC

Alternativa correta! Esta é uma forma correta de cálculo do IMC, utilizando *arrays* **Numpy**.

 Alternativa correta



peso = [106.0, 68.5, 75.0]

altura = [1.9, 1.53, 1.75]

IMC = peso / altura \*\* 2

IMC

Alternativa errada! Note que este código utiliza listas do Python para realizar este tipo de operação. Com este tipo de dado, não conseguimos realizar operações aritméticas de forma direta.

# 06Seleções com arrays

Anteriormente aprendemos a fazer seleções em listas do Python, inclusive quando elas possuem sublistas. Se você absorveu esse conteúdo, as seleções com arrays Numpy não serão um problema, já que são basicamente a mesma coisa. Como referência, continuaremos utilizando o desenho do array dados que criamos no vídeo anterior.



Esse array pode ser encarado com uma matriz/tabela em que temos duas linhas e cinco colunas, lembrando que a indexação se inicia no 0.



Começaremos nossos estudos criando uma variável contador que receberá a chamada de np.arange(10), gerando um array Numpy de tamanho 10.

contador = np.arange(10)

contadorCOPIAR CÓDIGO

array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

Sabemos que contador[0] nos retornará o primeiro elemento desse array, que é o 0. Agora imagine que desejamos criar um programa que nos retorna o sexto item de um array Numpy. Nesse caso, criaríamos uma variável item recebendo o valor 6, e outra, chamada index, que receberia item - 1 - ou seja, para acessarmos o índice correto basta subtrairmos em 1 o item desejado. Se fizermos contador[index], conseguiremos retornar corretamente o valor desse sexto item.

item = 6

index = item - 1

contador[index]COPIAR CÓDIGO

5

Dessa forma, se desejarmos acessar outros itens desse array, bastará alterarmos o valor de item. Se quisermos acessar o último elemento do array, também podemos utilizar o índice negativo -1

contador[-1]COPIAR CÓDIGO

9

Em relação ao nosso array dados, como vimos no vídeo anterior, a primeira linha pode ser acessada com dados[0], e a segunda com dados[1]. Mas e se quisermos selecionar parte desses arrays menores, como o valor 1990? Para isso, precisaremos passar os valores da linha e da coluna. Nesse caso, queremos a linha com índice 1 e a coluna com índice 2.

Existem duas formas de fazer isso. A primeira dela é usando dois pares de colchetes,seguindo a sintaxe nomedoarray[linha][coluna]. Outra opção é usando somente um par de colchetes e separando os dois índices com uma vírgula, ou seja, nomedoarray[linha,coluna]. Em ambos os casos o nosso retorno será o mesmo, como verificamos nos exemplos abaixo:

dados[1][2]COPIAR CÓDIGO

1990

dados[1,2]COPIAR CÓDIGO

1990

Agora falaremos sobre os **fatiamentos**. No notebook do curso deixamos uma breve explicação sobre a sintaxe dos fatiamentos, incluindo um conteúdo novo, que é o "passo".

A sintaxe para realizar fatiamento em um array Numpy é i:j:k onde i é o índice inicial, j é o índice de parada, e k é o indicador de passo ( k≠0 )

Lembrando que o i é incluído e o j não é incluído. Ou seja, se quisermos fatiar o segundo, terceiro e quarto item do nosso array, precisaremos iniciar no índice 1 e varrer o array até o índice 4, que não será incluído.

No contador que acabamos de criar, queremos selecionar os valores 1, 2 e 3. Como esse array se inicia com 0, o primeiro valor estará exatamente no índice 1. Para recuperarmos até o valor 3, usaremos o índice 4.

contador[1:4]COPIAR CÓDIGO

array([1, 2, 3])

Mas e o "passo", como funciona? Para testarmos, usaremos contador[1:8:2] - ou seja, vamos varrer o array contador do índice 1 até o 8, que não será incluído, contando dois passos a cada elemento encontrado.

contador[1:8:2]COPIAR CÓDIGO

array([1, 3, 5, 7])

O tamanho do passo determinado será o tamanho dos "pulos" dados na varredura do array. No próximo exemplo, queremos pegar todos os números pares do conjunto contador. Para isso, não passaremos nenhum índice inicial ou final, somente o passo 2 - ou seja, contador[::2].

contador[::2]COPIAR CÓDIGO

Como retorno, teremos somente os números pares:

array([0, 2, 4, 6, 8])

Se quisermos repetir o processo para os ímpares, basta começarmos no 1:

contador[1::2]COPIAR CÓDIGO

array([1, 3, 5, 7, 9])

Agora voltaremos ao nosso conjunto dados:

array([[44410., 5712., 37123., 0., 25757.], [ 2003., 1991., 1990., 2019., 2006.]])

Queremos, por exemplo, calcular a quilometragem média somente para o segundo, o terceiro e o quarto veículo desse conjunto. Para isso, primeiro recuperaremos todas as linhas do conjunto, ou seja, dados[:,]. Em seguida, queremos apenas os índices 1, 2 e 3. Conseguiremos isso com a instrução 1:4 - ou seja, os índices de 1 a 3, já que o quatro não é incluído.

dados[:, 1:4]COPIAR CÓDIGO

array([[ 5712., 37123., 0.], [ 1991., 1990., 2019.]])

Repare que é criado um array somente com as informações dos veículos que separamos. Se quisermos somente os veículos 1 e 2, podemos usar dados[:, 1:3].

array([[ 5712., 37123.], [ 1991., 1990.]])

É possível calcularmos a quilometragem média desses dois veículos, bastando seguirmos a fórmula abaixo:

dados[:, 1:3][0] / (2019 - dados[:, 1:3][1])COPIAR CÓDIGO

Parece confuso? Pois vamos explicar! Aqui estamos fatiamento o conjunto dados de modo a selecionarmos somente os veículos com os índices 1 e 2 (dados[:, 1:3]). Em seguida, pegamos somente a primeira linha - ou seja, o índice 0 desse conjunto. De posse desses valores, passamos a dividi-los por 2019 subtraído de dados[:, 1:3][1], a segunda linha do conjunto também fatiada somente com os veículos de índices 1 e 2. Com isso, estamos dividindo a quilometragem total dos veículos pelo tempo desde a sua fabricação, o que nos resulta na quilometragem média:

array([ 204. , 1280.10344828])

Outra forma de fazermos seleções é por meio dos arrays booleanos. A ideia é informarmos quais posições queremos que sejam mostradas com True, e quais não queremos com false. Nesse exemplo voltaremos a usar nosso contador:

contador = np.arange(10)

contadorCOPIAR CÓDIGO

Imagine que queremos saber quais são os elementos maiores do que 5 nesse array. Para isso, usaremos contador > 5. O retorno é um array Numpy de booleanos:

array([False, False, False, False, False, False, True, True, True, True])

Repare que todos os números menores ou iguais a 5 recebem False, mas 6 em diante recebem True. Isso significa que a expressão contador > 5 nos permitirá selecionar somente os elementos maiores do que 5 no array, da seguinte forma:

contador[contador >5]COPIAR CÓDIGO

array([6, 7, 8, 9])

Nesse retorno, só recebemos os valores que obtiverem True no "validador" que criamos. Tanto isso é verdade que podemos, inclusive, passar diretamente os valores False e True conforme desejarmos para o array. Nesse caso, manteremos como True todos os valores maiores do que 5.

contador[[False, False, False, False, False, False, True, True, True,

True]]COPIAR CÓDIGO

array([6, 7, 8, 9])

Vamos supor que queremos calcular a quilometragem média dos veículos fabricados apos o ano 2000. Para selecionarmos esses carros, podemos usar a expressão dados[1] > 2000, já que os anos se encontraram no índice 1 desse array.

dados[1] > 2000COPIAR CÓDIGO

array([ True, False, False, True, True])

Para selecionarmos somente esses dados, basta passarmos todo o conjunto como instrução das "linhas" e essa verificação como instrução das "colunas".

dados[:, dados[1] > 2000]COPIAR CÓDIGO

O retorno serão somente as colunas cujos veículos foram fabricados após o ano 2000.

ray([[44410., 0., 25757.], [ 2003., 2019., 2006.]])

Agora que fizemos a seleção, você pode realizar o cálculo como exercício desse tópico! No próximo vídeo aprenderemos um pouco sobre atributos e métodos de arrays Numpy.

# 07Seleções e fatiamentos

Considere o seguinte *array*:

dados = np.array(

[

['Roberto', 'casado', 'masculino'],

['Sheila', 'solteiro', 'feminino'],

['Bruno', 'solteiro', 'masculino'],

['Rita', 'casado', 'feminino']

]

)COPIAR CÓDIGO

Marque a alternativa que mostra o código que permite a seleção, sem utilização de operadores lógicos (and e or), das informações de **nome** e **estado civil**, somente das pessoas do sexo **masculino**.

*Selecione uma alternativa*

 dados[0::2, :2]

Alternativa correta! O código acima produz exatamente o que o enunciado pede:

array([['Roberto', 'casado'],

['Bruno', 'solteiro']], dtype='<U9')COPIAR CÓDIGO

 Alternativa correta



dados[:2, :2]

Alternativa errada! O código acima seleciona as informações das duas primeiras linhas e colunas:

array([['Roberto', 'casado'],

['Sheila', 'solteiro']], dtype='<U9')COPIAR CÓDIGO

 Alternativa correta



dados[0, 2][0, 1]

Alternativa errada! Este tipo de seleção não é possível com *arrays* **Numpy**.

 Alternativa correta



dados[:2, 0::2]

Alternativa errada! Tente imaginar que, dentro do colchete, o primeiro índice lida com as linhas de um array (matriz) e o segundo trata das colunas. O código acima deve retornar o seguinte resultado:

array([['Roberto', 'masculino'],

['Sheila', 'feminino']], dtype='<U9')

# 08Atributos e métodos do NumPy

Nosso assunto agora são os atributos e métodos com arrays Numpy. Já conhecemos alguns métodos de listas que facilitam bastante o trabalho com elas, e isso também se aplica à essa situação, já que os métodos que conheceremos facilitarão bastante o trabalho com os arrays. Discutiremos um pouco os atributos e métodos mais comuns, mas saiba que existem vários outros que podem ser encontrados na documentação do Numpy.

Nos exemplos usaremos o array dados que criamos anteriormente, um array Numpy bidimensional que possui na primeira linha as informações de quilometragem e na segunda linha os anos de fabricação dos veículos.

dados = np.array([[44410., 5712., 37123., 0., 25757.], [ 2003., 1991., 1990., 2019., 2006.]])

dadosCOPIAR CÓDIGO

array([[44410., 5712., 37123., 0., 25757.], [ 2003., 1991., 1990., 2019., 2006.]])

Os atributos podem ser acessados usando a sintaxe nomedoarray.nomedoatributo. O primeiro que conheceremos é o **shape**. Esse atributo retorna uma tupla com as dimensões do array. Nesse caso, temos a quantidade de linhas na primeira posição e a de colunas na segunda.

dados.shapeCOPIAR CÓDIGO

(2, 5)

Outro atributo é o **ndim**, que retorna o número de dimensões do array. No nosso caso, como temos linhas e colunas, temos duas dimensões.

dados.ndimCOPIAR CÓDIGO

2

O atributo **size** nos mostra o número de elementos de um array. Se o array dados possui duas linhas e cinco colunas, o número de elementos é a multiplicação desses dois valores.

dados.sizeCOPIAR CÓDIGO

10

Para sabermos o tipo de dado dos elementos armazenados no array, algo que já fizemos anteriormente com o Numpy, usamos o atributo **dtype**.

dados.dtypeCOPIAR CÓDIGO

dtype('float64')

Lembrando que se estivéssemos trabalhando com listas, poderíamos ter tipos variados dentro delas, algo que não é possível com arrays Numpy.

O atributo **T** nos retorna o array transposto, ou seja, ele transpõe uma matriz, convertendo linhas em colunas e vice-versa.

dados.TCOPIAR CÓDIGO

array([[44410., 2003.], [ 5712., 1991.], [37123., 1990.], [ 0., 2019.], [25757., 2006.]])

Note que antes tínhamos a quilometragem e os anos como linhas, e agora ambos são colunas. Esse atributo é equivalente à chamada do método transpose():

dados.transpose()COPIAR CÓDIGO

array([[44410., 2003.], [ 5712., 1991.], [37123., 1990.], [ 0., 2019.], [25757., 2006.]])

Dentre os diversos métodos disponíveis, começaremos conhecendo o **tolist()**, que transforma um array Numpy em uma lista do Python. Isso pode ser útil quando precisamos executar algum método ou procedimento que não é possível com os arrays Numpy.

dados.tolist()COPIAR CÓDIGO

[[44410.0, 5712.0, 37123.0, 0.0, 25757.0], [2003.0, 1991.0, 1990.0, 2019.0, 2006.0]]

O método **reshape()** retorna um array contendo os mesmos dados, mas com uma nova forma. Para o primeiro exemplo usaremos uma variável contador que receberá a chamada de np.arange(10), gerando um array Numpy de tamanho 10.

contador = np.arange(10)

contadorCOPIAR CÓDIGO

array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

Agora queremos transformá-lo em um novo array Numpy, mas com 5 linhas e 2 colunas. Note que estamos fazendo uma configuração na qual cabem todos os elementos. Para isso, chamaremos contador.reshape() passando como parâmetro o formato desejado - nesse caso, a tupla (5, 2), representando 5 linhas e 2 colunas.

contador.reshape((5,2))COPIAR CÓDIGO

array([[0, 1], [2, 3], [4, 5], [6, 7], [8, 9]])

O método reshape(), por padrão, usa a indexação da linguagem C. As configurações de indexação são feitas por meio do parâmetro order:

contador.reshape((5, 2), order='C')COPIAR CÓDIGO

array([[0, 1], [2, 3], [4, 5], [6, 7], [8, 9]])

Existe outra forma de indexação, que é a da linguagem Fortran, e podemos configurá-la usando a instrução order='F'.

contador.reshape((5, 2), order='F')COPIAR CÓDIGO

array([[0, 5], [1, 6], [2, 7], [3, 8], [4, 9]])

Isso pode ser útil dependendo do tipo de dado que você está trabalhando, e veremos um exemplo mais prático com os nossos dados futuramente. Note que na indexação da linguagem C os elementos vão sendo colocados nas linhas na mesma ordem em que estavam no array original; Já na indexação da linguagem Fortran, os elementos foram todos distribuídos na primeira posição de cada linha até as 5 serem completadas, e e o restante na segunda posição.

Para exemplificarmos, usaremos duas linhas:

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

anos = [2003, 1991, 1990, 2019, 2006]COPIAR CÓDIGO

Criaremos então uma lista info\_carros que será simplesmente a concatenação das duas anteriores.

info\_carros = km + anos

info\_carrosCOPIAR CÓDIGO

[44410, 5712, 37123, 0, 25757, 2003, 1991, 1990, 2019, 2006]

Imagine que já encontramos esses dados "bagunçados", de maneira que fica difícil trabalharmos com eles. O reshape() nos ajuda a solucionar esse tipo de problema. Como os dados são do mesmo tipo, podemos transformá-los em um array Numpy

np.array(info\_carros)COPIAR CÓDIGO

array([44410, 5712, 37123, 0, 25757, 2003, 1991, 1990, 2019, 2006])

Nós automaticamente podemos fazer o reshape() desses dados, por exemplo informando que desejamos um array com 2 linhas e 5 colunas.

np.array(info\_carros).reshape((2, 5))COPIAR CÓDIGO

array([[44410, 5712, 37123, 0, 25757], [ 2003, 1991, 1990, 2019, 2006]])

Repare que nosso resultado é justamente o array dados que tínhamos anteriormente, dessa vez feito a partir de uma lista "bagunçada". Vamos supor que seria mais prático se as informações de cada carro estivessem em linhas individuais - ou seja, cada linha representando a quilometragem e o ano de determinado carro. Para isso, usaríamos a tupla (5, 2), representando 5 linhas e 2 colunas. Mas repare o que acontece quando usamos a indexação padrão da linguagem C:

np.array(info\_carros).reshape((5, 2))COPIAR CÓDIGO

array([[44410, 5712], [37123, 0], [25757, 2003], [ 1991, 1990], [ 2019, 2006]])

Como os dados foram sendo colocados nas linhas de acordo com a ordem original do array, as informações acabaram desconexas. Entretanto, usando order='F' para instruirmos a indexação da linguagem F, os dados serão distribuídos da maneira correta:

np.array(info\_carros).reshape((5, 2), order='F')COPIAR CÓDIGO

array([[44410, 2003], [ 5712, 1991], [37123, 1990], [ 0, 2019], [25757, 2006]])

Agora a primeira linha contém as informações do primeiro carro, a segunda contém as do segundo carro e assim por diante. Esses são alguns "macetes" que vamos aprendendo conforme trabalhamos com dados, e são muito úteis para manipularmos as informações que utilizamos.

O último método que conheceremos nesse vídeo é o **resize()**, que altera a forma e o tamanho do array. Preste atenção nesse ponto, pois incorreremos em um erro propositalmente para ilustrarmos uma situação bastante comum no dia-a-dia de quem trabalha como cientista de dados.

Criaremos uma variável dados\_new que receberá uma cópia do array dados, ou seja, dados.copy().

dados\_new = dados.copy()

dados\_newCOPIAR CÓDIGO

Agora imagine que queremos calcular a quilometragem média desses carros e adicioná-la a uma nova linha do array dados\_new. Para isso, primeiramente usaríamos o resize() passando como parâmetro a tupla (3, 5), representando 3 linhas e 5 colunas.

dados\_new.resize((3, 5))COPIAR CÓDIGO

Como retorno teremos um erro informando que como esses dados são referenciados, é necessário passarmos um parâmetro refcheck-=false, que justamente informa que o Numpy não deverá checar essa referência.

dados\_new.resize((3, 5), refcheck=False)

dados\_newCOPIAR CÓDIGO

array([[44410., 5712., 37123., 0., 25757.], [ 2003., 1991., 1990., 2019., 2006.], [ 0., 0., 0., 0., 0.]])

Com isso, conseguiremos criar a linha extra no array, e poderemos preenchê-las com as novas informações. Para isso, atribuiremos à linha dados\_new[2] o cálculo de quilometragem média com o qual já estamos acostumados.

dados\_new[2] = dados\_new[0] / (2019 - dados\_new[1])

dados\_newCOPIAR CÓDIGO

array([[44410. , 5712. , 37123. , 0. ,

25757. ],

[ 2003. , 1991. , 1990. , 2019. ,

2006. ],

[ 2775.625 , 204. , 1280.10344828, nan,

1981.30769231]])COPIAR CÓDIGO

No próximo vídeo veremos algumas estatísticas descritivas que podemos fazer com arrays Numpy.