

# NÃO PODE FALTAR

## A LINGUAGEM PYTHON

Vanessa Cadan Scheffer

0

Ver anotações

### PRIMEIROS PASSOS EM PYTHON

Vamos conhecer o poder dessa linguagem de programação.



Fonte: Shutterstock.

### Deseja ouvir este material?

Áudio disponível no material digital.

In [1]:

```
print("hello world!")
```

```
hello world!
```

Diz a lenda dos programadores que, se você não imprimir o “hello world” quando começar a aprender uma linguagem de programação, você não conseguirá aprender nada da linguagem em questão (<https://cienciacomputacao.com.br>). Mito ou verdade? Por via das dúvidas, essa foi nossa primeira linha de comando, de muitas que aprenderemos nessa disciplina. Como você pode ver, um comando de sintaxe “limpa”, sem ponto e vírgula, sem colchetes, sem importação de bibliotecas é a essência da linguagem de programação Python, que foi criada para ter comandos e estruturas simples, de fácil leitura e compreensão.

Python foi criado no início dos anos 1990 por Guido van Rossum no Stichting Mathematisch Centrum (CWI), na Holanda, como sucessor de uma linguagem chamada ABC. Guido é o principal autor do Python, embora haja muitas contribuições de outros pesquisadores ([Python Reference Manual, 2020](#)). Guido passou por outras instituições, como a CNRI, em 1995, na Virginía (Corporação para Iniciativas Nacionais de Pesquisa) e na BeOpen, em 2000, onde formou a BeOpen PythonLabs. Em outubro do mesmo ano, a equipe do PythonLabs mudou-se para a Digital Creations (agora Zope Corporation). Em 2001, a Python Software Foundation (PSF) foi formada, uma organização sem fins lucrativos criada especificamente para possuir a propriedade intelectual relacionada ao Python.

Ver anotações

## POR QUE USAR PYTHON?

Segundo o guia de desenvolvimento para iniciantes Python ([Beginners Guide Overview](#)), Python é uma linguagem de programação orientada a objetos, clara e poderosa, comparável a Perl, Ruby, Scheme ou Java. Podemos destacar algumas características que tornam essa linguagem notável:

- Utiliza uma sintaxe elegante, facilitando a leitura dos programas que você escreve.
- É uma linguagem fácil de usar, o que torna o Python ideal para o desenvolvimento de protótipos e outras tarefas de programação ad-hoc, sem comprometer a manutenção.
- Vem com uma grande biblioteca padrão que suporta muitas tarefas comuns de programação, como se conectar a servidores da Web, pesquisar texto com expressões regulares, ler e modificar arquivos.

- Possui inúmeras bibliotecas que estendem seu poder de atuação.
- É uma linguagem interpretada, ou seja, uma vez escrito o código, este não precisa ser convertido em linguagem de máquina por um processo de compilação.
- Permite atribuição múltipla. Podemos atribuir valores a mais de uma variável em uma única instrução. Por exemplo, `a, b = 2, 3`.

Uma das grandes características da linguagem é sua sintaxe. Uma das principais ideias de Guido é que o código é lido com muito mais frequência do que está escrito ([PEP 8 - Style Guide for Python Code](#)). Tal aspecto é tão relevante, que um código que segue as regras do idioma python é chamado de “pythonic code”. Essas regras são definidas pelo PEP 8 (Python Enhancement Proposal) e dizem respeito a formatação, identação, parâmetros em funções e tudo mais que possa estar relacionado à sintaxe do código.

Python tem se mostrado uma linguagem muito poderosa e está sendo amplamente adotada por profissionais na área de dados.

O interpretador Python 3 utiliza unicode por padrão, o que torna possível usar nomes de variáveis com acento e até outros caracteres especiais, porém não é uma boa prática. Tal flexibilidade mostra porque a linguagem tem sido adotada por pessoas que não são da área de exatas, pois a leitura está mais próxima da interpretação humana do que da máquina

## ONDE PROGRAMAR?

Agora que já conhecemos um pouco dessa importante linguagem e já sabemos que existe até um “código” de escrita determinado pelo PEP 8, podemos nos perguntar: mas onde eu escrevo os códigos em Python e vejo os resultados? A implementação de códigos em Python pode ser feita tanto em ferramentas instaladas no seu computador quanto em ambientes em nuvem. Nesse universo de possibilidades, o primeiro ponto que você precisa entender é que, para obter os resultados de um código, precisamos ter um interpretador Python. Vamos começar analisando as opções que podem ser instaladas.

## INSTALAÇÃO DO INTERPRETADOR PYTHON

Ver anotações

Na página oficial da linguagem (<https://www.python.org/downloads/>), podemos encontrar as várias versões do interpretador Python puro (sem bibliotecas de terceiros), disponíveis para diversos sistemas operacionais. Para instalar, basta escolher o sistema operacional, fazer o download da versão mais atual do interpretador 3.X e, após concluir o download, clique no arquivo executável que foi descarregado do repositório oficial.

No processo de instalação do interpretador Python, é preciso marcar a opção Add Python 3.X to PATH (Figura 1.1). Ao marcar essa opção, a variável de ambiente Path é configurada para entender o comando “python”. Caso não seja marcada a opção, você precisará configurar manualmente depois.

Ver anotações

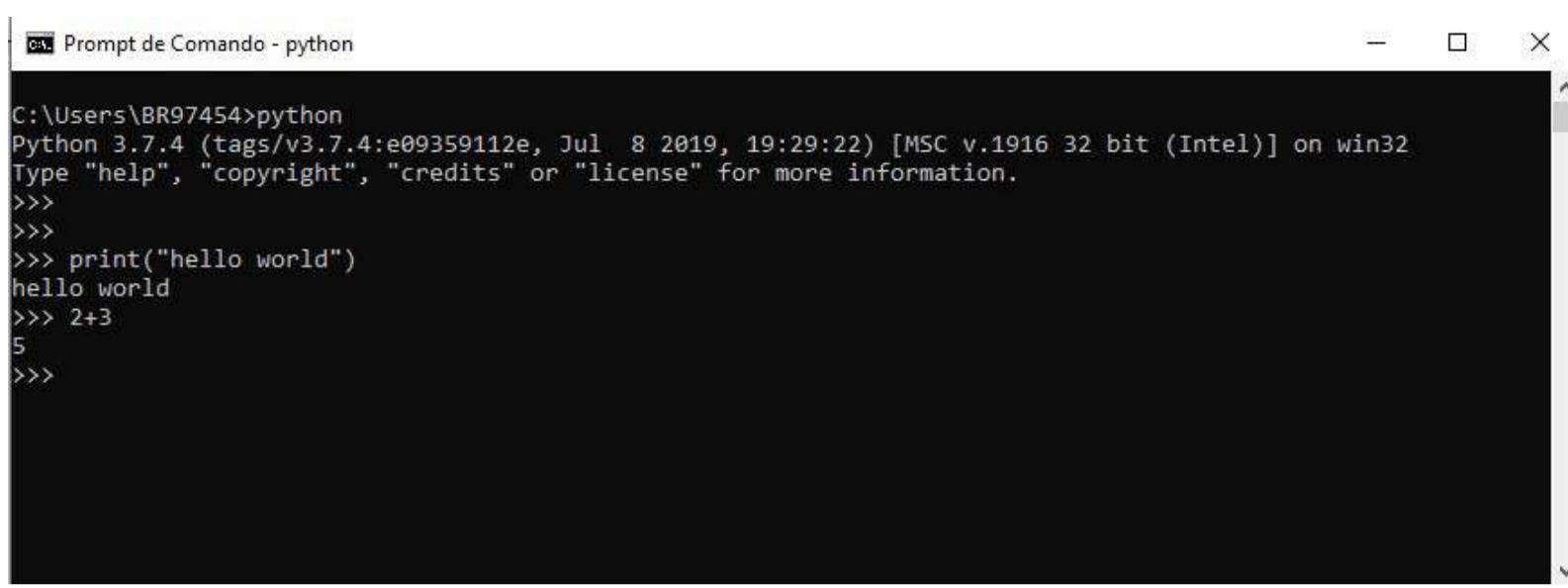
Figura 1.1 | Processo de instalação do interpretador



Fonte: captura de tela do instalador do Python.

Ao concluir a instalação do interpretador, se você marcou a opção “Add Python 3.X to PATH”, já podemos começar a programar através do modo iterativo. Para isso, abra o prompt de comando do Windows e digite “python” (Figura 1.2). Após aberto o modo iterativo, já podemos digitar comandos python e, ao apertar “enter”, o comando imediatamente é interpretado e o resultado é exibido.

Figura 1.2 | Modo de programação iterativo



```
C:\Users\BR97454>python
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul 8 2019, 19:29:22) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>>
>>> print("hello world")
hello world
>>> 2+3
5
>>>
```

Fonte: captura de tela do prompt de comando.

Ver anotações 0

Caso não queira navegar até o prompt de comando, você também pode utilizar o aplicativo IDLE, já pré-instalado no Windows.

## IDE'S PYCHARM E VISUAL STUDIO CODE

Utilizar o modo iterativo é muito simples e recomendado para testes rápidos, porém essa forma de implementação não é aplicável quando precisamos criar um código que será executado mais de uma vez. Isso acontece porque no modo iterativo, ao fechar o prompt, todo o código é perdido.

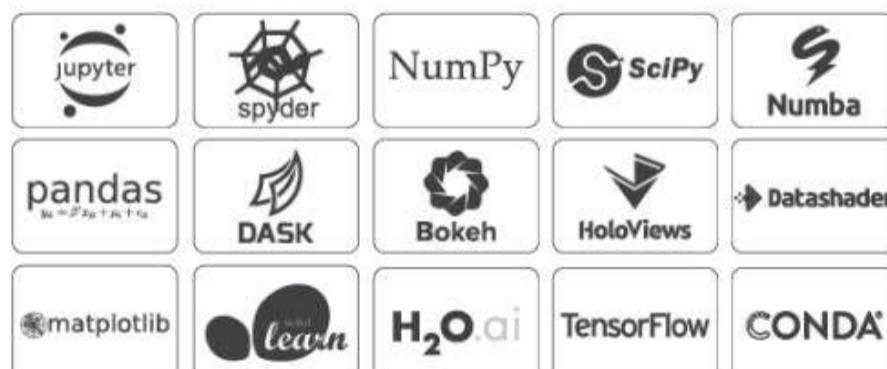
Para implementação de soluções (e não somente testes rápidos), nós programadores usamos uma IDE, (Integrated Development Environment) que traduzindo significa Ambiente de Desenvolvimento Integrado. Esse tipo de software possui uma série de ferramentas que auxiliam o desenvolvimento, como integração com sistemas de versionamento de código, refatoração de código, debug, etc. No mercado, duas IDE's disputam a preferência dos desenvolvedores Python, o PyCharm (<https://www.jetbrains.com/pycharm/>) e o Visual Studio Code (VSCode) (<https://visualstudio.microsoft.com/pt-br/>).

O PyCharm é oferecido em duas opções: Professional e Community, sendo a primeira paga e a segunda gratuita. Portanto, caso opte por essa IDE, você deverá fazer o download da versão Community no site (<https://www.jetbrains.com/pycharm/download/>). Já o VSCode é gratuito e pode ser encontrado no site (<https://visualstudio.microsoft.com/pt-br/>).

## PROJETO PYTHON ANACONDA

Para quem está começando a aprender Python, uma opção de ferramenta de trabalho é o projeto Python Anaconda (<https://www.anaconda.com/distribution/>). O projeto Anaconda consiste na união de diversas ferramentas Python, compostas por bibliotecas e IDE's (Figura 1.3).

Figura 1.3 | Ferramentas disponíveis no projeto Anaconda



Fonte: captura de tela de Anaconda (2020).

Fazendo a instalação do projeto Anaconda, você passa a ter tanto o interpretador Python quanto várias bibliotecas, além de duas interfaces de desenvolvimento: a IDE spyder e o projeto Jupyter. A IDE spyder, é similar ao PyCharm e ao VSCode, ou seja, é um ambiente de desenvolvimento com várias funcionalidades integradas.

Um dos grandes diferenciais do projeto Anaconda é ter o Jupyter Notebook (<https://jupyter.org/>) integrado na instalação.

O Jupyter Notebook é um ambiente de computação interativa, que permite aos usuários criar documentos de notebook que incluem: código ativo, gráficos, texto narrativo, equações, imagens e vídeo. Esses documentos fornecem um registro completo e independente de uma computação (um código) que pode ser convertida em vários formatos e compartilhada com outras pessoas usando email, Dropbox, sistemas de controle de versão (como git / GitHub). Uma das grandes vantagens do Jupyter Notebook é funcionar em um navegador de internet. No endereço (<https://jupyter.org/try>), você pode experimentar a ferramenta, sem precisar de instalação. Porém, instalando o projeto Python Anaconda, essa ferramenta já é instalada e você pode acessar fazendo a busca por Jupyter Notebook nos aplicativos do seu sistema operacional.

Em um Jupyter Notebook, o programador pode escrever trechos de códigos em células e, assim que executa a célula, o trecho é intrepretado e o resultado aparece logo abaixo, da mesma forma como fizemos no primeiro comando `print("hello world")`. No canal do Caio Dallaqua, você encontra um vídeo (<https://www.youtube.com/watch?v=m0FbNLhNyQ8>) de 2 minutos, com uma rápida introdução ao uso dessa ferramenta.

0

Ver anotações

## GOOGLE COLABORATORY (COLAB)

Colaboratory, ou "Colab", é um produto da Pesquisa do Google (<https://research.google.com/colaboratory/faq.html>). O Colab permite que qualquer pessoa escreva e execute código Python através do navegador. Além disso, é especialmente adequado para aprendizado de máquina, análise de dados e educação. Mais tecnicamente, o Colab é um serviço de notebook Jupyter hospedado que não requer configuração para ser usado, além de fornecer acesso gratuito a recursos de computação, incluindo GPUs.

O Colab é baseado no projeto de código aberto Jupyter. Portanto, o Colab permite que você use e compartilhe os notebooks Jupyter com outras pessoas sem precisar baixar, instalar ou executar nada. Ao criar um Jupyter Notebook no Colab, você pode compartilhar o link com pessoas específicas, ou então, compartilhar o trabalho no GitHub.

Para começar a usar o Colab, basta acessar o endereço

<https://colab.research.google.com/notebooks/> ou

[https://colab.research.google.com/notebooks/intro.ipynb?utm\\_source=scs-index](https://colab.research.google.com/notebooks/intro.ipynb?utm_source=scs-index) e desfrutar das vantagens dessa ferramenta.

***Devido a facilidade e vantagens do Colab, indicamos essa ferramenta como principal meio de trabalho para essa disciplina.***

Faça o link do seu Colab com seu Google drive, copie os códigos que serão apresentados, altere-os, explore!

Ver anotações

## MÃO NA MASSA!

Agora que já conhecemos um pouco do poder dessa linguagem de programação e já sabemos onde programar, chegou a hora de colocar a mão na massa e começarmos a codificar.

0

### VARIÁVEIS E TIPOS BÁSICOS DE DADOS EM PYTHON

Variáveis são espaços alocados na memória RAM para guardar valores temporariamente. Em Python, esses espaços não precisam ser tipados, ou seja, a variável pode ser alocada sem especificar o tipo de dado que ela aguardará. O interpretador Python é capaz de determinar o tipo de dado da variável com base no seu valor, ou seja, as variáveis são tipadas dinamicamente nessa linguagem.

Ver anotações

Veja alguns exemplos:

In [2]:

```
x = 10
nome = 'aluno'
nota = 8.75
fez_inscricao = True
```

Para saber o tipo de dado que uma variável guarda, podemos imprimir seu tipo usando a função `type()`, veja como:

In [3]:

```
print(type(x))
print(type(nome))
print(type(nota))
print(type(fez_inscricao))
```

```
<class 'int'>
<class 'str'>
<class 'float'>
<class 'bool'>
```

A célula de entrada 3 (In [3]) têm os comandos para imprimir na tela, os tipos das quatro variáveis que criamos anteriormente. Veja que foram impressas quatro diferentes classes de tipos de dados. A variável "x" é do tipo inteira (int). A variável "nome" é do tipo string (str). A variável "nota" é do tipo ponto flutuante (float). A variável "fez\_inscricao" é do tipo booleana (bool).

Em Python, tudo é objeto! Por isso os tipos de dados aparecem com a palavra "class", que significa classe. Teste você mesmo os códigos no emulador a seguir e aproveite para explorar outras variáveis!

The screenshot shows a Jupyter Notebook interface. At the top, there's a toolbar with icons for file operations, code editing, running cells, sharing, and remixing. Below the toolbar, the title bar displays "trinket" and "Python3". A dropdown menu shows "Python 3" is selected. The main area contains a single code cell with the following content:

```
< > matematica_fundamentos_01.csv
```

Escolher arquivo Nenhum arquivo escolhido

The cell has been run, and its output is displayed below it:

```
3
```

0

Ver anotações

Agora que já sabemos como criar variáveis, vamos aprimorar nosso hello world.

Vamos solicitar que o usuário digite um nome, para que possamos fazer uma saudação personalizada. A função `input()` faz a leitura de um valor digitado. Veja como usar:

In [4]:

```
nome = input("Digite um nome: ")  
print(nome)
```

```
Digite um nome: João  
João
```

Veja que dentro da função `input()`, colocamos a mensagem que irá aparecer na tela, o que o usuário digitar será capturado e guardado dentro da variável "nome". Na linha em seguida, imprimimos o que o usuário digitou. Mas como podemos combinar a frase "hello world" com o nome digitado? Como vamos imprimir variáveis e textos juntos? Como podemos imprimir: "Olá João! Bem-vindo à disciplina de programação. Parabéns pelo seu primeiro hello world"?

Temos uma variedade de formas de imprimir texto e variável em Python. Vejamos algumas: podemos usar formatadores de caracteres (igual em C), podemos usar a função `format()` e podemos criar uma string formatada. Vejamos cada uma delas:

In [5]:

```
# Modo 1 - usando formatadores de caracteres (igual na linguagem C) para imprimir variável e texto  
print("Olá %s, bem vindo a disciplina de programação. Parabéns pelo seu primeiro hello world" % (nome))
```

```
Olá João, bem vindo a disciplina de programação. Parabéns pelo seu primeiro hello world
```

In [6]:

```
# Modo 2 - usando a função format() para imprimir variável e texto  
print("Olá {}, bem vindo a disciplina de programação. Parabéns pelo seu primeiro hello world".format(nome))
```

```
Olá João, bem vindo a disciplina de programação. Parabéns pelo seu primeiro hello world
```

In [7]:

```
# Modo 3 - usando strings formatadas  
print(f"Olá {nome}, bem vindo a disciplina de programação. Parabéns pelo seu primeiro hello world")
```

```
Olá João, bem vindo a disciplina de programação. Parabéns pelo seu primeiro hello world
```

Em primeiro lugar, como vocês podem observar, usamos o hash # para criar comentários de uma linha. Lembrando que em qualquer linguagem de programação, os comentários não são nem interpretados, nem compilados.

Os três modos usados obtiveram o mesmo resultado e você poderia adotar o que achar mais conveniente. Entretanto, lembra que existe um código para sintaxe do "pythonic code"? Pois bem, a PEP 498 (<https://www.python.org/dev/peps/pep-0498/>) fala sobre a criação de strings com interpolação (mistura de variáveis com texto"). Nessa PEP, o autor destaca o uso do "modo 3" como a melhor opção chamando-a de "f-strings". Portanto, em nossos códigos, iremos adotar essa sintaxe para a criação de strings com interpolação.

**As strings formatadas com "f-strings" só podem ser compiladas com o interpretador Python na versão 3.6. Se tentar usar essa sintaxe em um interpretador em versões anteriores, será dado erro de sintaxe.**

0

Ver anotações

**O uso do interpretador na versão 2 é fortemente desencorajado pela equipe Python, pois em 2020 ele será descontinuado. Para saber mais, leia a seção Update do endereço: <https://www.python.org/dev/peps/pep-0373/>**

## OPERAÇÕES MATEMÁTICAS EM PYTHON

O Quadro 1.1 apresenta um resumo das operações matemáticas suportadas por Python. Com exceção das funções `abs()` e `pow()` e da notação de potência `**`, as outras operações e sintaxe são similares a diversas linguagens de programação.

0

Ver anotações

## Quadro 1.1 | Operações matemáticas em Python

Operação	Resultado
$x + y$	soma de $x$ e $y$
$x - y$	Diferença de $x$ e $y$
$x * y$	Produto de $x$ e $y$
$x / y$	Quociente de $x$ e $y$
$x // y$	Parte inteira do quociente de $x$ e $y$
$x \% y$	Resto de $x / y$
$abs(x)$	Valor absoluto de $x$
$pow(x, y)$	$x$ elevado a $y$
$x ** y$	$x$ elevado a $y$

Fonte: adaptada de (<https://docs.python.org/3/library/stdtypes.html>)

0

Ver anotações

Com relação às operações matemáticas, seja na programação, seja na resolução analítica, o mais importante é lembrar a ordem de precedências das operações.

1. Primeiro resolvem-se os parênteses, do mais interno para o mais externo.
2. Exponenciação.
3. Multiplicação e divisão.
4. Soma e subtração.

Vejamos alguns exemplos. Repare como é fundamental conhecer a ordem de precedência das operações para não criar cálculos errados durante a implementação de uma solução. Teste você mesmo os códigos no emulador a seguir e aproveite para explorar outras operações.

In [8]:

```
# Qual o resultado armazendo na variável operacao_1: 25 ou 17?
```

```
operacao_1 = 2 + 3 * 5
```

```
# Qual o resultado armazendo na variável operacao_2: 25 ou 17?
```

```
operacao_2 = (2 + 3) * 5
```

```
# Qual o resultado armazendo na variável operacao_3: 4 ou 1?
```

```
operacao_3 = 4 / 2 ** 2
```

```
# Qual o resultado armazendo na variável operacao_4: 1 ou 5?
```

```
operacao_4 = 13 % 3 + 4
```

```
print(f"Resultado em operacao_1 = {operacao_1}")
```

```
print(f"Resultado em operacao_2 = {operacao_2}")
```

```
print(f"Resultado em operacao_3 = {operacao_3}")
```

```
print(f"Resultado em operacao_4 = {operacao_4}")
```

```
Resultado em operacao_1 = 17
```

```
Resultado em operacao_2 = 25
```

```
Resultado em operacao_3 = 1.0
```

```
Resultado em operacao_4 = 5
```



Agora vamos juntar tudo o que aprendemos até o momento. Uma equação do segundo grau possui a fórmula:  $y = a*x**2 + b*x + c$ , onde  $a$ ,  $b$ ,  $c$  são constantes. O valor de  $y$  (resultado) depende do valor de  $x$ , ou seja,  $x$  é a variável independente e  $y$  a dependente. Considerando os valores  $a = 2$ ,  $b = 0.5$  e  $c = 1$ , vamos solicitar para o usuário um valor de  $x$  e retornar o valor de  $y$  correspondente ao  $x$  que ele digitou. Veja como deve ser implementado:

In [9]:

```
a = 2
b = 0.5
c = 1
x = input("Digite o valor de x: ")

y = a * x ** 2 + b * x + c

print(f"O resultado de y para x = {x} é {y}.")
```

Digite o valor de x: 3

```
-----
TypeError                                     Traceback (most recent call last)
<ipython-input-9-42060e5b5536> in <module>
      4 x = input("Digite o valor de x: ")
      5
----> 6 y = a * x ** 2 + b * x + c
      7
      8 print(f"O resultado de y para x = {x} é {y}.")
```

TypeError: unsupported operand type(s) for \*\* or pow(): 'str' and 'int'

Ver anotações

Erro? E agora? Calma, erros fazem parte da vida do desenvolvedor. Vamos ler a mensagem de erro. Primeiro o erro foi do tipo "TypeError", isso quer dizer que alguma variável não está com o tipo adequado para a situação. A mensagem nos diz que não é permitida a operação de potenciação (\*\*) entre um tipo string e inteiro. Portanto, o erro é porque estamos tentando fazer uma operação matemática entre string e um tipo numérico. Então, dentre as variáveis deve ter uma com o tipo errado. Vamos usar a função type() para verificar o tipo das variáveis usadas.

In [10]:

```
print(type(a))
print(type(b))
print(type(c))
print(type(x))
```

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

Olhando o resultado da entrada 10 (In [10]), vemos que o tipo da variável x é string (str), isso acontece porque ao usar a função input(), ela retorna uma string, independente do que o usuário digitou, sempre será string. Portanto, para corrigir o erro, precisamos converter o resultado da entrada em um tipo numérico. Como não sabemos se o usuário vai digitar um número inteiro ou decimal, vamos fazer a conversão usando a função float(). Veja como fica o código:

In [11]:

```
a = 2
b = 0.5
c = 1
x = input("Digite o valor de x: ")

x = float(x) # aqui fazemos a conversão da string para o tipo numérico

y = a * x ** 2 + b * x + c

print(f"O resultado de y para x = {x} é {y}.")
```

Ver anotações

Digite o valor de x: 3

O resultado de y para x = 3.0 é 20.5.

Aprendemos bastante até aqui, agora é hora de colocar em prática. Não deixe de explorar os links apresentados, quanto mais você buscar informações, mais fluente ficará na linguagem de programação.

## REFERÊNCIAS E LINKS ÚTEIS

A ORIGEM do “Hello World”. **Ciência da Computação**, 2015. Disponível em:

<https://cienciacomputacao.com.br/curiosidade/a-origem-do-hello-world/>. Acesso em 25 maio 2020.

ANACONDA Individual Edition: The World's Most Popular Python/R Data Science Platform. The World's Most Popular Python/R Data Science Platform. Disponível em: <https://www.anaconda.com/distribution/>. Acesso em: 10 abr. 2020.

DALLAQUA, C. **Curso Python 3**: aula 1 - notebook jupyter. 1 vídeo ( 2,54 min.), 2016. Disponível em: <https://www.youtube.com/watch?v=m0FbNIhNyQ8>. Acesso em: 10 abr. 2020.

VISUAL Studio Code. Disponível em: <https://visualstudio.microsoft.com/pt-br/>. Acesso em: 10 abr. 2020.

FARIA, F. A. **Lógica de Programação**: aula 01 - introdução. São José dos Campos: Unifesp, 2016. 33 slides, color. Disponível em:  
<https://www.ic.unicamp.br/~ffaria/lp2s2016/class01/lp-aula01.pdf>. Acesso em: 10 abr. 2020.

GOOGLE. **Colaboratory**: Frequently Asked Questions. Disponível em:  
<https://research.google.com/colaboratory/faq.html>. Acesso em: 10 abr. 2020.

PYCHARM: The Python IDE for Professional Developers. Disponível em:  
<https://www.jetbrains.com/pycharm/>. Acesso em: 10 abr. 2020.

PYTHON Beginners Guide. Última atualização em 19 set. 2019, por Chris M. Disponível em: <https://wiki.python.org/moin/BeginnersGuide/Overview>. Acesso em: 10 abr. 2020.

PYTHON SOFTWARE FOUNDATION (org.). **History of the software**. Disponível em:  
<https://docs.python.org/3.0/license.html>. Acesso em: 10 abr. 2020.

PYTHON SOFTWARE FOUNDATION. **Numeric Types**: int, float, complex. Disponível em: <https://docs.python.org/3/library/stdtypes.html>. Acesso em: 10 abr. 2020.

PYTHON SOFTWARE FOUNDATION. **PEP 373**: Python 2.7 Release Schedule. Disponível em: <https://www.python.org/dev/peps/pep-0373/>. Acesso em: 10 abr. 2020.

ROSSUM, G. V.; WARSAW, B.; COGHLAN, N. **PEP 8**: Style Guide for Python Code. Disponível em: <https://www.python.org/dev/peps/pep-0008/#introduction>. Acesso em: 10 abr. 2020.

THE JUPYTER Notebook. Disponível em: <https://jupyter.org/>. Acesso em: 10 abr. 2020.

VISUAL Studio Code. Disponível em: <https://visualstudio.microsoft.com/pt-br/>. Acesso em: 10 abr. 2020.

Ver anotações

# FOCO NO MERCADO DE TRABALHO

## A LINGUAGEM PYTHON

Vanessa Cadan Scheffer

0

Ver anotações

### DESENVOLVENDO UM PROTÓTIPO

Implementando algoritmos em Python.



Fonte: Shutterstock.

### Deseja ouvir este material?

Áudio disponível no material digital.

### DESAFIO

No mercado de trabalho, existem diversas empresas que construíram seu modelo de negócio baseado na prestação de serviço especializado para outras empresas e são chamadas de "consultorias". Dentre essas empresas de consultoria, há uma grande procura pelas que desenvolvem software, assim a empresa contratante não precisa ter uma área de desenvolvimento interna, ela contrata a consultoria e faz a encomenda da solução que necessita.

Como seu primeiro trabalho de desenvolvedor em uma empresa de consultoria de software, você foi designado para antender um cliente que fabrica peças automotivas e criar um protótipo da solução que ele necessita. O cliente relata que tem aumentado o número de peças e que gostaria de uma solução que fosse capaz de prever quantas peças serão vendidas em um determinado mês. Esse resultado é importante para ele, uma vez que dependendo da quantidade, ele precisa contratar mais funcionários, reforçar seu estoque e prever horas extras.

O cliente enviou para você o relatório de vendas dos últimos 6 meses (Figura 1.4). Agora você precisa analisar o gráfico, pensar no algoritmo que, a partir das informações no gráfico, seja capaz de prever quantas peças serão vendidas em um determinado mês. Por exemplo, considerando o mês de janeiro como o primeiro mês, ele vendeu  $x$  peças, em fevereiro (segundo mês) ele vendeu  $n$  peças, quantas peças ele vai vender no mês 10, e no mês 11 e no mês 32? Por se tratar de um protótipo, você deve utilizar somente as informações que lhe foram cedidas, não precisa, nesse momento, analisar o comportamento de fatores externos, por exemplo, comportamento da bolsa de valores, tendência de mercado, etc.

Ver anotações

Figura 1.4 | Relatório de vendas



Fonte: elaborada pela autora.

Você precisa escolher qual ferramenta de trabalho irá adotar para criar o protótipo, em seguida implementar o algoritmo que faça a previsão usando a linguagem de programação Python.

0

## RESOLUÇÃO

Foi lhe dada a missão de escolher uma ferramenta para desenvolver um protótipo para o cliente que fabrica peças automotivas. Uma opção é usar o Colab, pois nessa ferramenta você consegue implementar seu algoritmo usando a linguagem Python... Outra grande vantagem em utilizá-lo é o fato de ser on-line e não precisar de instalação. Uma vez pronto o protótipo, você pode enviar o link, tanto para seu gerente ver o resultado do seu trabalho, quanto para o cliente testar a solução.

Uma vez decidida a ferramenta, é hora de começar a pensar na solução. Tudo que você tem de informação está em um gráfico, portanto é preciso interpretá-lo.

Vamos extrair as informações de venda do gráfico e escrever em forma de tabela (Tabela 1.1).

Tabela 1.1 | Venda de peças

Mês	Resultado	Aumento
1	200	-
2	400	200
3	600	200
4	800	200
5	1000	200
6	1200	200

Fonte: elaborada pela autora.

Ver anotações

Ao tabular os dados do gráfico, aparece um valor interessante na coluna que mostra o aumento mês após mês. De acordo com as informações o aumento tem sido constante.

Se o aumento é constante, podemos usar uma função do primeiro grau para prever qual será o resultado em qualquer mês. A função será  $r = c * \text{mes}$ , onde,  $r$  é o resultado que queremos,  $c$  é a constante de crescimento e  $\text{mes}$  é a variável de entrada. Dessa forma, ao obter um mês qualquer (2, 4, 30, etc) podemos dizer qual o resultado.

Vamos testar nossa fórmula:

- $\text{mes} = 2; c = 200 \rightarrow r = 200 * 2 = 400$  (Valor correto para o mês 2).
- $\text{mes} = 3; c = 200 \rightarrow r = 200 * 3 = 600$  (Valor correto para o mês 3).
- $\text{mes} = 4; c = 200 \rightarrow r = 200 * 4 = 800$  (Valor correto para o mês 4).
- $\text{mes} = 5; c = 200 \rightarrow r = 200 * 5 = 1000$  (Valor correto para o mês 5).

Agora que já sabemos como resolver, vamos implementar usando a linguagem Python. Veja a seguir o código.

In [1]:

```
c = 200 # valor da constante

mes = input("Digite o mês que deseja saber o resultado: ") # Função para captura o mês que o cliente digitar
mes = int(mes) # Não esqueça de converter para numérico o valor captura pela função input()

r = c * mes # Equação do primeiro grau, também chamada função do primeiro grau ou de função linear.

print(f"A quantidade de peças para o mês {mes} será {r}") # Impressão do resultado usando string interpolada "f-strings" (PEP 498)
```

Digite o mês que deseja saber o resultado: 30

A quantidade de peças para o mês 30 será 6000

## DESAFIO DA INTERNET

Que tal treinar um pouco mais de programação e conhecer novas fontes de informações? Você, aluno, tem acesso à Biblioteca Virtual, um repositório de livros e artigos que pode ser acessado no endereço: (<http://biblioteca-virtual.com/>).

Na página 47 (capítulo 2: Objetos e Comandos de Entrada e Saída em Python) da obra: BANIN, S. L. **Python 3 - conceitos e aplicações: uma abordagem didática.** São Paulo: Érica, 2018, você encontra o exercício proposto 1. Utilize o emulador a seguir para resolver o desafio!

0  
Ver anotações

Utilize o emulador a seguir, para resolver o desafio!



NÃO PODE FALTAR

# ESTRUTURAS LÓGICAS, CONDICIONAIS E DE REPETIÇÃO EM PYTHON

Vanessa Cadan Scheffer

0

Ver anotações

## AS ESTRUTURAS DE COMANDO EM PYTHON

Vamos conhecer alguns trechos de código e sua lógica de execução.



Fonte: Shutterstock.

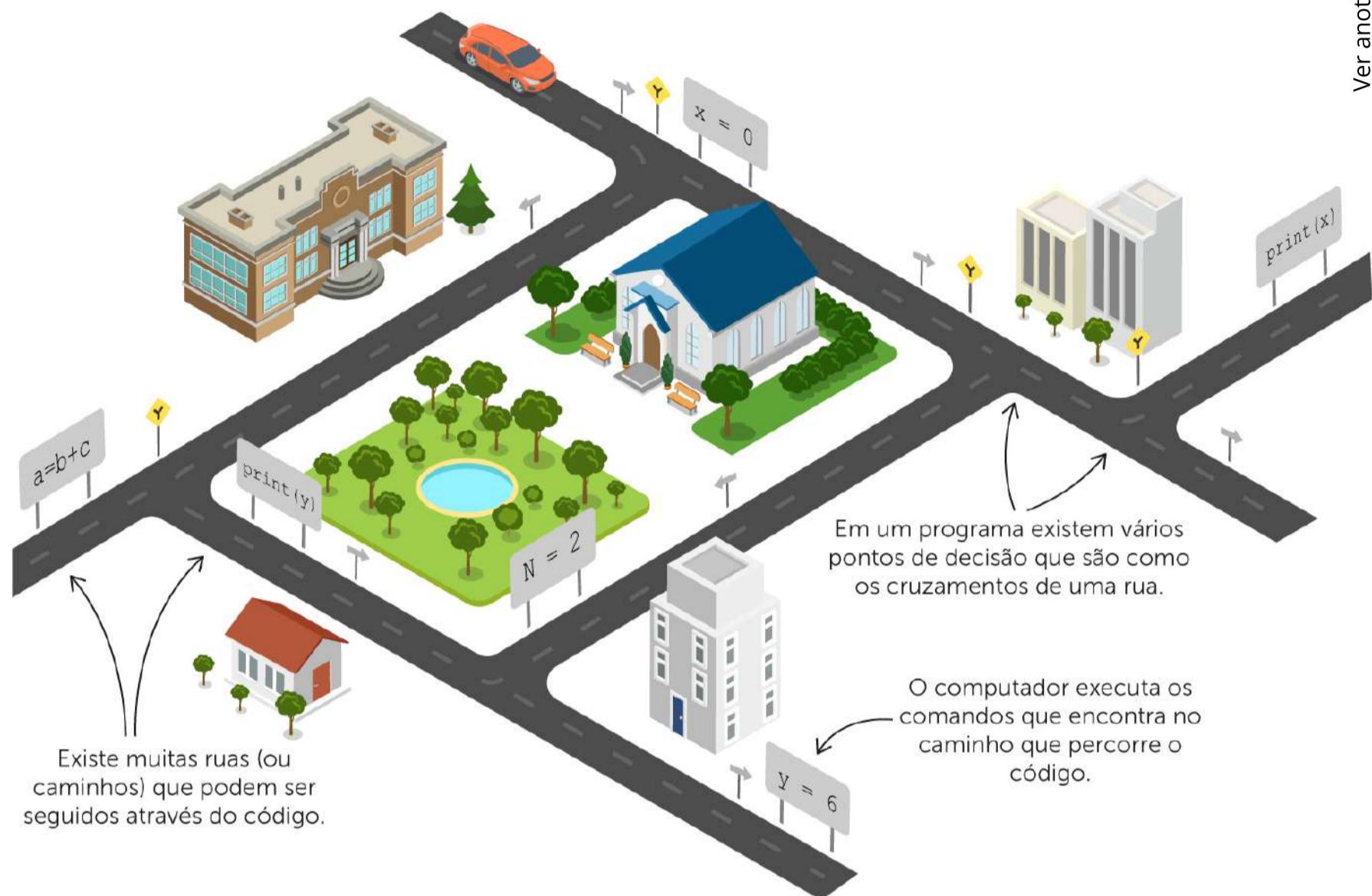
### Deseja ouvir este material?

Áudio disponível no material digital.

Criamos algoritmos para resolver problemas, o que implica tomar decisões. No universo da programação, a técnica que nos permite tomar decisões é chamada de estrutura condicional (MANZANO; OLIVEIRA, 2019). A Figura 1.5 ilustra uma cidade e suas vias. Ao dirigir em uma cidade, você precisa decidir quais caminhos (ruas) seguir com o objetivo de chegar a um destino; por exemplo, em um cruzamento, pode-se virar à direita ou à esquerda e/ou seguir em frente. Um programa funciona de forma similar: deve-se decidir qual caminho será executado em um código. Em geral, em um

programa você tem opções de caminhos ou lista de comandos que nada mais são que trechos de códigos que podem ser executados, devendo-se tomar decisões sobre qual trecho de código será executado em um determinado momento.

Figura 1.5 | Fluxo de tomada de decisões de um algoritmo para a escolha de um caminho em uma cidade.



Fonte: adaptada de Griffiths e Barry (2009, p. 13).

Para tomarmos decisões, precisamos dos operadores relacionais (Quadro 1.2), que já fazem parte do nosso vocabulário, desde as aulas de matemática do colégio. Vamos usá-los para fazer comparações entre objetos (lembrando que em Python tudo é objeto, então uma variável também é um objeto).

## Quadro 1.2 - Operadores relacionais

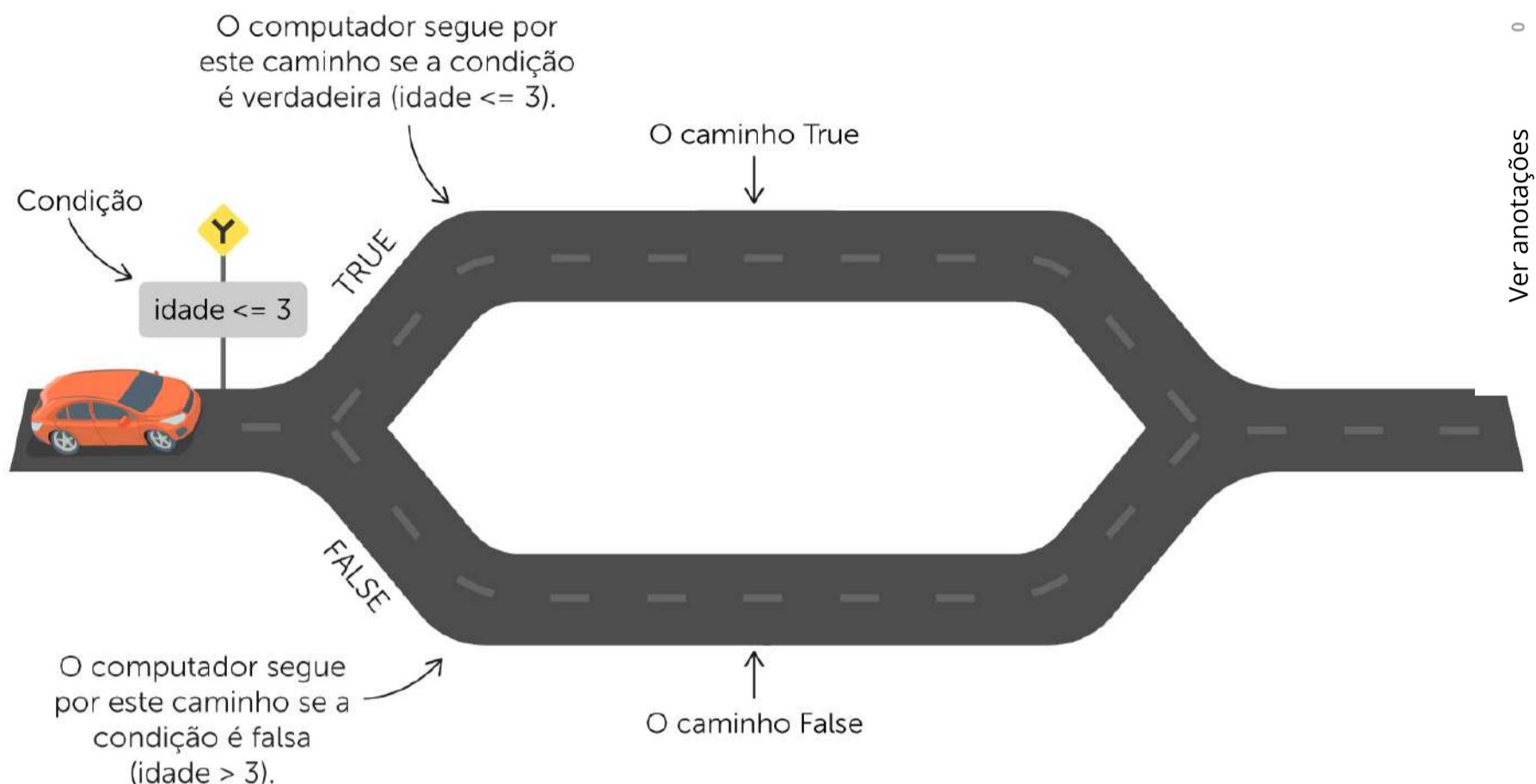
Operação	Significado
a < b	O valor de a é menor que b?
a <= b	O valor de a é menor OU igual que b?
a > b	O valor de a é maior que b?
a >= b	O valor de a é maior OU igual que b?
a == b	O valor de a é igual ao de b?
a != b	O valor de a é diferente do valor de b?
a is b	O valor de a é idêntico ao valor de b?
a is not b	O valor de a não é idêntico ao valor de b?

0  
Ver anotaçõesFonte: adaptado de <https://docs.python.org/3/library/stdtypes.html>.**ESTRUTURAS CONDICIONAIS EM PYTHON: IF, ELIF, ELSE**

O comando if.. else.. significam se.. senão.. e são usados para construir as estruturas condicionais. Se uma condição for satisfeita, então o fluxo de execução segue por um caminho, se não for satisfeito então segue por outro caminho. A Figura 1.6 ilustra um exemplo de tomada de decisão. No exemplo, dois caminhos podem ser seguidos pelo condutor: um verdadeiro (TRUE), e um falso (FALSE). Assim funciona uma condição: ela pode ser satisfeita (Verdade) ou não (Falsa). No exemplo, se a idade for menor ou igual a 3, o condutor seguirá o caminho verdadeiro. Por sua vez, se ela for maior que 3, seguirá o caminho falso.

Figura 1.6 | Tomada de decisão utilizando Se/senão (If/else)

## if / else



Fonte: adaptada de Griffiths e Barry (2009, p. 13).

Python possui uma sintaxe especial para a construção das estruturas condicionais.

Observe o trecho a seguir:

In [1]:

```
a = 5
b = 10

if a < b:
    print("a é menor do que b")
    r = a + b
    print(r)
```

```
a é menor do que b
```

```
15
```

Como você pode observar, a construção do if (se) deve ser feita com dois pontos ao final da condição, e o bloco de ações que devem ser feitas. Caso o trecho seja verdadeiro, deve possuir uma indentação específica de uma tabulação ou 4 espaços em branco. Repare que não precisamos escrever `if a < b == True`.

O código apresentado na entrada 1(In [1]) refere-se a uma estrutura condicional simples, pois só existem ações caso a condição seja verdadeira. Veja como construir uma estrutura composta:

In [2]:

```
a = 10
b = 5

if a < b:
    print("a é menor do que b")
    r = a + b
    print(r)
else:
    print("a é maior do que b")
    r = a - b
    print(r)
```

a é maior do que b

5

0

Ver anotações

O código apresentado na entrada 2(In [2]) refere-se a uma estrutura condicional composta, pois existem ações tanto para a condição verdadeira quanto para a falsa. Veja que o else (senão) está no mesmo nível de identação do if, e os blocos de ações com uma tabulação de indentação.

Além das estruturas simples e compostas, existem as estruturas encadeadas. Em diversas linguagens como C, Java, C#, C++, etc, essa estrutura é construída como o comando switch..case. Em Python, não existe o comando switch (<https://docs.python.org/3/tutorial/controlflow.html>). Para construir uma estrutura encadeada, devemos usar o comando "elif", que é uma abreviação de else if. Veja um exemplo desse tipo de estrutura:

In [3]:

```
codigo_compra = 5111

if codigo_compra == 5222:
    print("Compra à vista.")
elif codigo_compra == 5333:
    print("Compra à prazo no boleto.")
elif codigo_compra == 5444:
    print("Compra à prazo no cartão.")
else:
    print("Código não cadastrado")
```

Código não cadastrado

0

Ver anotações

O código apresentado na entrada 3 (In [3]) mostra a construção de uma estrutura condicional encadeada. Veja que começamos pelo teste com `if`, depois testamos várias alternativas com o `elif` e, por fim, colocamos uma mensagem, caso nenhuma opção seja verdadeira, no `else`.

## ESTRUTURAS LÓGICAS EM PYTHON: AND, OR, NOT

Além dos operadores relacionais, podemos usar os operadores booleanos para construir estruturas de decisões mais complexas.

- Operador booleano `and`: Esse operador faz a operação lógica E, ou seja, dada a expressão (`a and b`), o resultado será True, somente quando os dois argumentos forem verdadeiros.
- Operador booleano `or`: Esse operador faz a operação lógica OU, ou seja, dada a expressão (`a or b`), o resultado será True, quando pelo menos um dos argumentos for verdadeiro.
- Operador booleano `not`: Esse operador faz a operação lógica NOT, ou seja, dada a expressão (`not a`), ele irá inverter o valor do argumento. Portanto, se o argumento for verdadeiro, a operação o transformará em falso e vice-versa.

Observe o exemplo a seguir, que cria uma estrutura condicional usando os operadores booleanos. Um aluno só pode ser aprovado caso ele tenha menos de 5 faltas e média final superior a 7.

In [4]:

```
qtde_faltas = int(input("Digite a quantidade de faltas: "))

media_final = float(input("Digite a média final: "))

if qtde_faltas <= 5 and media_final >= 7:
    print("Aluno aprovado!")
else:
    print("Aluno reprovado!")
```

Digite a quantidade de faltas: 2

Digite a média final: 7

Aluno aprovado!

0

Ver anotações

No exemplo da entrada 4 (In [4]), fizemos a conversão do tipo string para numérico, encadeando funções. Como já sabemos, primeiro são resolvidos os parênteses mais internos, ou seja, na linha 1, será feita primeiro a função input(), em seguida, a entrada digitada será convertida para inteira com a função int(). A mesma lógica se aplica na linha 2, na qual a entrada é convertida para ponto flutuante. Na linha 4 fizemos o teste condicional, primeiro são resolvidos os operadores relacionais, em seguida, os lógicos.

**Obs: not tem uma prioridade mais baixa que os operadores relacionais.**

**Portanto, not a == b é interpretado como: not (a == b) e a == not b gera um erro de sintaxe.**

**Obs2: Assim como as operações matemáticas possuem ordem de precedência, as operações booleanas também têm. Essa prioridade obedece à seguinte ordem: not primeiro, and em seguida e or por último (BANIN, 2018).**

Veja o exemplo a seguir:

In [5]:

```
A = 15
B = 9
C = 9

print(B == C or A < B and A < C)

print((B == C or A < B) and A < C )
```

True

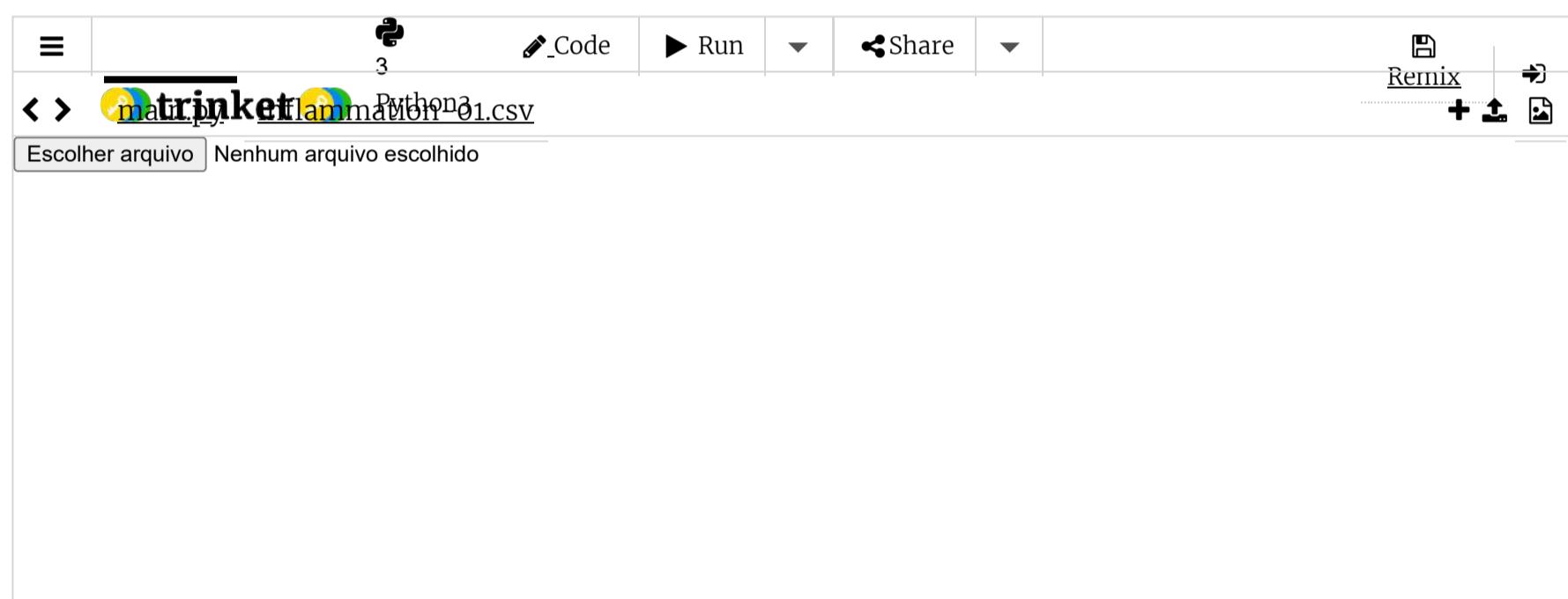
False

Na entrada 5 (In [5]), temos os seguintes casos:

Linha 5: True or False and False. Nesse caso será feito primeiro o and, que resulta False, mas em seguida é feito o or, que então resulta em verdadeiro.

Linha 6: (True or False) and False. Nesse caso será feito primeiro o or, que resulta True, mas em seguida é feito o and, que então resulta em falso, pois ambos teriam que ser verdadeiro para o and ser True.

Utilize o emulador a seguir para testar os códigos. Crie novas variáveis e condições para treinar.



## ESTRUTURAS DE REPETIÇÃO EM PYTHON: WHILE E FOR

É comum, durante a implementação de uma solução, criarmos estruturas que precisam executar várias vezes o mesmo trecho de código. Tal estrutura é chamada de estrutura de repetição e, em Python, pode ser feita pelo comando while (enquanto) ou pelo comando for (para). Antes de apresentarmos a implementação, observe a Figura 1.7. Temos a ilustração de uma estrutura de repetição por meio de uma montanha russa. Enquanto uma condição for verdadeira, ou seja, enquanto  $x$  for menor ou igual a 3, o condutor continuará no loop; quando for falsa, ou seja, quando  $x$  for maior que 3, então ele sairá.

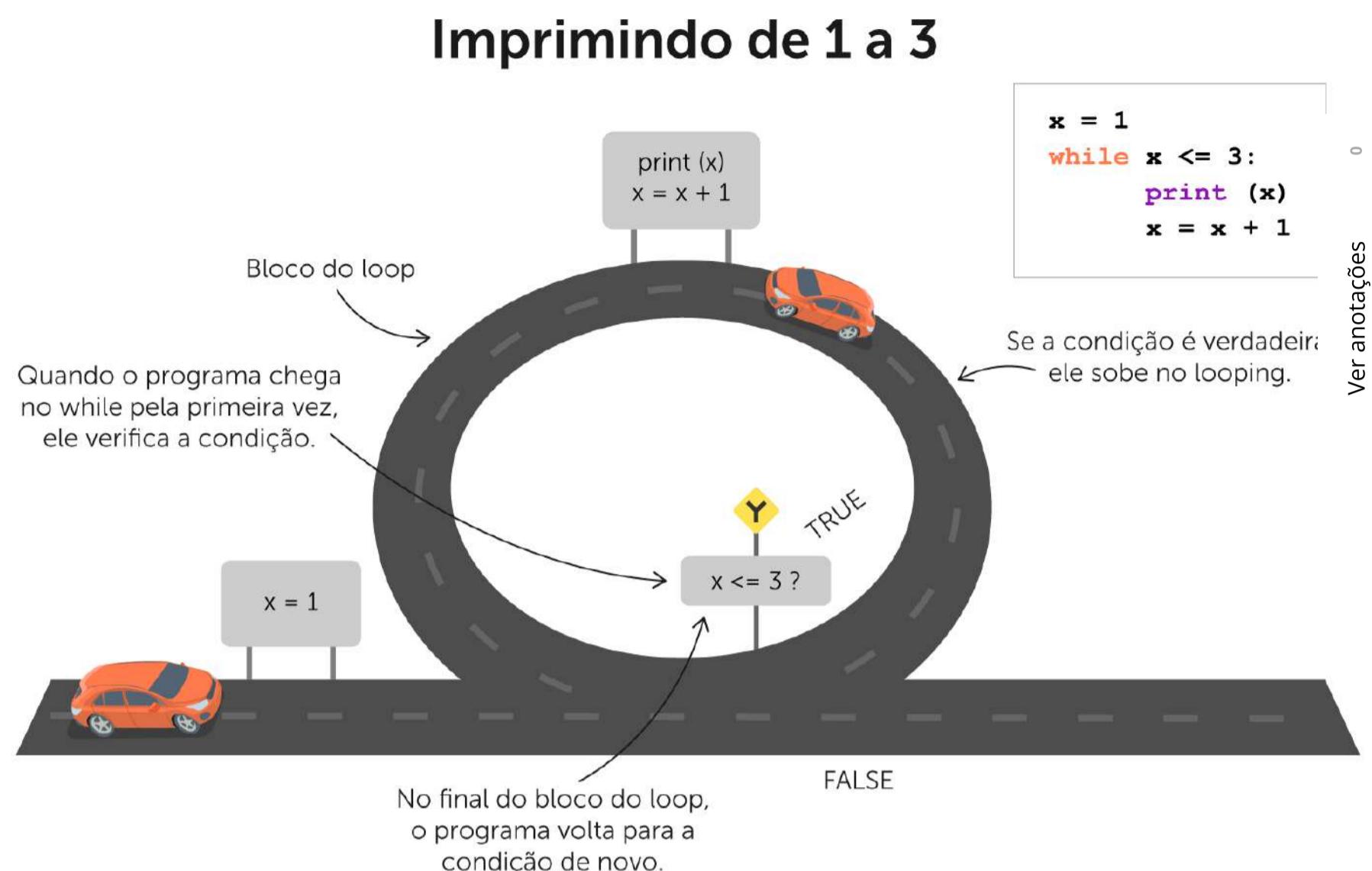
Em uma estrutura de repetição sempre haverá uma estrutura decisão, pois a repetição de um trecho de código sempre está associada a uma condição. Ou seja, um bloco de comandos será executado repetidas vezes, até que uma condição não

seja mais satisfeita.

0

Ver anotações

Figura 1.7 | Fluxograma estrutura de repetição



Fonte: adaptada de Griffiths e Barry (2009, p. 28).

O comando while deve ser utilizado para construir e controlar a estrutura decisão, sempre que o número de repetições não seja conhecido. Por exemplo, quando queremos solicitar e testar se o número digitado pelo usuário é par ou ímpar. Quando ele digitar zero, o programa se encerra. Veja, não sabemos quando o usuário irá digitar, portanto somente o while é capaz de solucionar esse problema. Veja a implementação desse problema a seguir.

In [6]:

```
numero = 1
while numero != 0:
    numero = int(input("Digite um número: "))
    if numero % 2 == 0:
        print("Número par!")
    else:
        print("Número ímpar!")
```

```
Digite um número: 4
```

Número par!

```
Digite um número: 7
```

Número ímpar!

```
Digite um número: 0
```

Número par!

A seguir, você pode testar o código, verificando o passo a passo de sua execução.

clicar em "Next", você irá para a próxima linha a ser executada do "while" para cada um dos casos. Ao clicar em "Prev", você retorna a linha executada anteriormente.

0

[Ver anotações](#)

Observe que a seta verde na linha de código representa a linha em execução, e a seta vermelha representa a próxima linha a ser executada. Digite um número, submeta, clique em "Next" e observe o fluxo de execução e as saídas do código.

Veja a construção do comando while na entrada 6 (In [6]). Na linha 1, criamos uma variável chamada número com valor 1. Na linha 2 criamos a estrutura de repetição, veja que o comando while possui uma condição: o número tem que ser diferente de zero para o bloco executar. Todo o bloco com a identação de uma tabulação (4 espaços) faz parte da estrutura de repetição. Lembre: todos os blocos de comandos em Python são controlados pela indentação.

Na prática é comum utilizarmos esse tipo de estrutura de repetição, com while, para deixarmos serviços executando em servidores. Por exemplo, o serviço deve ficar executando enquanto a hora do sistema for diferente de 20 horas.

Outro comando muito utilizado para construir as estruturas de repetição é o for. A instrução for em Python difere um pouco do que ocorre em outras linguagens, como C ou Java, com as quais você pode estar acostumado

(<https://docs.python.org/3/tutorial/controlflow.html>). Em vez de sempre dar ao usuário a capacidade de definir a etapa de iteração e a condição de parada (como C), a instrução Python for itera sobre os itens de qualquer sequência, por exemplo, iterar sobre os caracteres de uma palavra, pois uma palavra é um tipo de sequência.

Observe um exemplo a seguir.

In [7]:

```
nome = "Guido"  
for c in nome:  
    print(c)
```

G  
u  
i  
d  
o

0

Ver anotações

Na entrada 7 (In [7]) construímos nossa primeira estrutura de repetição com for.

Vamos analisar a sintaxe da estrutura. Usamos o comando "for" seguido da variável de controle "c", na sequência o comando "in", que traduzindo significa "em", por fim, a sequência sobre a qual a estrutura deve iterar. Os dois pontos marcam o início do bloco que deve ser repetido.

Junto com o comando for, podemos usar a função enumerate() para retornar à posição de cada item, dentro da sequência. Considerando o exemplo dado, no qual atribuímos a variável "nome" o valor de "Guido", "G" ocupa a posição 0 na sequência, "u" ocupa a posição 1, "i" a posição 2, e assim por diante. A função enumerate() recebe como parâmetro a sequência e retorna a posição. Para que possamos capturar tanto a posição quanto o valor, vamos precisar usar duas variáveis de controle. Observe o código a seguir, veja que a variável "i" é usada para capturar a posição e a variável "c" cada caracter da palavra.

In [8]:

```
nome = "Guido"  
for i, c in enumerate(nome):  
    print(f"Posição = {i}, valor = {c}")
```

Posição = 0, valor = G  
Posição = 1, valor = u  
Posição = 2, valor = i  
Posição = 3, valor = d  
Posição = 4, valor = o

## CONTROLE DE REPETIÇÃO COM RANGE, BREAK E CONTINUE

Como já dissemos, o comando `for` em Python requer uma sequência para que ocorra a iteração. Mas e se precisarmos que a iteração ocorra em uma sequência numérica, por exemplo, de 0 a 5? Para criar uma sequência numérica de iteração em Python podemos usar a função `range()`. Observe o código a seguir.

In [9]:

```
for x in range(5):  
    print(x)
```

```
0  
1  
2  
3  
4
```

Ver anotações

No comando, "x" é a variável de controle, ou seja, a cada iteração do laço, seu valor é alterado, já a função `range()` foi utilizada para criar um "iterable" numérico (objeto iterável) para que as repetições acontecesse. A função `range()` pode ser usada de três formas distintas:

1. Método 1: passando um único argumento que representa a quantidade de vezes que o laço deve repetir;
2. Método 2: passando dois argumentos, um que representa o início das repetições e outro o limite superior (NÃO INCLUÍDO) do valor da variável de controle;
3. Método 3: Passando três argumentos, um que representa o início das repetições; outro, o limite superior (NÃO INCLUÍDO) do valor da variável de controle e um que representa o incremento. Observe as três maneiras a seguir.

A execução dos três métodos pode ser visualizada a seguir. Ao clicar em "Next", você irá para a próxima linha a ser executada do "for" para cada um dos casos. Observe que a seta verde na linha de código representa a linha em execução, e a seta vermelha representa a próxima linha a ser executada.

Além de controlar as iterações com o tamanho da sequência, outra forma de influenciar no fluxo é por meio dos comandos "break" e "continue". O comando `break` para a execução de uma estrutura de repetição, já com o comando `continue`,

conseguimos "pular" algumas execuções, dependendo de uma condição. Veja o exemplo:

In [10]:

```
# Exemplo de uso do break
disciplina = "Linguagem de programação"
for c in disciplina:
    if c == 'a':
        break
    else:
        print(c)
```

L  
i  
n  
g  
u

0

Ver anotações

No exemplo com uso do break, perceba que foram impressas as letras L i n g u, quando chegou a primeira letra a, a estrutura de repetição foi interrompida. Agora observe o exemplo com continue.

In [11]:

```
# Exemplo de uso do continue
disciplina = "Linguagem de programação"
for c in disciplina:
    if c == 'a':
        continue
    else:
        print(c)
```

L  
i  
n  
g  
u  
g  
e  
m  
  
d  
e  
  
p  
r  
o  
g  
r  
m  
ç  
ã  
o

0 Ver anotações

No exemplo com uso do continue, perceba que foram impressas todas as letras, exceto as vogais "a", pois toda vez que elas eram encontradas, o continue determina que se pule, mas que a repetição prossiga para o próximo valor. Utilize o emulador para executar os códigos e crie novas combinações para explorar.



0

[Ver anotações](#)

## EXEMPLIFICANDO

Vamos brincar um pouco com o que aprendemos. Vamos criar uma solução que procura pelas vogais "a", "e" em um texto (somente minúsculas). Toda vez que essas vogais são encontradas, devemos informar que encontramos e qual posição do texto ela está. Nossa texto será:

texto = A inserção de comentários no código do programa é uma prática normal. Em função disso, toda linguagem de programação tem alguma maneira de permitir comentários sejam inseridos nos programas. O objetivo é adicionar descrições em partes do código, seja para documentá-lo ou para adicionar uma descrição do algoritmo implementado (BANIN, 2018, p. 45).

In [12]:

```
texto = """  
A inserção de comentários no código do programa é uma prática normal.  
Em função disso, toda linguagem de programação tem alguma maneira de permitir que  
comentários sejam inseridos nos programas.  
O objetivo é adicionar descrições em partes do código, seja para documentá-lo ou  
para adicionar uma descrição do algoritmo implementado (BANIN, p. 45, 2018)."  
"""\n  
  
for i, c in enumerate(texto):  
    if c == 'a' or c == 'e':  
        print(f"Vogal '{c}' encontrada, na posição {i}")  
    else:  
        continue
```

Ver anotações

Vogal 'e' encontrada, na posição 6  
Vogal 'e' encontrada, na posição 13  
Vogal 'e' encontrada, na posição 18  
Vogal 'a' encontrada, na posição 45  
Vogal 'a' encontrada, na posição 47  
Vogal 'a' encontrada, na posição 53  
Vogal 'a' encontrada, na posição 61  
Vogal 'a' encontrada, na posição 67  
Vogal 'a' encontrada, na posição 91  
Vogal 'a' encontrada, na posição 98  
Vogal 'e' encontrada, na posição 100  
Vogal 'e' encontrada, na posição 104  
Vogal 'a' encontrada, na posição 111  
Vogal 'a' encontrada, na posição 113  
Vogal 'e' encontrada, na posição 119  
Vogal 'a' encontrada, na posição 122  
Vogal 'a' encontrada, na posição 127  
Vogal 'a' encontrada, na posição 130  
Vogal 'e' encontrada, na posição 132  
Vogal 'a' encontrada, na posição 135  
Vogal 'e' encontrada, na posição 138  
Vogal 'e' encontrada, na posição 141  
Vogal 'e' encontrada, na posição 151  
Vogal 'e' encontrada, na posição 156  
Vogal 'e' encontrada, na posição 166  
Vogal 'a' encontrada, na posição 168  
Vogal 'e' encontrada, na posição 174  
Vogal 'a' encontrada, na posição 190  
Vogal 'a' encontrada, na posição 192  
Vogal 'e' encontrada, na posição 201  
Vogal 'a' encontrada, na posição 209  
Vogal 'a' encontrada, na posição 216  
Vogal 'e' encontrada, na posição 220  
Vogal 'e' encontrada, na posição 227  
Vogal 'e' encontrada, na posição 230  
Vogal 'a' encontrada, na posição 234

0  
Ver anotações

Vogal 'e' encontrada, na posição 237  
Vogal 'e' encontrada, na posição 252  
Vogal 'a' encontrada, na posição 254  
Vogal 'a' encontrada, na posição 257  
Vogal 'a' encontrada, na posição 259  
Vogal 'e' encontrada, na posição 266  
Vogal 'a' encontrada, na posição 278  
Vogal 'a' encontrada, na posição 280  
Vogal 'a' encontrada, na posição 282  
Vogal 'a' encontrada, na posição 289  
Vogal 'a' encontrada, na posição 294  
Vogal 'e' encontrada, na posição 297  
Vogal 'a' encontrada, na posição 309  
Vogal 'e' encontrada, na posição 323  
Vogal 'e' encontrada, na posição 325  
Vogal 'a' encontrada, na posição 328

0

Ver anotações

Uma novidade nesse exemplo foi o uso das aspas triplas para que pudéssemos quebrar a linha do texto. As aspas triplas também podem ser usadas para criar comentários de várias linhas em Python. Que tal alterar a solução de busca por vogais e contar quantas vezes cada vogal foi encontrada. Lembre-se de que, quanto mais praticar, mais desenvolverá sua lógica de programação e fluência em uma determinada linguagem!



## REFERÊNCIAS E LINKS ÚTEIS

BANIN, S. L. **Python 3 - conceitos e aplicações**: uma abordagem didática. São Paulo: Érica, 2018.

GRIFFITHS, D.; BARRY, P. **Head First Programming**: A learner's guide to programming using the Python language. [S.I.]: O'Reilly Media, 2009.

MANZANO, J. A. N. G.; OLIVEIRA, J. F. de. **Algoritmos**: lógica para desenvolvimento de programação de computadores. 29. ed. São Paulo: Érica, 2019.

PYTHON SOFTWARE FOUNDATION. **More Control Flow Tools**. Disponível em: <https://docs.python.org/3/tutorial/controlflow.html>. Acesso em: 15 abr. 2020.

PYTHON SOFTWARE FOUNDATION. **Numeric Types**: int, float, complex. Disponível em: <https://docs.python.org/3/library/stdtypes.html>. Acesso em: 15 abr. 2020.

SUBSECRETARIA DE TRIBUTAÇÃO E CONTENCIOSO. **Imposto sobre a renda das pessoas físicas**. Disponível em: <http://receita.economia.gov.br/acesso-rapido/tributos/irpf-imposto-de-renda-pessoa-fisica>. Acesso em: 18 abr. 2020.

Ver anotações

## FOCO NO MERCADO DE TRABALHO

# ESTRUTURAS LÓGICAS, CONDICIONAIS E DE REPETIÇÃO EM PYTHON

0

Vanessa Cadan Scheffer

Ver anotações

## IMPLEMENTANDO UMA SOLUÇÃO

Desenvolvendo novas funcionalidades de sistema.



Fonte: Shutterstock.

## Deseja ouvir este material?

Áudio disponível no material digital.

## DESAFIO

Trabalhar em uma empresa de consultoria focada no desenvolvimento de software é desafiador, pois você pode atender diferentes clientes e atuar em diferentes projetos. Embora desafiador, sem dúvida é uma experiência enriquecedora.

Dando continuidade ao seu trabalho na empresa de consultoria de software, o cliente que fabrica peças automotivas requisitou uma nova funcionalidade para o sistema: calcular o imposto de renda a ser deduzido do salário dos colaboradores. O imposto

de renda "*Incide sobre a renda e os proventos de contribuintes residentes no País ou residentes no exterior que recebam rendimentos de fontes no Brasil. Apresenta alíquotas variáveis conforme a renda dos contribuintes, de forma que os de menor renda não sejam alcançados pela tributação.*" ([IRPF, 2020](#))

o

Você foi designado para pesquisar a tabela de valores para o imposto de renda c  
ano de 2020, pensar no algoritmo e implementar a primeira versão da solução.

Ver anotações

Quando o cliente aprovar, a versão passará para o time de produção. Nessa pri  
versão, o programa deve solicitar o salário do colaborador e então, informar qua  
valor do imposto que será deduzido do salário.

Já sabe qual ferramenta vai utilizar para implementar a solução? Onde buscar as  
informações confiáveis sobre o imposto de renda? Como implementar a solução?  
Faça um bom trabalho e mostre para seu gestor sua evolução!

## RESOLUÇÃO

Em seu novo projeto, você foi designado a implementar uma solução que envolve o  
cálculo do imposto de renda, baseado nos dados do ano de 2020. O primeiro passo é  
encontrar uma fonte confiável de informação, que nesse caso é o portal da Receita  
Federal. No endereço (<http://receita.economia.gov.br/acesso-rapido/tributos/irpf-imposto-de-renda-pessoa-fisica>) você encontra vários links a respeito desse imposto,  
e no link ([http://receita.economia.gov.br/acesso-rapido/tributos/irpf-imposto-de-renda-pessoa-fisica#calculo\\_mensal\\_IRPF](http://receita.economia.gov.br/acesso-rapido/tributos/irpf-imposto-de-renda-pessoa-fisica#calculo_mensal_IRPF)) você tem acesso às tabelas de incidência  
mensal desse imposto para cada ano. Como o ano solicitado foi 2020, a última tabela  
disponível é a de 2015 (Tabela 1.2).

Tabela 1.2 | Tabela de incidência mensal, ano 2015

Base de cálculo em reais Alíquota (%) Parcela a deduzir do IRPF em reais

Até 1.903,98	-	-
De 1.903,99 até 2.826,65	7,5	142,80
De 2.826,66 até 3.751,05	15	354,80
De 3.751,06 até 4.664,68	22,5	636,13
Acima de 4.664,68	27,5	869,36

Fonte: [http://receita.economia.gov.br/acesso-rapido/tributos/irpf-imposto-de-renda-pessoa-fisica#calculo\\_mensal\\_IRPF](http://receita.economia.gov.br/acesso-rapido/tributos/irpf-imposto-de-renda-pessoa-fisica#calculo_mensal_IRPF)

0

Ver anotações

Com a tabela de incidência mensal em mãos, é hora de escolher a ferramenta que irá implementar a solução, pensar no algoritmo e fazer a implementação. Você pode usar o Colab, uma vez que não precisa instalar e por ser online, pode compartilhar o resultado, tanto com seu gestor quanto com o cliente.

O programa deve receber um salário, com base no valor informado, e algoritmo deve verificar em qual faixa do imposto esse valor se enquadra, quando encontrar a faixa, o programa imprime o valor a ser deduzido. Pois, bem, agora vamos implementar esse algoritmo, observe o código a seguir.

In [1]:

```
salario = 0

salario = float(input("Digite o salário do colaborador: "))

if salario <= 1903.98:
    print(f"O colaborador é isento de imposto.")
elif salario <= 2826.65:
    print(f"O colaborador deve pagar R$ 142,80 de imposto.")
elif salario <= 3751.05:
    print(f"O colaborador deve pagar R$ 354,80 de imposto.")
elif salario <= 4664.68:
    print(f"O colaborador deve pagar R$ 636,13 de imposto.")
else:
    print(f"O colaborador deve pagar R$ 869,36 de imposto.")
```

0

Ver anotações

```
Digite o salário do colaborador: 2000
O colaborador deve pagar R$ 142,80 de imposto.
```

Na solução apresentada, veja que na linha 1, criamos e inicializamos a variável "salario", embora não seja obrigatório em Python é uma boa prática de programação.

Veja como fizemos o teste do valor, por que usamos somente um valor para testar? Fizemos dessa forma, porque em uma estrutura condicional encadeada, quando a primeira condição for satisfeita, as demais não são testadas. Por exemplo, considerando um colaborada que ganha R\$ 2 mil por mês, embora esse valor seja menor que 3751,05 e 4664,68, a primeira condição que é satisfeita para o caso é a da linha 6 "elif salario <= 2826.65", logo essa será executada e não as demais.

Que tal mostrar para seu gerente e cliente sua pró-atividade e, além de criar uma solução que exibe o valor do desconto, também já seja impresso o valor do salário com o desconto aplicado? Observe o código com essa nova feature!

In [2]:

```
salario = 0

salario = float(input("Digite o salário do colaborador: "))

if salario <= 1903.98:
    print(f"O colaborador é isento de imposto.")
    print(f"Salário a receber = {salario}")
elif salario <= 2826.65:
    print(f"O colaborador deve pagar R$ 142,80 de imposto.")
    print(f"Salário a receber = {salario - 142.80}")
elif salario <= 3751.05:
    print(f"O colaborador deve pagar R$ 354,80 de imposto.")
    print(f"Salário a receber = {salario - 354.80}")
elif salario <= 4664.68:
    print(f"O colaborador deve pagar R$ 636,13 de imposto.")
    print(f"Salário a receber = {salario - 636.13}")
else:
    print(f"O colaborador deve pagar R$ 869,36 de imposto.")
    print(f"Salário a receber = {salario - 869.36}")
```

Digite o salário do colaborador: 2000

O colaborador deve pagar R\$ 142,80 de imposto.

Salário a receber = 1857.2

0

Ver anotações

## DESAFIO DA INTERNET

Que tal treinar um pouco mais de programação e conhecer novas fontes de informações? Você, aluno, tem acesso à Biblioteca Virtual, um repositório de livros e artigos que pode ser acessado no endereço: (<http://biblioteca-virtual.com/>).

Na página 75 (capítulo 3) da obra: BANIN, S. L. **Python 3 - conceitos e aplicações:** uma abordagem didática. São Paulo: Érica, 2018, você encontra o exercício 3 (uso de condições mistas). Utilize o emulador a seguir, para resolver o desafio.

The screenshot shows a Jupyter Notebook interface. At the top, there's a toolbar with icons for file operations, a code editor, run, share, and remix. Below the toolbar, a file browser shows a folder named 'matrinx' containing a file 'flammation\_01.csv'. A message box says 'Escolher arquivo' and 'Nenhum arquivo escolhido'. On the right side, there's a sidebar with the text 'Ver anotações' and a small number '0'.

# NÃO PODE FALTAR

## FUNÇÕES EM PYTHON

Vanessa Cadan Scheffer

0

Ver anotações

### IMPLEMENTANDO SOLUÇÕES EM PYTHON MEDIANTE FUNÇÕES

Vamos conhecer as funções em Python e o modo como aplicá-las a fim de facilitar a leitura e a manutenção das soluções propostas.



Fonte: Shutterstock.

### Deseja ouvir este material?

Áudio disponível no material digital.

O assunto desta seção será funções. Por que precisamos aprender essa técnica de programação? Segundo Banin (2018, p. 119), "... funções constituem um elemento de fundamental importância na moderna programação de computadores, a ponto de ser possível afirmar que atualmente nenhum programa de computador é desenvolvido sem o uso desse recurso".

Para entendermos, com clareza, o que é uma função, vamos pensar na organização de um escritório de advocacia. Nesse escritório, existe um secretário que possui a função de receber os clientes e agendar horários. Também trabalham nesse escritório três advogados, que possuem a função de orientar e representar seus clientes com base nas leis. Não podemos deixar de mencionar os colaboradores possuem a função de limpar o escritório e fazer reparos. Mas o que o escritório tem em comum com nossas funções? Tudo! Citamos algumas funções que podem ser encontradas nesse ambiente de trabalho, ou seja, um conjunto de tarefas/ações associadas a um "nome". Podemos, então, resumir que uma função é uma forma de organização, usada para delimitar ou determinar quais tarefas podem ser realizadas por uma determinada divisão.

Essa ideia de organização em funções é o que se aplica na programação. Poderíamos implementar uma solução em um grande bloco de código, nesse caso, teríamos um cenário quase impossível de dar manutenção. Em vez de escrever dessa forma, podemos criar uma solução dividindo-a em funções (blocos), além de ser uma boa prática de programação, tal abordagem facilita a leitura, a manutenção e a escalabilidade da solução.

## FUNÇÕES BUILT-IN EM PYTHON

Desde que escrevemos nossa primeira linha de código nessa disciplina `>>print("hello world")`, já começamos a usar funções, pois `print()` é uma função built-in do interpretador Python. Uma função built-in é um objeto que está integrado ao núcleo do interpretador, ou seja, não precisa ser feita nenhuma instalação adicional, já está pronto para uso. O interpretador Python possui várias funções disponíveis, veja o Quadro 1.3.

Ver anotações  
1  
S

## Quadro 1.3 | Funções built-in em Python

0

Ver anotações

abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	dir()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	

Fonte: <https://docs.python.org/3/library/functions.html>

Ao observar o Quadro 1.3, podemos identificar algumas funções que já usamos:

- print(), para imprimir um valor na tela.
- enumerate(), para retornar a posição de um valor em uma sequência.
- input(), para capturar um valor digitado no teclado.
- int() e float(), para converter um valor no tipo inteiro ou float.
- range(), para criar uma sequência numérica.
- type(), para saber qual o tipo de um objeto (variável).

Ao longo da disciplina, iremos conhecer várias outras, mas certamente vale a pena você acessar a documentação <https://docs.python.org/3/library/functions.html> e explorar tais funções, aumentando cada vez mais seu repertório. Para mostrar o poder das funções e deixar um gostinho de quero mais, veja o código a seguir, com a utilização da função eval().

## In [1]:

```
a = 2
b = 1

equacao = input("Digite a fórmula geral da equação linear (a * x + b): ")
print(f"\nA entrada do usuário {equacao} é do tipo {type(equacao)}")

for x in range(5):
    y = eval(equacao)
    print(f"\nResultado da equação para x = {x} é {y}")
```

0

Ver anotações

Digite a fórmula geral da equação linear (a \* x + b): a \* x + b

A entrada do usuário a \* x + b é do tipo <class 'str'>

Resultado da equação para x = 0 é 1

Resultado da equação para x = 1 é 3

Resultado da equação para x = 2 é 5

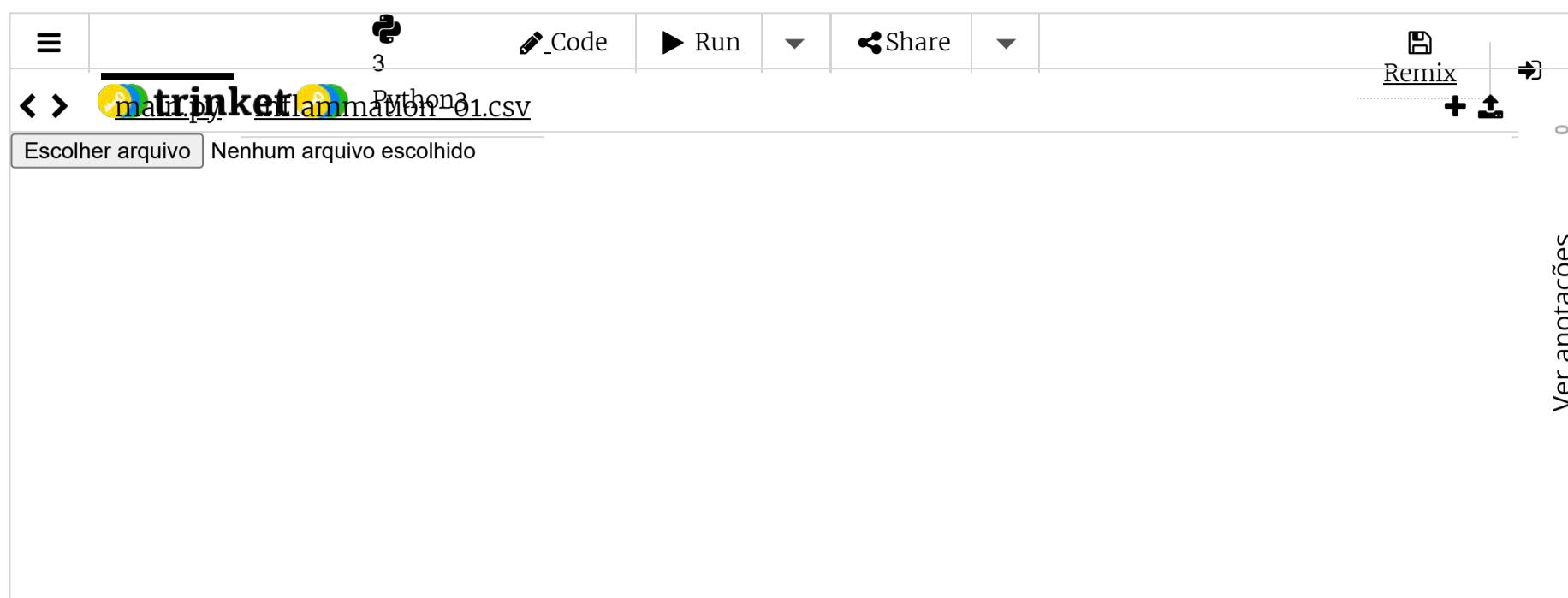
Resultado da equação para x = 3 é 7

Resultado da equação para x = 4 é 9

A função eval() usada no código recebe como entrada uma string (sequência de caracteres) digitada pelo usuário, que nesse caso é uma equação linear. Essa entrada é analisada e avaliada como uma expressão Python pela função eval(). Veja que, para cada valor de x, a fórmula é executada como uma expressão matemática (linha 8) e retorna um valor diferente.

A função eval() foi mostrada a fim de exemplificar a variedade de funcionalidades que as funções built-in possuem. Entretanto, precisamos ressaltar que eval é uma instrução que requer prudência para o uso, pois é fácil alguém externo à aplicação fazer uma "injection" de código intruso.

Utilize o emulador a seguir, para testar o código e experimentar novas funções built-in em Python.



## FUNÇÃO DEFINIDA PELO USUÁRIO

Python possui 70 funções built-in (Quadro 1.3), que facilitam o trabalho do desenvolvedor, porém cada solução é única e exige implementações específicas. Diante desse cenário, surge a necessidade de implementar nossas próprias funções, ou seja, trechos de códigos que fazem uma determinada ação e que nós, como desenvolvedores, podemos escolher o nome da função, sua entrada e sua saída.

Vamos começar a desenvolver nossas funções, entendendo a sintaxe de uma função em Python. Observe o código a seguir.

In [2]:

```
def nome_funcao():
    # bloco de comandos
```

Veja que na linha 1 da entrada 2 (In [2]), usamos o comando "def" para indicar que vamos definir uma função. Em seguida, escolhemos o nome da função "imprimir\_mensagem", veja que não possui acento, nem espaço, conforme recomenda a PEP 8 <https://www.python.org/dev/peps/pep-0008/#function-and-variable-names>: os nomes das funções devem estar em minúsculas, com as palavras separadas por underline, conforme necessário, para melhorar a legibilidade. Os nomes de variáveis seguem a mesma convenção que os nomes de funções. Nomes com mixedCase (mistura de maiúsculas e minúsculas) são permitidos apenas em contextos em que o nome já existe com o formato recomendado.

Em toda declaração de função, após especificar o nome, é preciso abrir e fechar parênteses, pois é dentro dos parênteses que os parâmetros de entrada da função devem ser definidos. Nessa versão da nossa função "imprimir\_mensagem", não existe nenhum parâmetro sendo passado como entrada. A linha 2 representa o conjunto de ações que essa função deve executar, no caso, somente imprimir uma mensagem na tela. Para que uma função execute suas ações, ela precisa ser "invocada", fazendo isso na linha 5, colocando o nome da função, acompanhada dos parênteses.

Ver anotações

Agora vamos criar uma segunda versão da nossa função "imprimir\_mensagem".

Observe o código a seguir:

In [3]:

```
def imprimir_mensagem(disciplina, curso):
    print(f"Minha primeira função em Python desenvolvida na disciplina:
{disciplina}, do curso: {curso}.")
```

```
imprimir_mensagem("Python", "ADS")
```

```
Minha primeira função em Python desenvolvida na disciplina: Python, do curso: ADS.
```

Veja na linha 1 da entrada 3 (In [3]), que agora a função recebe dois parâmetros. Esses parâmetros são variáveis locais, ou seja, são variáveis que existem somente dentro da função. Na linha 2, imprimimos uma mensagem usando as variáveis passadas como parâmetro e na linha 5, invocamos a função, passando como parâmetros dois valores do tipo string. O valor "Python" vai para o primeiro parâmetro da função e o valor "ADS" vai para o segundo parâmetro.

O que acontece se tentarmos atribuir o resultado da função "imprimir\_mensagem" em uma variável, por exemplo:

```
resultado = imprimir_mensagem("Python", "ADS")? Como a função
"imprimir_mensagem" não possui retorno, a variável "resultado" receberá "None".
```

Teste o código a seguir e veja o resultado:

In [4]:

```
def imprimir_mensagem(disciplina, curso):
    print(f"Minha primeira função em Python desenvolvida na disciplina:
{disciplina}, do curso: {curso}.")
```

```
resultado = imprimir_mensagem("Python", "ADS")
```

```
print(f"Resultado = {resultado}")
```

```
Minha primeira função em Python desenvolvida na disciplina: Python, do curso:
```

```
Resultado = None
```

0

Ver anotações

0

[Ver anotações](#)

Para que o resultado de uma função possa ser guardado em uma variável, a função precisa ter o comando "return". A instrução "return", retorna um valor de uma função. Veja a nova versão da função "imprimir\_mensagem", agora, em vez de imprimir a mensagem, ela retorna a mensagem para chamada.

In [5]:

```
def imprimir_mensagem(disciplina, curso):
    return f"Minha primeira função em Python desenvolvida na disciplina:
{disciplina}, do curso: {curso}."

mensagem = imprimir_mensagem("Python", "ADS")
print(mensagem)
```

Minha primeira função em Python desenvolvida na disciplina: Python, do curso: ADS.

Veja na linha 2 da entrada 5 (In [5]) que, em vez de imprimir a mensagem, a função retorna (return) um valor para quem a invocou. O uso do "return" depende da solução e das ações que se deseja para a função. Por exemplo, podemos criar uma função que limpa os campos de um formulário, esse trecho pode simplesmente limpar os campos e não retornar nada, mas também pode retornar um valor booleano como True, para informar que a limpeza aconteceu com sucesso. Portanto, o retorno deve ser analisado, levando em consideração o que se espera da função e como se pretende tratar o retorno, quando necessário.

## EXEMPLIFICANDO

Vamos implementar uma função que recebe uma data no formato dd/mm/aaaa e converte o mês para extenso. Então, ao se receber a data 10/05/2020, a função deverá retornar: 10 de maio de 2020. Observe a implementação a seguir.

In [6]:

o

Ver anotações

```
def converter_mes_para_extenso(data):
    mes = '''janeiro fevereiro março
              abril maio junho julho agosto
              setembro outubro novembro
              dezembro'''.split()
    d, m, a = data.split('/')
    mes_extenso = mes[int(m) - 1] # 0 mês 1, estará na posição 0!
    return f'{d} de {mes_extenso} de {a}'
```

```
print(converter_mes_para_extenso('10/05/2021'))
```

```
10 de maio de 2021
```

0

Ver anotações

- Linha 1 - Definimos o nome da função e os parâmetros que ela recebe.
- Linha 2 - Criamos uma lista com os meses, por extenso. Veja que criamos uma string e usamos a função split(), que "quebra" a string a cada espaço em branco, criando uma lista e elementos.
- Linha 6 - Usamos novamente a função split(), mas agora passando como parâmetro o caractere '/', isso quer dizer que a string será cortada sempre que ele aparecer. Nessa linha também usamos a atribuição múltipla. Ao cortar a string 'dd/mm/aaaa', temos uma lista com três elementos: ['dd', 'mm', 'aaaa'], ao usar a atribuição múltipla, cada valor da lista é guardado dentro de uma variável, na ordem em que foram declaradas. Então d = 'dd', m = 'mm', a = 'aaaa'. O número de variáveis tem que ser adequado ao tamanho da lista, senão ocorrerá um erro.
- Linha 7 - Aqui estamos fazendo a conversão do mês para extenso. Veja que buscamos na lista "mes" a posição m - 1, pois, a posição inicia em 0. Por exemplo, para o mês 5, o valor "maio", estará na quarta posição a lista "mes".
- Linha 8 - Retornamos a mensagem, agora com o mês em extenso.
- Linha 10 - Invocamos a função, passando como parâmetro a data a ser convertida.

Aproveite o emulador para testar o código e implementar outras funções.

The screenshot shows a Jupyter Notebook interface. At the top, there's a toolbar with icons for file operations, code, run, share, and remix. Below the toolbar, the title bar displays "matplabx Python3" and "Flammation\_01.csv". A dropdown menu says "Escolher arquivo" and "Nenhum arquivo escolhido". On the right side, there's a vertical sidebar with "Ver anotações" and a count of "0". The main area is currently empty, indicating no code or output has been run.

## FUNÇÕES COM PARÂMETROS DEFINIDOS E INDEFINIDOS

Sobre os argumentos que uma função pode receber, para nosso estudo, vamos classificar em seis grupos:

1. Parâmetro posicional, obrigatório, sem valor default (padrão).
2. Parâmetro posicional, obrigatório, com valor default (padrão).
3. Parâmetro nominal, obrigatório, sem valor default (padrão).
4. Parâmetro nominal, obrigatório, com valor default (padrão).
5. Parâmetro posicional e não obrigatório (**args**).
6. Parâmetro nominal e não obrigatório (**kwargs**).

No grupo 1, temos os parâmetros que vão depender da ordem em que forem passados, por isso são chamados de posicionais (a posição influencia o resultado). Os parâmetros desse grupo são obrigatórios, ou seja, tentar um invocar a função, sem passar os parâmetros, acarreta um erro. Além disso, os parâmetros não possuem valor default. Observe a função "somar" a seguir.

In [7]:

```
def somar(a, b):  
    return a + b  
  
r = somar(2)  
print(r)
```

```
-----  
TypeError                                         Traceback (most recent call last)  
<ipython-input-7-4456bbb97a27> in <module>  
      2     return a + b  
      3  
----> 4 r = somar(2)  
      5 print(r)
```

```
TypeError: somar() missing 1 required positional argument: 'b'
```

0

Ver anotações

A função "somar" na entrada 7 (In [7]) foi definida de modo a receber dois parâmetros, porém na linha 4, quando ela foi invocada, somente um parâmetro foi passado, o que resultou no erro "missing 1 required positional argument", que traduzindo quer dizer: "falta 1 argumento posicional obrigatório". Para que a função execute sem problema, ela deve ser invocada passando os dois argumentos, por exemplo: `r = somar(2, 3)`.

Ver anotações

No grupo 2, também temos os parâmetros posicionais e obrigatórios, porém vamos definir um valor default (padrão), assim, quando a função for invocada, caso nenhum valor seja passado, o valor default é utilizado. Observe a função "calcular\_desconto" a seguir.

In [8]:

```
def calcular_desconto(valor, desconto=0): # O parâmetro desconto possui zero valor default
    valor_com_desconto = valor - (valor * desconto)
    return valor_com_desconto

valor1 = calcular_desconto(100) # Não aplicar nenhum desconto
valor2 = calcular_desconto(100, 0.25) # Aplicar desconto de 25%

print(f"\nPrimeiro valor a ser pago = {valor1}")
print(f"\nSegundo valor a ser pago = {valor2}")
```

Primeiro valor a ser pago = 100

Segundo valor a ser pago = 75.0

A função "calcular\_desconto" na entrada 8 (In [8]) foi definida de modo a receber dois parâmetros: "valor" e "desconto". O parâmetro "valor" não possui valor default, já o parâmetro "desconto" possui zero como valor default, ou seja, se a função for invocada e o segundo parâmetro não for passado, será usado o valor padrão definido para o argumento. Veja que, na linha 5, a função é invocada, sem passar o segundo argumento e não causa erro, pois existe o valor default. Já na linha 6 a função é invocada passando tanto o valor quanto o desconto a ser aplicado.

A obrigatoriedade do argumento, quando não atendida, pode resultar em um erro, conforme vimos na entrada 7 (In [7]). Porém, para o conceito de parâmetros posicionais não existe nenhum erro de interpretação associado, ou seja, o interpretador não irá informar o erro, mas pode haver um erro de lógica. Observe o código a seguir:

In [9]:

```
def cadastrar_pessoa(nome, idade, cidade):
    print("\nDados a serem cadastrados:")
    print(f"Nome: {nome}")
    print(f"Idade: {idade}")
    print(f"Cidade: {cidade}")

cadastrar_pessoa("João", 23, "São Paulo")
cadastrar_pessoa("São Paulo", "João", 23)
```

Dados a serem cadastrados:

Nome: João

Idade: 23

Cidade: São Paulo

Dados a serem cadastrados:

Nome: São Paulo

Idade: João

Cidade: 23

A função "cadastrar\_pessoa" na entrada 9 (In [9]) foi definida de modo a receber três parâmetros: "nome", "idade" e "cidade". Observe a chamada da função da linha 8, foram passados os argumentos: "João", 23, "São Paulo". O primeiro valor, "João", foi atribuído ao primeiro parâmetro na função, "nome". O segundo valor, 23, foi atribuído ao segundo parâmetro na função, "idade". O terceiro valor, "São Paulo", foi atribuído ao terceiro parâmetro na função, "cidade", portanto o resultado foi exatamente o que esperávamos. Agora observe a chamada na linha 9, foram passados os argumentos: "São Paulo", "João", 23. O primeiro valor, "São Paulo", foi

Ver anotações

atribuído ao primeiro parâmetro na função, "nome". O segundo valor, "João", foi atribuído ao segundo parâmetro na função, "idade". O terceiro valor, 23, foi atribuído ao terceiro parâmetro na função "cidade", tais atribuições implicam um erro lógico, pois os dados não foram atribuídos às variáveis corretas.

Com o exemplo da função "cadastrar\_pessoa", fica claro como a posição dos argumentos, na hora de chamar a função, deve ser conhecida e respeitada, pois passagem dos valores na posição incorreta pode acarretar erros lógicos.

Ver anotações

O grupo de parâmetros 3 é caracterizado por ter parâmetros nominais, ou seja, ~~ágora~~ não mais importa a posição dos parâmetros, pois eles serão identificados pelo nome, os parâmetros são obrigatórios, ou seja, na chamada da função é obrigatório passar todos os valores e sem valor default (padrão). Observe a função "converter\_maiuscula".

In [10]:

```
def converter_maiuscula(texto, flag_maiuscula):
    if flag_maiuscula:
        return texto.upper()
    else:
        return texto.lower()

texto = converter_maiuscula(flag_maiuscula=True, texto="João") # Passagem nominal
de parâmetros
print(texto)
```

JOÃO

A função "converter\_maiuscula" na entrada 10 (In [10]) foi definida de modo a receber dois parâmetros: "texto" e "flag\_maiuscula". Caso "flag\_maiuscula" seja True, a função deve converter o texto recebido em letras maiúsculas, com a função built-in upper(), caso contrário, em minúsculas, com a função built-in lower(). Como a função "converter\_maiuscula" não possui valores default para os parâmetros, então a função deve ser invocada passando ambos valores. Agora observe a chamada na linha 8, primeiro foi passado o valor da flag\_maiuscula e depois o texto. Por que não houve

um erro lógico? Isso acontece porque a chamada foi feita de modo nominal, ou seja, atribuindo os valores às variáveis da função e, nesse caso, a atribuição não é feita de modo posicional.

O grupo de funções da categoria 4 é similar ao grupo 3: parâmetro nominal, obrigatório, mas nesse grupo os parâmetros podem possuir valor default (padrão).

Observe a função "converter\_minuscula" a seguir.

In [11]:

```
def converter_minuscula(texto, flag_minuscula=True): # O parâmetro flag_minuscula
    possui True como valor default
    if flag_minuscula:
        return texto.lower()
    else:
        return texto.upper()

texto1 = converter_minuscula(flag_minuscula=True, texto="LINGUAGEM de Programação")
texto2 = converter_minuscula(texto="LINGUAGEM de Programação")

print(f"\nTexto 1 = {texto1}")
print(f"\nTexto 2 = {texto2}")
```

Texto 1 = linguagem de programação

Texto 2 = linguagem de programação

A função "converter\_minuscula" na entrada 11 (In [11]) foi definida de modo a receber dois parâmetros, porém um deles possui valor default. O parâmetro flag\_minuscula, caso não seja passado na chamada da função, receberá o valor True. Veja a chamada da linha 8, passamos ambos os parâmetros, mas na chamada da linha 9, passamos somente o texto. Para ambas as chamadas o resultado foi o mesmo, devido o valor default atribuído na função. Se não quiséssemos o comportamento default, aí sim precisaríamos passar o parâmetro, por exemplo: texto = converter\_minuscula(flag\_minuscula=False, texto="LINGUAGEM de Programação").

0

Ver anotações

Até o momento, para todas as funções que criamos, sabemos exatamente o número de parâmetros que ela recebe. Mas existem casos em que esses parâmetros podem ser arbitrários, ou seja, a função poderá receber um número diferente de parâmetros a cada invocação. Esse cenário é o que caracteriza os grupos 5 e 6 de funções que vamos estudar.

No grupo 5, temos parâmetros posicionais indefinidos, ou seja, a passagem de valores é feita de modo posicional, porém a quantidade não é conhecida. Observe a função "obter\_parametros" a seguir.

In [12]:

```
def imprimir_parametros(*args):
    qtde_parametros = len(args)
    print(f"Quantidade de parâmetros = {qtde_parametros}")

    for i, valor in enumerate(args):
        print(f"Posição = {i}, valor = {valor}")

print("\nChamada 1")
imprimir_parametros("São Paulo", 10, 23.78, "João")

print("\nChamada 2")
imprimir_parametros(10, "São Paulo")
```

Chamada 1

```
Quantidade de parâmetros = 4
Posição = 0, valor = São Paulo
Posição = 1, valor = 10
Posição = 2, valor = 23.78
Posição = 3, valor = João
```

Chamada 2

```
Quantidade de parâmetros = 2
Posição = 0, valor = 10
Posição = 1, valor = São Paulo
```

Ver anotações

A função "imprimir\_parametros" na entrada 12 (ln [12]) foi definida de modo a obter parâmetros arbitrários. Tal construção é feita, passando como parâmetro o \*args. O parâmetro não precisa ser chamado de args, mas é uma boa prática. Já o asterisco antes do parâmetro é obrigatório. Na linha 2, usamos a função built-in len() (leng →) para saber a quantidade de parâmetros que foram passados. Como se trata de parâmetros posicionais não definidos, conseguimos acessar a posição e o valor c argumento, usando a estrutura de repetição for e a função enumerate(). Agora observe as chamadas feitas nas linhas 10 e 13, cada uma com uma quantidade diferente de argumentos, mas veja na saída que os argumentos seguem a ordem posicional, ou seja, o primeiro vai para a posição 0, o segundo para a 1 e assim por diante.

A seguir você pode testar o passo a passo de execução do código. Clique em "next" para visualizar a execução de cada linha de código.

Python 3.6

```
→ 1 def imprimir_parametros(*args):
 2     qtde_parametros = len(args)
 3     print(f"Quantidade de parâmetros
 4
 5     for i, valor in enumerate(args):
 6         print(f"Posição = {i}, valor
 7
 8
 9     print("\nChamada 1")
10    imprimir_parametros("São Paulo", 10,
11
12    print("\nChamada 2")
13    imprimir_parametros(10, "São Paulo")
```

Print output (drag lower right corner to resize)

Frames Objects

Edit this code

line that just executed

next line to execute

< Prev Next >

Step 1 of 27

Visualized with [pythontutor.com](#)

[Move and hide objects](#)

O último grupo de funções possui parâmetro nominal e não obrigatório. O mecanismo é parecido com as do grupo 5, porém agora a passagem é feita de modo nominal e não posicional, o que nos permite acessar tanto o valor do parâmetro

Ver anotações

quanto o nome da variável que o armazena. Veja a função "imprimir\_parametros" a seguir:

In [13]:

```
def imprimir_parametros(**kwargs):  
    print(f"Tipo de objeto recebido = {type(kwargs)}\n")  
    qtde_parametros = len(kwargs)  
    print(f"Quantidade de parâmetros = {qtde_parametros}")  
  
    for chave, valor in kwargs.items():  
        print(f"variável = {chave}, valor = {valor}")  
  
print("\nChamada 1")  
imprimir_parametros(cidade="São Paulo", idade=33, nome="João")  
  
print("\nChamada 2")  
imprimir_parametros(desconto=10, valor=100)
```

Chamada 1

Tipo de objeto recebido = <class 'dict'>

Quantidade de parâmetros = 3

variável = cidade, valor = São Paulo

variável = idade, valor = 33

variável = nome, valor = João

Chamada 2

Tipo de objeto recebido = <class 'dict'>

Quantidade de parâmetros = 2

variável = desconto, valor = 10

variável = valor, valor = 100

A função "imprimir\_parametros" na entrada 13 (In [13]) foi definida de modo a obter parâmetros nominais arbitrários. Tal construção é feita, passando como parâmetro o \*\*kwargs. O parâmetro não precisa ser chamado de kwargs, mas é uma boa prática.

Já os dois asteriscos antes do parâmetro é obrigatório. Na linha 2, estamos imprimindo o tipo de objeto recebido, você pode ver na saída que é um dict (dicionário), o qual estudaremos nas próximas aulas. Na linha 3, usamos a função built-in len() (length) para saber a quantidade de parâmetros que foram passados. Como se trata de parâmetros nominais não definidos, conseguimos acessar o nome da variável em que estão atribuídos o valor e o próprio valor do argumento, usar a estrutura de repetição "for" e a função items() na linha 17. A função items não é inline, ela pertence aos objetos do tipo dicionário, por isso a chamada é feita como "kwargs.items()" (ou seja, objeto.função). Agora observe as chamadas feitas nas linhas 11 e 14, cada uma com uma quantidade diferente de argumentos, mas veja na saída que os argumentos estão associados ao nome da variável que foi passado.

Utilize o emulador a seguir para testar os códigos e criar novas funções para explorar as diferentes formas de construir uma função e passar parâmetros.

A screenshot of the Trinket Python 3 environment. The interface includes a toolbar with code, run, share, and remix buttons. Below the toolbar, there's a file navigation bar with back, forward, and upload buttons, and a dropdown menu for file operations. The main workspace shows a code editor with the following content:

```
< > trinket Python 01.csv
```

The code editor has a placeholder text "Escolher arquivo" and "Nenhum arquivo escolhido".

## FUNÇÕES ANÔNIMAS EM PYTHON

Já que estamos falando sobre funções, não podemos deixar de mencionar um poderoso recurso da linguagem Python: a expressão "lambda". Expressões lambda (às vezes chamadas de formas lambda) são usadas para criar funções anônimas (<https://docs.python.org/3/reference/expressions.html#lambda>). Uma função anônima é uma função que não é construída com o "def" e, por isso, não possui nome. Esse tipo de construção é útil, quando a função faz somente uma ação e é usada uma única vez. Observe o código a seguir:

In [14]:

```
(lambda x: x + 1)(x=3)
```

Ver anotações

Out[ ]:

```
4
```

Na entrada 14 (In [14]), criamos uma função que recebe como parâmetro um valor e retorna esse valor somado a 1. Para criar essa função anônima, usamos a palavra reservada "lambda" passando como parâmetro "x". O dois pontos é o que separa a definição da função anônima da sua ação, veja que após os dois pontos, é feito o cálculo matemático  $x + 1$ . Na frente da função, já a invocamos passando como parâmetro o valor  $x=3$ , veja que o resultado é portanto 4.

Ver anotações

A função anônima pode ser construída para receber mais de um parâmetro. Observe o código a seguir:

In [15]:

```
(lambda x, y: x + y)(x=3, y=2)
```

Out[ ]:

```
5
```

Na entrada 15 (In [15]), criamos uma função anônima que recebe como parâmetro dois valores e retorna a soma deles. Para criar essa função anônima, usamos a palavra reservada "lambda" passando como parâmetro "x, y". Após os dois pontos, é feito o cálculo matemático  $x + y$ . Na frente da função, já a invocamos passando como parâmetro o valor  $x=3$  e  $y=2$ , veja que o resultado é portanto 5.

A linguagem Python, nos permite atribuir a uma variável uma função anônima, dessa forma, para invocar a função, fazemos a chamada da variável. Observe o código a seguir:

In [16]:

```
somar = lambda x, y: x + y
somar(x=5, y=3)
```

Out[ ]:

```
8
```

Na entrada 16 (In [16]), criamos uma função anônima que recebe como parâmetro dois valores e retorna a soma deles, essa função foi atribuída a uma variável chamada "somar". Veja que na linha 2, fazemos a chamada da função através do nome da variável, passando os parâmetros que ela requer.

Na próxima aula, vamos aprender sobre outros tipos de dados em Python e voltaremos a falar mais sobre as funções lambda aplicadas nessas estruturas.

A screenshot of a Jupyter Notebook interface. At the top, there's a toolbar with icons for file operations, code editing, running cells, sharing, and remixing. Below the toolbar, the title bar shows 'trinket' and 'Python 01.csv'. A dropdown menu says 'Escolher arquivo' and 'Nenhum arquivo escolhido'. The main area is a large white space for code input.

0

Ver anotações

## REFERÊNCIAS E LINKS ÚTEIS

BANIN, S. L. **Python 3 - conceitos e aplicações**: uma abordagem didática. São Paulo: Érica, 2018.

MANZANO, J. A. N. G; OLIVEIRA, J. F. de. **Estudo Dirigido de Algoritmos**. 15. ed. São Paulo: Érica, 2012.

PYTHON SOFTWARE FOUNDATION. **Built-in Functions**. Disponível em:

<https://docs.python.org/3/library/functions.html>. Acesso em: 20 abr. 2020.

PYTHON SOFTWARE FOUNDATION. **Function and Variable Names**. Disponível em:

<https://www.python.org/dev/peps/pep-0008/#function-and-variable-names>. Acesso em: 20 abr. 2020.

PYTHON SOFTWARE FOUNDATION. **Lambdas**. Disponível em:

<https://docs.python.org/3/reference/expressions.html#lambda>. Acesso em: 20 abr. 2020.

# FOCO NO MERCADO DE TRABALHO

## FUNÇÕES EM PYTHON

Vanessa Cadan Scheffer

0

Ver anotações

### IMPLEMENTANDO SOLUÇÕES POR MEIO DE FUNÇÕES

Como funções em Python são capazes de lidar com vários tipos de dinâmicas e implementar soluções.



Fonte: Shutterstock.

### Deseja ouvir este material?

Áudio disponível no material digital.

## DESAFIO

O desenvolvimento de um software pode envolver diversas pessoas e até mesmo diferentes equipes. Orquestrar todos os envolvidos com as etapas do projeto é um desafio em qualquer empresa. Para ajudar nessa missão, a engenharia de software fornece diversas ferramentas, que especificam padrões de projeto e de trabalho. A metodologia ágil é uma dessas ferramentas, que habilita a entrega de partes do software para o cliente, por exemplo, em cada sprint (intervalo de tempo, que

normalmente dura 15 dias) uma nova funcionalidade ou melhoria é entregue. Para conseguir essa agilidade na entrega, pessoas com diferentes expertise (conhecimento) são colocadas para trabalhar juntas, por exemplo, um design trabalha no layout da interface gráfica, um desenvolvedor trabalha nas funcionalidades front-end, enquanto outro trabalha no back-end e podemos ainda ter um DBA (Data Base Administrator) cuidando das conexões com o banco de dados.

Dando continuidade ao seu trabalho na empresa de consultora de software, o cliente que fabrica peças automotivas requisitou uma nova funcionalidade para o sistema de vendas. O cliente precisa calcular o total de vendas. Como seus negócios estão expandindo, o cliente solicitou que o sistema seja capaz de receber valores em reais, dólar ou euro, e que seja feita a conversão para o valor em reais. Como essa entrega demanda a construção de uma interface gráfica, além da comunicação com banco de dados, dentre outros requisitos técnicos, você foi alocado em uma equipe ágil para criar a função que fará o cálculo do valor.

Ver anotações

Após uma primeira reunião, a equipe fez um levantamento de requisitos e concluiu que a função a ser construída precisa considerar os seguintes itens:

- O valor do produto (obrigatório).
- A quantidade do produto (obrigatório).
- A moeda em que deve ser feito o cálculo (obrigatório, sendo padrão o real).
- A porcentagem do desconto que será concedida na compra (opcional).
- A porcentagem de acréscimo, que depende da forma de pagamento (opcional).

Uma conta não pode ter desconto e acréscimo ao mesmo tempo. Nessa primeira versão, você deve considerar o valor do dólar em R\$ 5,00 e do euro 5,70. Ainda não foram definidos os detalhes de como os dados serão capturados e tratados após serem digitados e submetidos. Porém, você deve entregar a versão inicial da função, para que a equipe comece a fazer os primeiros testes.

Qual nome você dará para a função? Como você especificará os parâmetros que a função pode receber? A função retornará um valor ou somente imprimirá na tela o resultado? Hora de colocar a mão na massa e implementar a solução.

## RESOLUÇÃO

Em seu novo projeto, você foi designado a implementar uma solução que envolve o cálculo de uma compra. Para esse cálculo existem parâmetros que são obrigatórios e outros opcionais, portanto a função deve ser capaz de lidar com esse tipo de dinâmica. Observe uma possível implementação a seguir.

In [1]:

```
def calcular_valor(valor_prod, qtde, moeda="real", desconto=None, acrescimo=None):
    v_bruto = valor_prod * qtde

    if desconto:
        v_liquido = v_bruto - (v_bruto * (desconto / 100))
    elif acrescimo:
        v_liquido = v_bruto + (v_bruto * (acrescimo / 100))
    else:
        v_liquido = v_bruto

    if moeda == 'real':
        return v_liquido
    elif moeda == 'dolar':
        return v_liquido * 5
    elif moeda == 'euro':
        return v_liquido * 5.7
    else:
        print("Moeda não cadastrada!")
        return 0

valor_a_pagar = calcular_valor(valor_prod=32, qtde=5, desconto=5)
print(f'O valor final da conta é {valor_a_pagar}')
```

O valor final da conta é 152.0

Sobre a solução proposta, observe o nome da função "calcular\_valor", veja que estamos seguindo as recomendações de nomenclatura, usando somente letras em minúsculo e com o underline separando as palavras. Outro detalhe é a utilização do

Ver anotações

verbo no infinitivo "calcular", toda função executa ações, por isso, é uma boa prática escolher verbos infinitos.

A função foi desenvolvida de modo a receber cinco parâmetros, sendo três deles obrigatórios e dois opcionais. Nessa implementação, para construir os parâmetros opcionais atribuímos o valor `None` às variáveis, nesse caso, como tem um valor padrão, mesmo sendo `None`, a função pode ser invocada sem a passagem desse parâmetros.

0

Ver anotações

Usamos as estruturas condicionais (`if`) para verificar se foram passados valores para desconto ou acréscimo, caso tenha valores, serão diferentes de `None` e, então, os devidos cálculos são realizados. Após os cálculos de desconto, é feito o teste para saber qual moeda foi usada na compra e fazer a conversão para real.

Muitas vezes, uma solução pode ser implementada de diferentes formas. Observe no código a seguir uma outra implementação, usando o `**kwargs` para os argumentos opcionais. Nesse caso, um dicionário é recebido e precisa ter o valor extraído. Na próxima aula, você aprenderá esse tipo de objeto.

In [2]:

```
def calcular_valor(valor_prod, qtde, moeda="real", **kwargs):
    v_bruto = valor_prod * qtde

    if 'desconto' in kwargs:
        desconto = kwargs['desconto']
        v_liquido = v_bruto - (v_bruto * (desconto / 100))

    elif 'acrescimo' in kwargs:
        acrescimo = kwargs['acrescimo']
        v_liquido = v_bruto + (v_bruto * (acrescimo / 100))

    else:
        v_liquido = v_bruto

    if moeda == 'real':
        return v_liquido
    elif moeda == 'dolar':
        return v_liquido * 5
    elif moeda == 'euro':
        return v_liquido * 5.7
    else:
        print("Moeda não cadastrada!")
        return 0

valor_a_pagar = calcular_valor(valor_prod=32, qtde=5, desconto=5)
print(f'O valor final da conta é {valor_a_pagar}')
```

O valor final da conta é 152.0

0

Ver anotações

## DESAFIO DA INTERNET

Ganhar habilidade em programação exige estudo e treino (muito treino). Acesse a biblioteca virtual no endereço <http://biblioteca-virtual.com/> e busque pelo livro: MANZANO, J. A. N. G; OLIVEIRA, J. F. de. **Estudo Dirigido de Algoritmos**. 15. ed. São Paulo: Érica, 2012. Na página 209, do capítulo 12 da referida obra: **Sub-rotinas do tipo função**, você encontrará o exemplo 3. que te desafia a criar uma função

"calculadora". Dependendo do parâmetro que a função receber, uma operação deve ser feita. O livro discute a solução usando fluxograma e português estruturado, utilize o emulador e faça a implementação usando a linguagem Python.

The screenshot shows the Trinket web-based Python code editor. At the top, there's a toolbar with icons for file operations (three horizontal lines, save, run, share, remix), a code editor icon (Python logo), and a file count (3). Below the toolbar, the title bar displays "trinket" and "Python3". A dropdown menu shows "Python3" and "Python3\_01.csv". A "Code" button is present. To the right of the title bar are "Run" and "Share" buttons with dropdown menus. On the far right, there's a "Remix" button and a user icon. The main workspace is empty, showing a placeholder message "Nenhum arquivo escolhido". On the left, there's a sidebar with a "Escolher arquivo" button. On the right side of the page, there's a vertical sidebar with the text "Ver anotações" and a small number "0".

NÃO PODE FALTAR

## ESTRUTURAS DE DADOS EM PYTHON

Vanessa Cadan Scheffer

0

Ver anotações

### COMO DEFINIMOS ESTRUTURAS DE DADOS EM PYTHON?

Em Python existem objetos em que podemos armazenar mais de um valor, aos quais damos o nome de estruturas de dados.



Fonte: Shutterstock.

### Deseja ouvir este material?

Áudio disponível no material digital.

## INTRODUÇÃO

"Todos os dados em um programa Python são representados por objetos ou pela relação entre objetos." (PSF, p. 1, 2020a). Tudo em Python é um objeto. Mas, afinal, o que são objetos? Embora tenhamos uma seção inteira dedicada a esse tema, por

ora basta entendermos que um objeto é uma estrutura que possui certas características e ações.

Já conhecemos alguns tipos de objetos em Python, tais como o int (inteiro), o str (string), o float (ponto flutuante), tipos que nos lembram das variáveis primitivas de outras linguagens, como C ou Java.

Um objeto pode ser mais complexo que um tipo primitivo. Por exemplo, o tipo primitivo int, na linguagem C, ocupa no máximo 4 bytes (32 bits) e pode armazenar valores entre -2.147.483.648 e 2.147.483.647 (<https://bit.ly/2NRspI9>). Já o objeto do tipo int, na linguagem Python, não possui um limite definido, uma vez que fica limitado apenas à memória RAM (random access memory) disponível no ambiente (<https://bit.ly/3g6S0J1>).

Outro exemplo interessante é a classe str (string). Em linguagem como a C, uma string é um vetor do tipo primitivo *char*. Por sua vez, em Python, esse objeto, além de tamanho ilimitado, possui vários métodos para manipulação de textos, como o split(), o replace(), dentre outros.

O tipo do objeto determina os valores que ele pode armazenar e as operações que podem ser feitas com tal estrutura. Nesta seção, portanto, nosso objetivo é aprender novos tipos de objetos suportados pela linguagem Python. Vamos, mais especificamente, aprender sobre os tipos de estruturas de dados: listas, tuplas, conjuntos, dicionário e matriz.

## ESTRUTURAS DE DADOS EM PYTHON

Em Python, existem objetos que são usados para armazenar mais de um valor, também chamados de estruturas de dados. Cada objeto é capaz de armazenar um tipo de estrutura de dados, particularidade essa que habilita funções diferentes para cada tipo. Portanto, a escolha da estrutura depende do problema a ser resolvido.

Ver anotações

resolvido. Para nosso estudo, vamos dividir as estruturas de dados em objetos do tipo *sequência*, do tipo *set*, do tipo *mapping* e do tipo *array NumPy*. Cada grupo pode possuir mais de um objeto. Vamos estudar esses objetos na seguinte ordem:

1. Objetos do tipo *sequência*: texto, listas e tuplas.
2. Objetos do tipo *set* (conjunto).
3. Objetos do tipo *mapping* (dicionário).
4. Objetos do tipo *array NumPy*.

0

Ver anotações

## OBJETOS DO TIPO SEQUÊNCIA

Os objetos do tipo *sequência* são estruturas de dados capazes de armazenar mais de um valor. Essas estruturas de dados representam sequências finitas indexadas por números não negativos. O primeiro elemento de uma sequência ocupa o índice 0; o segundo, 1; o último elemento, a posição  $n - 1$ , em que  $n$  é capacidade de armazenamento da sequência. As estruturas de dados desse grupo possuem algumas operações em comum, apresentadas no Quadro 2.1.

Quadro 2.1 | Operações em comum dos objetos do tipo sequência

Operação	Resultado	Observação
x in s	True caso um item de s seja igual a x, senão False	True caso um item de s seja igual a x, senão False
s + t	Concatenação de s e t	Concatena (junta) duas sequências
n * s	Adiciona s a si mesmo n vezes	Multiplica a sequência n vezes
s[i]	Acessa o valor guardado na posição i da sequência	O primeiro valor ocupa a posição 0
s[i:j]	Acessa os valores da posição i até j	O valor j não está incluído
s[i:j:k]	Acessa os valores da posição i até j, com passo k	O valor j não está incluído
len(s)	Comprimento de s	Função built-in usada para saber o tamanho da sequência
min(s)	Menor valor de s	Função built-in usada para saber o menor valor da sequência
max(s)	Maior valor de s	Função built-in usada para saber o maior valor da sequência
s.count(x)	Número total de ocorrência de x em s	Conta quantas vezes x foi encontrado na sequência

Fonte: adaptado de PSF (2020c).

## SEQUÊNCIA DE CARACTERES

Um texto é um objeto da classe *str* (strings), que é um tipo de sequência. Os objetos da classe *str* possuem todas as operações apresentadas no Quadro 2.1, mas são objetos imutáveis, razão pela qual não é possível atribuir um novo valor a uma posição específica. Vamos testar algumas operações. Observe o código a seguir.

0

Ver anotações

## In [1]:

```
texto = "Aprendendo Python na disciplina de linguagem de programação."  
  
print(f"Tamanho do texto = {len(texto)}")  
print(f"Python in texto = {'Python' in texto}")  
print(f"Quantidade de y no texto = {texto.count('y')}")  
print(f"As 5 primeiras letras são: {texto[0:6]}")
```

```
Tamanho do texto = 60  
Python in texto = True  
Quantidade de y no texto = 1  
As 5 primeiras letras são: Aprend
```

0

Ver anotações

Na entrada 1, usamos algumas operações das sequências. A operação `len()` permite saber o tamanho da sequência. O operador '`in`', por sua vez, permite saber se um determinado valor está ou não na sequência. O operador `count` permite contar a quantidade de ocorrências de um valor. E a notação com colchetes permite fatiar a sequência, exibindo somente partes dela. Na linha 6, pedimos para exibir da posição 0 até a 5, pois o valor 6 não é incluído.

Além das operações disponíveis no Quadro 2.1, a classe `str` possui vários outros métodos. A lista completa das funções para esse objeto pode ser encontrada na documentação oficial (PSF, 2020c). Podemos usar a função `lower()` para tornar um objeto `str` com letras minúsculas, ou, então, a função `upper()`, que transforma para maiúsculo. A função `replace()` pode ser usada para substituir um caractere por outro. Observe o código a seguir.

## In [2]:

```
texto = "Aprendendo Python na disciplina de linguagem de programação."  
  
print(texto.upper())  
print(texto.replace("i", 'XX'))
```

## APRENDENDO PYTHON NA DISCIPLINA DE LINGUAGEM DE PROGRAMAÇÃO.

Aprendendo Python na disciplina de linguagem de programação.

Vamos falar agora sobre a função `split()`, usada para "cortar" um texto e transformá-lo em uma lista. Essa função pode ser usada sem nenhum parâmetro: `texto.split()`. Nesse caso, a string será cortada a cada espaço em branco que for encontrado. Caso seja passado um parâmetro: `texto.split(",")`, então o corte será feito no parâmetro especificado. Observe o código a seguir.

0

Ver anotações

In [3]:

```
texto = "Aprendendo Python na disciplina de linguagem de programação."
print(f"texto = {texto}")
print(f"Tamanho do texto = {len(texto)}\n")
```

```
palavras = texto.split()
print(f"palavras = {palavras}")
print(f"Tamanho de palavras = {len(palavras)}")
```

```
texto = Aprendendo Python na disciplina de linguagem de programação.
```

```
Tamanho do texto = 60
```

```
palavras = ['Aprendendo', 'Python', 'na', 'disciplina', 'de', 'linguagem', 'de',
'programação.']")
Tamanho de palavras = 8
```

Ver anotações

Na linha 5 da entrada 3 (In [3]), usamos a função `split()` para cortar o texto e guardamos o resultado em uma variável chamada "palavras". Veja no resultado que o texto tem tamanho 60, ou seja, possui uma sequência de 60 caracteres. Já o tamanho da variável "palavras" é 8, pois, ao cortar o texto, criamos uma lista com as palavras (em breve estudaremos as listas). A função `split()`, usada dessa forma, corta um texto em cada espaço em branco.

#### EXEMPLIFICANDO

Vamos usar tudo que aprendemos até o momento para fazer um exercício de análise de texto. Dado um texto sobre strings em Python, queremos saber quantas vezes o autor menciona a palavra *string* ou *strings*. Esse tipo de raciocínio é a base para uma área de pesquisa e aplicação dedicada a criar algoritmos e técnicas para fazer análise de sentimentos em textos. Veja no código a seguir, como é simples fazer essa contagem, usando as operações que as listas possuem.

## In [4]:

```
texto = """Operadores de String

Python oferece operadores para processar texto (ou seja, valores de
string).

Assim como os números, as strings podem ser comparadas usando operadores
de comparação:

==, !=, <, > e assim por diante.

O operador ==, por exemplo, retorna True se as strings nos dois lados do
operador tiverem o mesmo valor (Perkovic, p. 23, 2016).

"""

print(f"Tamanho do texto = {len(texto)}")

texto = texto.lower()

texto = texto.replace(",","").replace(".","").replace("(", ","
").replace(")", "").replace("\n", " ")

lista_palavras = texto.split()

print(f"Tamanho da lista de palavras = {len(lista_palavras)}")

total = lista_palavras.count("string") + lista_palavras.count("strings")

print(f"Quantidade de vezes que string ou strings aparecem = {total}")
```

```
Tamanho do texto = 348
```

```
Tamanho da lista de palavras = 58
```

```
Quantidade de vezes que string ou strings aparecem = 4
```

Na entrada 4 (In [4]), começamos criando uma variável chamada "texto" que recebe uma citação do livro: PERKOVIC, Ljubomir. **Introdução à computação usando Python**: um foco no desenvolvimento de aplicações. Rio de Janeiro: LTC, 2016 (disponível na biblioteca virtual).

Na linha 8, aplicamos a função *lower()* a essa string para que todo o texto fique com letras minúsculas e guardamos o resultado dessa transformação, dentro da própria variável, sobrescrevendo, assim, o texto original.

0

Ver anotações

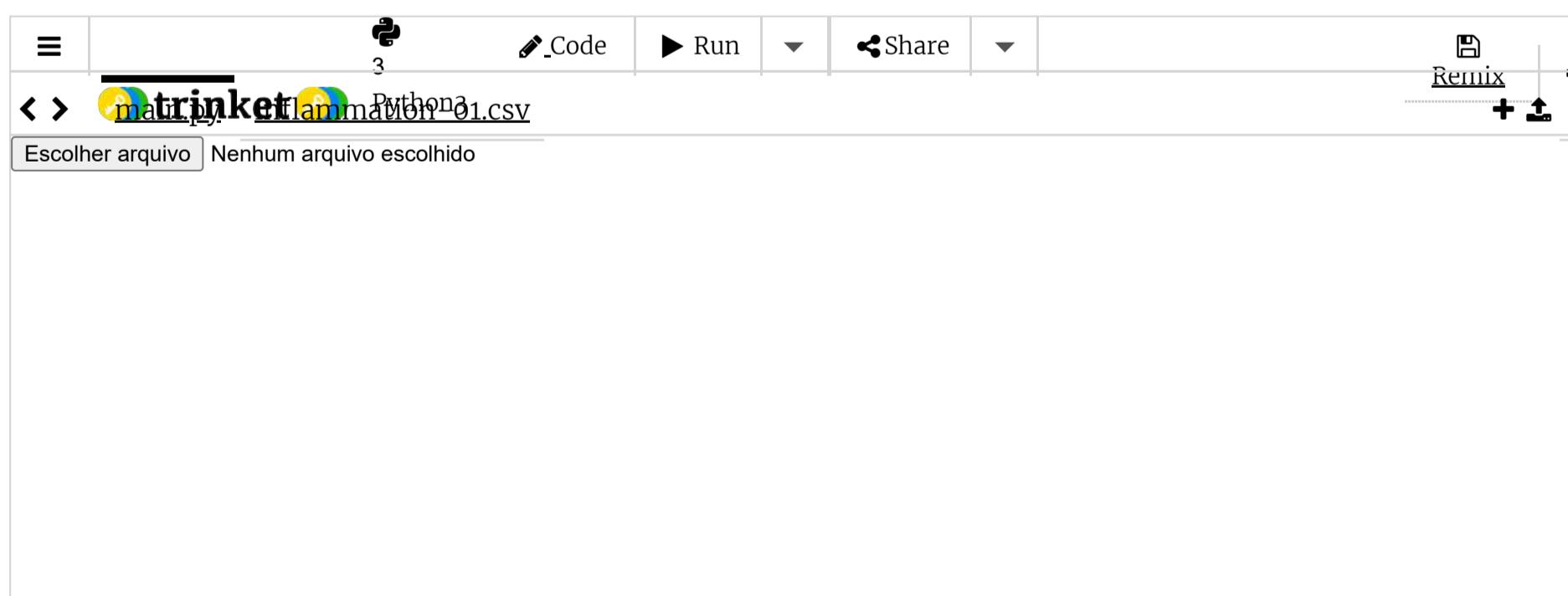
Na linha 9, fazemos uma série encadeada de transformações usando a função `replace()`, que age substituindo o primeiro parâmetro pelo segundo. No primeiro `replace(",", "")`, substituímos todas as vírgulas por nada. Então, na verdade, estamos apagando as vírgulas do texto sem colocar algo no lugar. No último `replace("\n", " ")`, substituímos as quebras de linhas por um espaço em branco.

Na linha 10 criamos uma lista ao aplicar a função `split()` ao texto. Nesse caso, sempre que houver um espaço em branco, a string será cortada, criando um novo elemento na lista. Veja no resultado que criamos uma lista com 58 elementos.

Na linha 13 está a "mágica": usamos a função `count()` para contar tanto a palavra "string" no singular quanto o seu plural "strings". Uma vez que a função retorna um número inteiro, somamos os resultados e os exibimos na linha 15.

Ver anotações

Essa foi uma amostra de como as estruturas de dados, funcionando como objetos na linguagem Python, permite implementar diversas soluções com poucas linhas de código, já que cada objeto já possui uma série de operações implementadas, prontas para serem usadas. Utilize o emulador a seguir para testar o código e explorar novas possibilidades.



## LISTAS

Lista é uma estrutura de dados do tipo sequencial que possui como principal característica ser mutável. Ou seja, novos valores podem ser adicionados ou removidos da sequência. Em Python, as listas podem ser construídas de várias maneiras:

0

Ver anotações

- Usando um par de colchetes para denotar uma lista vazia: lista1 = []
- Usando um par de colchetes e elementos separados por vírgulas: lista2 = ['a', 'b', 'c']
- Usando uma "list comprehension": [x for x in iterable]
- Usando o construtor de tipo: list()

Os objetos do tipo *sequência* são indexados, o que significa que cada elemento ocupa uma posição que começa em 0. Portanto, um objeto com 5 elementos terá índices que variam entre 0 e 4. O primeiro elemento ocupa a posição 0; o segundo, a posição 1; o terceiro, a posição 2; o quarto, a posição 3; e o quinto, a posição 4. Para facilitar a compreensão, pense na sequência como um prédio com o andar térreo. Embora, ao olhar de fora um prédio de 5 andares, contemos 5 divisões, ao adentrar no elevador, teremos as opções: T, 1, 2, 3, 4.

Observe, no código a seguir, a criação de uma lista chamada de "vogais". Por meio da estrutura de repetição, imprimos cada elemento da lista juntamente com seu índice. Veja que a sequência possui a função *index*, que retorna a posição de um valor na sequência.

In [5]:

```
vogais = ['a', 'e', 'i', 'o', 'u'] # também poderia ter sido criada usando aspas duplas
```

```
for vogal in vogais:
```

```
    print (f'Posição = {vogais.index(vogal)}', valor = {vogal}')
```

```
Posição = 0, valor = a
```

```
Posição = 1, valor = e
```

```
Posição = 2, valor = i
```

```
Posição = 3, valor = o
```

```
Posição = 4, valor = u
```

0

Ver anotações

Que tal testar o código da entrada 5 na ferramenta Python Tutor? Aproveite a ferramenta e explore novas possibilidades.

Python 3.6

```
→ 1 vogais = ['a', 'e', 'i', 'o', 'u']
  2 for vogal in vogais:
  3     print (f'Posição = {vogais.index('
  ▶ [ ] ▶
  Edit this code
```

Print output (drag lower right corner to resize)

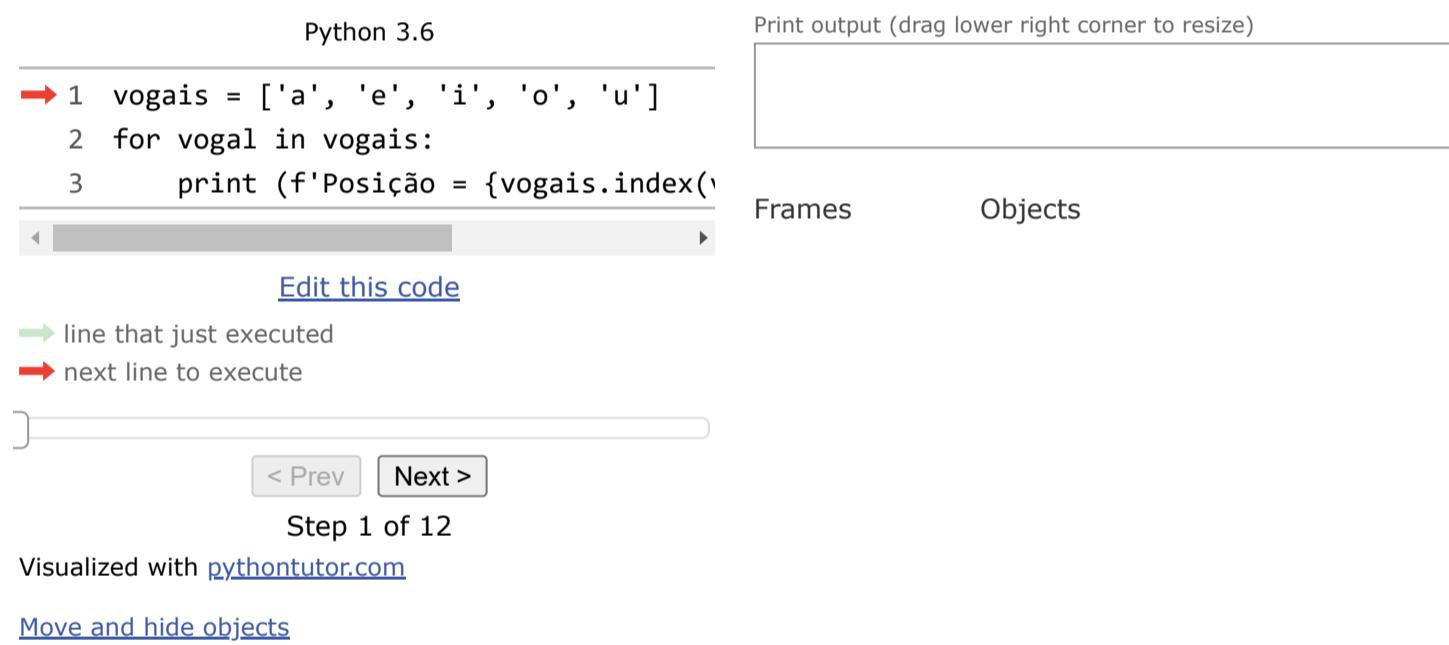
Frames Objects

< Prev Next >

Step 1 of 12

Visualized with [pythontutor.com](https://pythontutor.com)

[Move and hide objects](#)



In [6]:

```
# Mesmo resultado.

vogais = []

print(f"Tipo do objeto vogais = {type(vogais)}")

vogais.append('a')
vogais.append('e')
vogais.append('i')
vogais.append('o')
vogais.append('u')

for p, x in enumerate(vogais):
    print(f"Posição = {p}, valor = {x}")

Tipo do objeto vogais = <class 'list'>
Posição = 0, valor = a
Posição = 1, valor = e
Posição = 2, valor = i
Posição = 3, valor = o
Posição = 4, valor = u
```

0

Ver anotações

Veja, na entrada 6 (In [6]), que repetimos o exemplo com algumas alterações, a primeira das quais foi criar uma lista vazia na linha 3. Observe que, mesmo estando vazia, ao imprimirmos o tipo do objeto, o resultado é "class list", pois o colchete é a sintaxe para a construção de listas. Outra novidade foi usar a função *append()*, que adiciona um novo valor ao final da lista. Na sintaxe usamos *vogais.append(valor)*, notação que significa que a função *append()* é do objeto *lista*. Também substituímos o contador manual ("p") pela função *enumerate()*, que é usada para percorrer um objeto iterável retornando a posição e o valor. Por isso, na estrutura de repetição precisamos usar as variáveis *p* e *x*. A primeira guarda a posição e a segunda guarda o valor. Usamos o nome *x* propositalmente para que você perceba que o nome da variável é de livre escolha.

Agora que já sabemos criar uma lista, vamos testar algumas das operações mencionadas no Quadro 2.1. Observe o código a seguir:

In [7]:

```
frutas = ["maça", "banana", "uva", "mamão", "maça"]  
notas = [8.7, 5.2, 10, 3.5]  
  
print("maça" in frutas) # True  
print("abacate" in frutas) # False  
print("abacate" not in frutas) # True  
print(min(frutas)) # banana  
print(max(notas)) # 10  
print(frutas.count("maça")) # 2  
print(frutas + notas)  
print(2 * frutas)
```

```
True  
False  
True  
banana  
10  
2  
['maça', 'banana', 'uva', 'mamão', 'maça', 8.7, 5.2, 10, 3.5]  
['maça', 'banana', 'uva', 'mamão', 'maça', 'maça', 'banana', 'uva', 'mamão',  
'maça']
```

Na entrada 7 (In [7]), definimos duas listas, e, da linha 4 à linha 11, exploramos algumas operações com esse tipo de estrutura de dados. Nas linhas 4 e 5 testamos, respectivamente, se os valores "maça" e "abacate" estavam na lista, e os resultados foram True e False. Na linha 6, testamos se a palavra "abacate" não está na lista, e obtivemos True, uma vez que isso é verdade. Nas linhas 7 e 8 usamos as funções *mínimo* e *máximo* para saber o menor e o maior valor de cada lista. O mínimo de uma lista de palavras é feito sobre a ordem alfabética. Na linha 9,

0

Ver anotações

contamos quantas vezes a palavra "maça" aparece na lista. Na linha 10, concatenamos as duas listas e, na linha 11, multiplicamos por 2 a lista de frutas – veja no resultado que uma "cópia" da própria lista foi criada e adicionada ao final.

Até o momento, quando olhamos para a sintaxe de construção da lista, encontramos semelhanças com a construção de arrays. No entanto, a lista é um objeto muito versátil, pois sua criação suporta a mistura de vários tipos de dados conforme mostra o código a seguir. Além dessa característica, o fatiamento (slice) de estruturas sequenciais é uma operação muito valiosa. Observe o mencionado código:

In [8]:

```
lista = ['Python', 30.61, "Java", 51 , ['a', 'b', 20], "maça"]

print(f"Tamanho da lista = {len(lista)}")

for i, item in enumerate(lista):
    print(f"Posição = {i},\t valor = {item} -----> tipo individual = {type(item)})"

print("\nExemplos de slices:\n")

print("lista[1] = ", lista[1])
print("lista[0:2] = ", lista[0:2])
print("lista[:2] = ", lista[:2])
print("lista[3:5] = ", lista[3:5])
print("lista[3:6] = ", lista[3:6])
print("lista[3:] = ", lista[3:])
print("lista[-2] = ", lista[-2])
print("lista[-1] = ", lista[-1])
print("lista[4][1] = ", lista[4][1])
```

0

Ver anotações

```
Tamanho da lista = 6  
Posição = 0,      valor = Python -----> tipo individual = <class  
'str'>  
Posição = 1,      valor = 30.61 -----> tipo individual = <class  
'float'>  
Posição = 2,      valor = Java -----> tipo individual = <class 'str'  
Posição = 3,      valor = 51 -----> tipo individual = <class 'int'>  
Posição = 4,      valor = ['a', 'b', 20] -----> tipo individual =  
<class 'list'>  
Posição = 5,      valor = maça -----> tipo individual = <class 'str'>
```

0

Ver anotações

Exemplos de slices:

```
lista[1] = 30.61  
lista[0:2] = ['Python', 30.61]  
lista[:2] = ['Python', 30.61]  
lista[3:5] = [51, ['a', 'b', 20]]  
lista[3:6] = [51, ['a', 'b', 20], 'maça']  
lista[3:] = [51, ['a', 'b', 20], 'maça']  
lista[-2] = ['a', 'b', 20]  
lista[-1] = maça  
lista[4][1] = b
```

Na entrada 8 (In [8]), criamos uma lista que contém: texto, número (float e int) e lista! Isto mesmo: uma lista dentro de outra lista. Em Python, conseguimos colocar uma estrutura de dados dentro de outra sem ser necessário ser do mesmo tipo. Por exemplo, podemos colocar um dicionário dentro de uma lista. Para auxiliar a explicação do código, criamos a estrutura de repetição com enumerate, que indica o que tem em cada posição da lista. Veja que cada valor da lista pode ser um objeto diferente, sem necessidade de serem todos do mesmo tipo, como acontece em um array em C, por exemplo. Da linha 10 à linha 18 criamos alguns exemplos de slices. Vejamos a explicação de cada um:

o

Ver anotações

- lista[1]: acessa o valor da posição 1 da lista.
- lista[0:2]: acessa os valores que estão entre as posições 0 e 2. Lembre-se de que o limite superior não é incluído. Ou seja, nesse caso serão acessados os valores das posição 0 e 1.
- lista[:2]: quando um dos limites não é informado, o interpretador considera o limite máximo. Como não foi informado o limite inferior, então o slice será dos índices 0 a 2 (2 não incluído).
- lista[3:5]: acessa os valores que estão entre as posições 3 (incluído) e 5 (não incluído). O limite inferior sempre é incluído.
- lista[3:6]: os índices da lista variam entre 0 e 5. Quando queremos pegar todos os elementos, incluindo o último, devemos fazer o slice com o limite superior do tamanho da lista. Nesse caso, é 6, pois o limite superior 6 não será incluído.
- lista[3:]: outra forma de pegar todos os elementos até o último é não informar o limite superior.
- lista[-2]: o slice usando índice negativo é interpretado de trás para frente, ou seja, índice -2 quer dizer o penúltimo elemento da lista.
- lista[-1]: o índice -1 representa o último elemento da lista.

- lista[4][1]: no índice 5 da lista há uma outra lista, razão pela qual usamos mais um colchete para conseguir acessar um elemento específico dessa lista interna.

Esses exemplos nos dão uma noção do poder que as listas têm. Utilize o emulador a seguir para criar e testar novos fatiamentos em listas. O que acontece se tentarmos acessar um índice que não existe? Por exemplo: tentar imprimir print(lista[6]). Aproveite a oportunidade e explore-a!

0

Ver anotações

The screenshot shows a Python 3.6 code editor with the following code:

```

Python 3.6
Print output (drag lower right corner to resize)

1 lista = ['Python', 30.61, "Java", 5]
2
3 print(f"Tamanho da lista = {len(lista)}")
4
5 for i, item in enumerate(lista):
6     print(f"Posição = {i},\t valor = {item}")
7
8 print("\nExemplos de slices:\n")
9
10 print("lista[1] = ", lista[1])
11 print("lista[0:2] = ", lista[0:2])
12 print("lista[:2] = ", lista[:2])
13 print("lista[3:5] = ", lista[3:5])
14 print("lista[3:6] = ", lista[3:6])
15 print("lista[3:] = ", lista[3:])
16 print("lista[-2] = ", lista[-2])
17 print("lista[-1] = ", lista[-1])

```

The code is visualized with colored highlights: green for executed lines (e.g., line 1), red for the next line to execute (line 2), and grey for unexecuted lines (e.g., lines 3-17). To the right is a 'Print output' window with a 'Frames' tab selected, showing an empty list. Below the code editor are buttons for 'Edit this code', navigation arrows, and step controls ('< Prev', 'Next >', 'Step 1 of 25'). At the bottom, it says 'Visualized with [pythontutor.com](#)'.

As listas possuem diversas funções, além das operações já mencionadas. Na documentação oficial (PSF, 2020b) você encontra uma lista completa com todas as operações possíveis. No decorrer do nosso estudo vamos explorar várias delas.

## LIST COMPREHENSION (COMPREENSÕES DE LISTA)

A list comprehension, também chamada de listcomp, é uma forma pythônica de criar uma lista com base em um objeto iterável. Esse tipo de técnica é utilizada quando, dada uma sequência, deseja-se criar uma nova sequência, porém com as informações originais transformadas ou filtradas por um critério. Para entender essa técnica, vamos supor que tenhamos uma lista de palavras e desejamos padronizá-las para minúsculo. Observe o código a seguir.

o

Ver anotações

In [9]:

```
linguagens = ["Python", "Java", "JavaScript", "C", "C#", "C++", "Swift", "Go",  
"Kotlin"]
```

```
#linguagens = '''Python Java JavaScript C C# C++ Swift Go Kotlin'''.split() #  
Essa sintaxe produz o mesmo resultado que a linha 1
```

```
print("Antes da listcomp = ", linguagens)
```

```
linguagens = [item.lower() for item in linguagens]
```

```
print("\nDepois da listcomp = ", linguagens)
```

```
Antes da listcomp = ['Python', 'Java', 'JavaScript', 'C', 'C#', 'C++', 'Swift',  
'Go', 'Kotlin']
```

```
Depois da listcomp = ['python', 'java', 'javascript', 'c', 'c#', 'c++',  
'swift', 'go', 'kotlin']
```

Na entrada 9 (In [9]), criamos uma lista chamada "linguagens" que contém algumas linguagens de programação. Na linha 2, deixamos comentado outra forma de criar uma lista com base em um texto com utilização da função *split()*. Na linha 6, criamos nossa primeira list comprehension. Observação: como se trata da criação de uma lista, usam-se colchetes! Dentro do colchetes há uma variável chamada "item" que representará cada valor da lista original. Veja que usamos *item.lower()* for *item* in *linguagens*, e o resultado foi guardado dentro da mesma variável original, ou seja, sobrescrevemos o valor de "linguagens". Na saída fizemos a impressão antes e depois da listcomp. Veja a diferença.

A listcomp é uma forma pythônica de escrever um *for*. O código da entrada 9 poderia ser escrito conforme mostrado a seguir, e o resultado é o mesmo.

In [10]:

```
linguagens = '''Python Java JavaScript C C# C++ Swift Go Kotlin'''.split()
print("Antes da listcomp = ", linguagens)

for i, item in enumerate(linguagens):
    linguagens[i] = item.lower()

print("\nDepois da listcomp = ", linguagens)
```

```
Antes da listcomp =  ['Python', 'Java', 'JavaScript', 'C', 'C#', 'C++', 'Swift',
'Go', 'Kotlin']

Depois da listcomp =  ['python', 'java', 'javascript', 'c', 'c#', 'c++',
'swift', 'go', 'kotlin']
```

Agora vamos usar a listcomp para construir uma lista que contém somente as linguagens que possuem "Java" no texto. Veja o código a seguir.

In [11]:

Ver anotações

```
linguagens = '''Python Java JavaScript C C# C++ Swift Go Kotlin'''.split()

linguagens_java = [item for item in linguagens if "Java" in item]

print(linguagens_java)
['Java', 'JavaScript']
```

0

Ver anotações

Na entrada 11 (In [11]), a listcomp é construída com uma estrutura de decisão (if) fim de criar um filtro. Veja que a variável *item* será considerada somente se ela tiver "Java" no texto. Portanto, dois itens da lista original atendem a esse requisito e são adicionados à nova lista. Vale ressaltar que a operação *x* in *s* significa "*x* está em *s*?"; portanto, Java está em JavaScript. Se precisarmos criar um filtro que retorne somente palavras idênticas, precisamos fazer: linguagens\_java = [item for item in linguagens if "Java" == item]. A listcomp da entrada 6 poderia ser escrita conforme o código a seguir. Veja que precisaríamos digitar muito mais instruções.

In [12]:

```
linguagens = '''Python Java JavaScript C C# C++ Swift Go Kotlin'''.split()
linguagens_java = []

for item in linguagens:
    if "Java" in item:
        linguagens_java.append(item)

print(linguagens_java)
['Java', 'JavaScript']
```

## ■ FUNÇÕES MAP() E FILTER()

Não poderíamos falar de listas sem mencionar duas funções *built-in* que são usadas por esse tipo de estrutura de dados: map() e filter(). A função *map()* é utilizada para aplicar uma determinada função em cada item de um objeto iterável.

Para que essa transformação seja feita, a função *map()* exige que sejam passados dois parâmetros: a função e o objeto iterável. Observe os dois exemplos a seguir.

In [13]:

```
# Exemplo 1
print("Exemplo 1")
linguagens = '''Python Java JavaScript C C# C++ Swift Go Kotlin'''.split()

nova_lista = map(lambda x: x.lower(), linguagens)
print(f"A nova lista é = {nova_lista}\n")

nova_lista = list(nova_lista)
print(f"Agora sim, a nova lista é = {nova_lista}")

# Exemplo 2
print("\n\nExemplo 2")
numeros = [0, 1, 2, 3, 4, 5]

quadrados = list(map(lambda x: x*x, numeros))
print(f"Lista com o número elevado a ele mesmo = {quadrados}\n")
```

Exemplo 1

A nova lista é = <map object at 0x0000022E6F7CD0B8>

Agora sim, a nova lista é = ['python', 'java', 'javascript', 'c', 'c#', 'c++',  
'swift', 'go', 'kotlin']

Exemplo 2

Lista com o número elevado a ele mesmo = [0, 1, 4, 9, 16, 25]

0

Ver anotações

No exemplo 1 da entrada 13 (In [13]), criamos uma lista na linha 3; e, na linha 5, aplicamos a função *map()* para transformar cada palavra da lista em letras minúsculas. Veja que, como o primeiro parâmetro da função *map()* precisa ser uma função, optamos por usar uma função lambda. Na linha 6, imprimimos a nova lista. Se você olhar o resultado, verá: A nova lista é = map object at 0x00000237EB0EF32.

No entanto, onde está a nova lista?

A função *map* retorna um objeto iterável. Para que possamos ver o resultado, precisamos transformar esse objeto em uma lista. Fazemos isso na linha 8 ao aplicar o construtor *list()* para fazer a conversão. Por fim, na linha 9, fazemos a impressão da nova lista e, portanto, conseguimos ver o resultado.

No exemplo 2 da entrada 11 (In [11]), criamos uma lista numérica na linha 14; e, na linha 16, usamos a função *lambda* para elevar cada número da lista a ele mesmo (quadrado de um número). Na própria linha 16, já fazemos a conversão do resultado da função *map()* para o tipo *list*.

A função *filter()* tem as mesmas características da função *map()*, mas, em vez de usarmos uma função para transformar os valores da lista, nós a usamos para filtrar. Veja o código a seguir.

In [14]:

```
numeros = list(range(0, 21))

numeros_pares = list(filter(lambda x: x % 2 == 0, numeros))

print(numeros_pares)
```

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

Na entrada 14 (In [14]), a função *range()* cria um objeto numérico iterável. Então usamos o construtor *list()* para transformá-lo em uma lista com números, que variam de 0 a 20. Lembre-se de que o limite superior do argumento da função

*range()* não é incluído. Na linha 3, criamos uma nova lista com a função *filter*, que, com a utilização da expressão *lambda*, retorna somente os valores pares.

0

Ver anotações

## TUPLAS

As tuplas também são estruturas de dados do grupo de objetos do tipo sequêncio. A grande diferença entre listas e tuplas é que as primeiras são mutáveis, razão pqual, com elas, conseguimos fazer atribuições a posições específicas: por exemplo, lista[2] = 'maça'. Por sua vez, nas tuplas isso não é possível, uma vez que são objetos imutáveis.

Em Python, as tuplas podem ser construídas de três maneiras:

- Usando um par de parênteses para denotar uma tupla vazia: tupla1 = ()
- Usando um par de parênteses e elementos separados por vírgulas: tupla2 = ('a', 'b', 'c')
- Usando o construtor de tipo: tuple()

Observe o código a seguir, no qual criamos uma tupla chamada de "vogais" e, por meio da estrutura de repetição, imprimos cada elemento da tupla. Usamos, ainda, uma variável auxiliar *p* para indicar a posição que o elemento ocupa na tupla.

In [15]:

```
vogais = ('a', 'e', 'i', 'o', 'u')
print(f"Tipo do objeto vogais = {type(vogais)}")

for p, x in enumerate(vogais):
    print(f"Posição = {p}, valor = {x}")
```

```
Tipo do objeto vogais = <class 'tuple'>
Posição = 0, valor = a
Posição = 1, valor = e
Posição = 2, valor = i
Posição = 3, valor = o
Posição = 4, valor = u
```

Ver anotações

Com o exemplo apresentado na entrada 15 (In [15]), você pode estar pensando: "não vi diferença nenhuma entre usar lista e usar tupla". Em alguns casos, mais de uma estrutura realmente pode resolver o problema, mas em outros não. Voltamos à discussão inicial da nossa seção: objetos podem ter operações em comum entre si, mas cada objeto possui operações próprias. Por exemplo, é possível usar as operações do Quadro 2.1 nas tuplas. No entanto, veja o que acontece se tentarmos usar a função *append()* em uma tupla.

Ver anotações

In [16]:

```
vogais = ()  
  
vogais.append('a')
```

```
-----  
AttributeError Traceback (most recent call last)  
<ipython-input-16-df6bfea0326b> in <module>  
      1 vogais = ()  
      2  
----> 3 vogais.append('a')  
  
AttributeError: 'tuple' object has no attribute 'append'
```

In [17]:

```
vogais = ('a', 'e', 'i', 'o', 'u')  
  
for item in enumerate(vogais):  
    print(item)  
  
# print(tuple(enumerate(vogais)))  
# print(list(enumerate(vogais)))
```

```
(0, 'a')  
(1, 'e')  
(2, 'i')  
(3, 'o')  
(4, 'u')
```

A entrada 16 (In [16]) resulta no erro "AttributeError: 'tuple' object has no attribute 'append'", uma vez que a tupla não possui a operação de *append*.

Como a tupla é imutável, sua utilização ocorre em casos nos quais a ordem dos elementos é importante e não pode ser alterada, já que o objeto *tuple* garante essa característica. A função *enumerate()*, que normalmente usamos nas estruturas de repetição, retorna uma tupla cujo primeiro elemento é sempre o

Ver anotações

índice da posição e cujo segundo elemtno é o valor em si. Observe o código a seguir e veja que, para cada item de um enumerate(), é impressa uma tupla. Deixamos comentados duas outras formas de ver o resultado da função *enumerate()*. No primeiro comentário usamos o construtor *tuple()* para transformar o resultado em uma tupla, caso no qual temos uma tupla de tuplas. No segundo comentário, usamos o contrutor *list()*, caso no qual temos uma lista de tuplas. Não deixe de testar os códigos e explorar novas possibilidades.

0

Ver anotações

## OBJETOS DO TIPO *SET*

A tradução "conjunto" para *set* nos leva diretamente à essência desse tipo de estrutura de dados em Python. Um objeto do tipo *set* habilita operações matemáticas de conjuntos, tais como: união, intersecção, diferença, etc. Esse tipo de estrutura pode ser usado, portanto, em testes de associação e remoção de valores duplicados de uma sequência (PSF, 2020c).

Das operações que já conhecemos sobre sequências, conseguimos usar nessa nova estrutura:

- `len(s)`
- `x in s`
- `x not in s`

Além dessas operações, podemos adicionar um novo elemento a um conjunto com a função `add(valor)`. Também podemos remover com `remove(valor)`. Veja uma lista completa de funções no endereço <https://bit.ly/2NF7eIT>.

Em Python, os objetos do tipo *set* podem ser construídos destas maneiras:

- Usando um par de chaves e elementos separados por vírgulas: `set1 = {'a', 'b', 'c'}`
- Usando o construtor de tipo: `set(iterable)`

Não é possível criar um set vazio, com `set = {}`, pois essa é a forma de construção de um dicionário. Para construir com utilização da função `set(iterable)`, obrigatoriamente temos de passar um objeto iterável para ser transformado em conjunto. Esse objeto pode ser uma lista, uma tupla ou até mesmo uma string (que é um tipo de sequência). Veja os exemplos de construção a seguir.

Ver anotações

In [18]:

```
vogais_1 = {'aeiou'} # sem uso do construtor
print(type(vogais_1), vogais_1)

vogais_2 = set('aeiouaaaaaaaa') # construtor com string
print(type(vogais_2), vogais_2)

vogais_3 = set(['a', 'e', 'i', 'o', 'u']) # construtor com lista
print(type(vogais_3), vogais_3)

vogais_4 = set(('a', 'e', 'i', 'o', 'u')) # construtor com tupla
print(type(vogais_4), vogais_4)

print(set('banana'))
```

```
<class 'set'> {'aeiou'}
<class 'set'> {'u', 'o', 'i', 'e', 'a'}
<class 'set'> {'u', 'o', 'i', 'e', 'a'}
<class 'set'> {'u', 'o', 'i', 'e', 'a'}
{'n', 'a', 'b'}
```

Na entrada 18 (In [18]), criamos 4 exemplos de construção de objetos `set`. Com exceção do primeiro, no qual não usamos o construtor `set()`, os demais resultam na mesma estrutura. Veja que, no exemplo 2, passamos como parâmetro uma sequência de caracteres 'aeiouaaaaaaaa' e, propositalmente, repetimos a vogal *a*. O construtor interpreta como um iterável e cria um conjunto em que cada

caractere é um elemento, eliminando valores duplicados. Veja, na linha 13, o exemplo no qual transformamos a palavra 'banana' em um set, cujo resultado é a eliminação de caracteres duplicados.

0

### EXEMPLIFICANDO

O poder do objeto *set* está em suas operações matemáticas de conjuntos. Vejamos um exemplo: uma loja de informática recebeu componentes usados de um computador para avaliar se estão com defeito. As peças que não estiverem com defeito podem ser colocadas à venda. Como, então, podemos criar uma solução em Python para resolver esse problema? A resposta é simples: usando objetos do tipo set. Observe o código a seguir.

Ver anotações

In [19]:

```
def create_report():

    componentes_verificados = set(['caixas de som', 'cooler', 'dissipador
de calor', 'cpu', 'hd', 'estabilizador', 'gabinete', 'hub', 'impressora',
'joystick', 'memória ram', 'microfone', 'modem', 'monitor', 'mouse', 'no-
break', 'placa de captura', 'placa de som', 'placa de vídeo', 'placa
mãe', 'scanner', 'teclado', 'webcam'])

    componentes_com_defeito = set(['hd', 'monitor', 'placa de som',
'scanner'])

    qtde_componentes_verificados = len(componentes_verificados)
    qtde_componentes_com_defeito = len(componentes_com_defeito)

    componentes_ok =
        componentes_verificados.difference(componentes_com_defeito)

    print(f"Foram verificados {qtde_componentes_verificados}
componentes.\n")
    print(f"\n{qtde_componentes_com_defeito} componentes apresentaram
defeito.\n")

    print("Os componentes que podem ser vendidos são:")
    for item in componentes_ok:
        print(item)

create_report()
```

Ver anotações

Foram verificados 23 componentes.

4 componentes apresentaram defeito.

Os componentes que podem ser vendidos são:

placa mãe

no-break

cpu

dissipador de calor

estabilizador

mouse

placa de vídeo

hub

teclado

microfone

modem

caixas de som

memória ram

gabinete

webcam

cooler

placa de captura

impressora

joystick

0

Ver anotações

Na entrada 19 (In [19]), criamos uma função que gera o relatório das peças aptas a serem vendidas. Nessa função, são criados dois objetos *set*: "componentes\_verificados" e "componentes\_com\_defeito". Nas linhas 5 e 6, usamos a função *len()* para saber quantos itens há em cada conjunto. Na linha 8, fazemos a "mágica"! Usamos a função *difference()* para obter os itens que estão em *componentes\_verificados*, mas não em *componentes\_com\_defeito*. Essa operação também poderia ser feita com o

sinal de subtração: `componentes_ok = componentes_verificados - componentes_com_defeito`. Com uma única operação conseguimos extrair uma importante informação!

0

Aproveite o emulador a seguir para testar o exemplo e explorar novas possibilidades. Teste as funções `union()` e `intersection()` em novos conjuntos e vej o resultado.

Ver anotações



```
componentes_verificados = set(componentes)
componentes_com_defeito = set(componentes_verificados - componentes)
componentes_ok = componentes_verificados - componentes_com_defeito
```

## OBJETOS DO TIPO *MAPPING*

As estruturas de dados que possuem um mapeamento entre uma chave e um valor são consideradas objetos do tipo *mapping*. Em Python, o objeto que possui essa propriedade é o dict (dicionário). Uma vez que esse objeto é mutável, conseguimos atribuir um novo valor a uma chave já existente.

Podemos construir dicionários em Python das seguintes maneiras:

- Usando um par de chaves para denotar um dict vazio: `dicionario1 = {}`
- Usando um par de elementos na forma, chave : valor separados por vírgulas: `dicionario2 = {'one': 1, 'two': 2, 'three': 3}`
- Usando o construtor de tipo: `dict()`

Observe os exemplos a seguir com maneiras de construir o dicionário.

In [20]:

```
# Exemplo 1 - Criação de dicionário vazio, com atribuição posterior de chave e valor
dici_1 = {}
dici_1['nome'] = "João"
dici_1['idade'] = 30

# Exemplo 2 - Criação de dicionário usando um par elementos na forma, chave : valor
dici_2 = {'nome': 'João', 'idade': 30}

# Exemplo 3 - Criação de dicionário com uma lista de tuplas. Cada tupla representa um par chave : valor
dici_3 = dict([('nome', "João"), ('idade', 30)])

# Exemplo 4 - Criação de dicionário com a função built-in zip() e duas listas, uma com as chaves, outra com os valores.
dici_4 = dict(zip(['nome', 'idade'], ["João", 30]))
```

print(dici\_1 == dici\_2 == dici\_3 == dici\_4) # Testando se as diferentes construções resultam em objetos iguais.

True

Na entrada 20 (In [20]), usamos 4 sintaxes distintas para criar e atribuir valores a um dicionário. Da linha 2 à linha 4, criamos um dicionário vazio e, em seguida, criamos as chaves e atribuímos valores. Na linha 7, já criamos o dicionário com as chaves e os valores. Na linha 10, usamos o construtor *dict()* para criar, passando como parâmetro uma lista de tuplas: *dict([(tupla 1), (tupla 2)])*. Cada tupla deve ser uma combinação de chave e valor. Na linha 13, também usamos o construtor *dict()*, mas agora combinado com a função *built-in zip()*. A função *zip()* é usada para combinar valores de diferentes sequências e retorna um iterável de

0

Ver anotações

tuplas, em que o primeiro elemento é referente ao primeiro elemento da sequência 1, e assim por diante. Na linha 16, testamos se as diferentes construções produzem o mesmo objeto. O resultado True para o teste indica que são iguais.

Para acessar um valor em um dicionário, basta digitar: nome\_dicionario[chave]; para atribuir um novo valor use: nome\_dicionario[chave] = novo\_valor.

Que tal utilizar a ferramenta Python Tutor para explorar essa implementação e analisar o passo a passo do funcionamento do código. Aproveite a oportunidade!

0  
Ver anotações

Python 3.6

```
1 # Exemplo 1 - Criação de dicionário '
→ 2 dici_1 = {}
3 dici_1['nome'] = "João"
4 dici_1['idade'] = 30
5
6 # Exemplo 2 - Criação de dicionário '
7 dici_2 = {'nome': 'João', 'idade': 30}
8
9 # Exemplo 3 - Criação de dicionário '
10 dici_3 = dict([('nome', "João"), ('idade', 30)])
11
12 # Exemplo 4 - Criação de dicionário '
13 dici_4 = dict(zip(['nome', 'idade'],
14
15
16 print(dici_1 == dici_2 == dici_3 == dici_4))
```

Print output (drag lower right corner to resize)

Frames Objects

Edit this code

line that just executed  
next line to execute

< Prev Next >

Step 1 of 7

Visualized with [pythontutor.com](#)

Move and hide objects

Uma única chave em um dicionário pode estar associada a vários valores por meio de uma lista, tupla ou até mesmo outro dicionário. Nesse caso, também conseguimos acessar os elementos internos. No código a seguir, a função `keys()` retorna uma lista com todas as chaves de um dicionário. A função `values()` retorna uma lista com todos os valores. A função `items()` retorna uma lista de tuplas, cada uma das quais é um par chave-valor.

In [21]:

```
cadastro = {  
    'nome' : ['João', 'Ana', 'Pedro', 'Marcela'],  
    'cidade' : ['São Paulo', 'São Paulo', 'Rio de Janeiro', 'Curitiba'],  
    'idade' : [25, 33, 37, 18]  
}  
  
print("len(cadastro) = ", len(cadastro))  
  
print("\n cadastro.keys() = \n", cadastro.keys())  
print("\n cadastro.values() = \n", cadastro.values())  
print("\n cadastro.items() = \n", cadastro.items())  
  
print("\n cadastro['nome'] = ", cadastro['nome'])  
print("\n cadastro['nome'][2] = ", cadastro['nome'][2])  
print("\n cadastro['idade'][2:] = ", cadastro['idade'][2:])
```

Ver anotações

```
len(cadastro) = 3

cadastro.keys() =
dict_keys(['nome', 'cidade', 'idade'])

cadastro.values() =
dict_values([('João', 'Ana', 'Pedro', 'Marcela'), ('São Paulo', 'São Paulo',
'Rio de Janeiro', 'Curitiba'), [25, 33, 37, 18]])

cadastro.items() =
dict_items([('nome', ('João', 'Ana', 'Pedro', 'Marcela')), ('cidade', ('São
Paulo', 'São Paulo', 'Rio de Janeiro', 'Curitiba')), ('idade', [25, 33, 37,
18]))]

cadastro['nome'] = ['João', 'Ana', 'Pedro', 'Marcela']

cadastro['nome'][2] = Pedro

cadastro['idade'][2:] = [37, 18]
```

0

Ver anotações

Vamos avaliar os resultados obtidos com os códigos na entrada 21 (In [21]).

Primeiro, veja que criamos um dicionário em que cada chave está associada a uma lista de valores. A função `len()`, na linha 7, diz que o dicionário possui tamanho 3, o que está correto, pois `len()` conta quantas chaves existem no dicionário. Veja os demais comandos:

- `cadastro.keys()`: retorna uma lista com todas as chaves no dicionário.
- `cadastro.values()`: retorna uma lista com os valores. Como os valores também são listas, temos uma lista de listas.
- `cadastro.items()`: retorna uma lista de tuplas, cada uma das quais é composta pela chave e pelo valor.
- `cadastro['nome']`: acessa o valor atribuído à chave 'nome'; nesse caso, uma lista de nomes.
- `cadastro['nome'][2]`: acessa o valor na posição 2 da lista atribuída à chave 'nome'.
- `cadastro['idade'][2:]`: acessa os valores da posição 2 até o final da lista atribuída à chave 'nome'.

Como vimos, a função `len()` retorna quantas chaves um dicionário possui. No entanto, e se quiséssemos saber o total de elementos somando quantos há em cada chave? Embora não exista função que resolva esse problema diretamente, como conseguimos acessar os valores de cada chave, basta contarmos quantos eles são. Veja o código a seguir.

In [22]:

```
print(len(cadastro['nome']))
print(len(cadastro['cidade']))
print(len(cadastro['idade']))

qtde_itens = sum([len(cadastro[chave]) for chave in cadastro])

print(f"\n\nQuantidade de elementos no dicionário = {qtde_itens}")
```

4

4

4

Quantidade de elementos no dicionário = 12

0

Ver anotações

Nas três primeiras linhas da entrada 22, imprimimos a quantidade de elementos atribuídos a cada chave. Embora possamos simplesmente somar esses valores, o que faríamos se tivéssemos centenas de chaves? Para fazer essa contagem, independentemente de quantas chaves e valores existam, podemos criar uma list comprehension. Fizemos isso na linha 5. Veja que, para cada chave, usamos a função *len()*, criando assim uma lista de valores inteiros. A essa lista aplicamos a função *built-in sum()* para somar e obter a quantidade total de itens.

Como podemos ver, as estruturas de dados em Python são muito poderosas e versáteis. A combinação de diferentes estruturas permite criar soluções complexas com poucas linhas de código. Utilize o emulador para testar o código e explorar novas possibilidades.

The screenshot shows a Jupyter Notebook interface. At the top, there are tabs for 'Code', 'Run', 'Share', and 'Remix'. Below the tabs, the title 'matrinxflammati01.csv' is visible, along with a note 'Nenhum arquivo escolhido'. On the right side of the interface, there is a vertical sidebar with the text 'Ver anotações' and a small number '0'.

## OBJETOS DO TIPO ARRAY NUMPY

Quando o assunto é estrutura de dados em Python, não podemos deixar de falar dos objetos *array numpy*. Primeiramente, todas os objetos e funções que usamos até o momento fazem parte do *core* do interpretador Python, o que quer dizer que tudo está já instalado e pronto para usar. Além desses inúmeros recursos já disponíveis, podemos fazer um certo tipo de instalação e usar objetos e funções que outras pessoas/organizações desenvolveram e disponibilizaram de forma gratuita. Esse é o caso da biblioteca NumPy, criada especificamente para a computação científica com Python. O NumPy contém, entre outras coisas:

- Um poderoso objeto de matriz (array) N-dimensional.
- Funções sofisticadas.
- Ferramentas para integrar código C/C++ e Fortran.
- Recursos úteis de álgebra linear, transformação de Fourier e números aleatórios.

O endereço com a documentação completa da biblioteca NumPy está disponível em <https://numpy.org/>. Sem dúvida, o NumPy é a biblioteca mais poderosa para trabalhar com dados tabulares (matrizes), além de ser um recurso essencial para os desenvolvedores científicos, como os que desenvolvem soluções de inteligência artificial para imagens.

Para utilizar a biblioteca NumPy é preciso fazer a instalação com o comando pip install numpy. No entanto, se você estiver usando o projeto Anaconda, ou o Google Colab, esse recurso já estará instalado. Além da instalação, toda vez que for usar recursos da biblioteca, é necessário importar a biblioteca para o projeto, como o comando import numpy. Observe o código a seguir.

0

Ver anotações

In [23]:

```
import numpy

matriz_1_1 = numpy.array([1, 2, 3]) # Cria matriz 1 linha e 1 coluna
matriz_2_2 = numpy.array([[1, 2], [3, 4]]) # Cria matriz 2 linhas e 2 colunas
matriz_3_2 = numpy.array([[1, 2], [3, 4], [5, 6]]) # Cria matriz 3 linhas e 2
colunas
matriz_2_3 = numpy.array([[1, 2, 3], [4, 5, 6]]) # Cria matriz 2 linhas e 3
colunas

print(type(matriz_1_1))

print('\n matriz_1_1 = ', matriz_1_1)
print('\n matriz_2_2 = \n', matriz_2_2)
print('\n matriz_3_2 = \n', matriz_3_2)
print('\n matriz_2_3 = \n', matriz_2_3)
```

```
<class 'numpy.ndarray'>

matrix_1_1 = [1 2 3]

matrix_2_2 =
[[1 2]
[3 4]]

matrix_3_2 =
[[1 2]
[3 4]
[5 6]]

matrix_2_3 =
[[1 2 3]
[4 5 6]]
```

0

Ver anotações

Na entrada 23, criamos várias formas de matrizes com a biblioteca NumPy. Veja que, na linha 1, importamos a biblioteca para que pudéssemos usar seus objetos e funções. Para criar uma matriz, usamos `numpy.array(forma)`, em que *forma* são listas que representam as linhas e colunas. Veja que, nas linhas 5 e 6, criamos matrizes, respectivamente, com 3 linhas e 2 colunas e 2 linhas e 3 colunas. O que mudou de uma construção para a outra é que, para construir 3 linhas com 2 colunas, usamos três listas internas com dois valores, já para construir 2 linhas com 3 colunas, usamos duas listas com três valores cada.

NumPy é uma biblioteca muito rica. Veja algumas construções de matrizes usadas em álgebra linear já prontas, com um único comando.

In [24]:

```
m1 = numpy.zeros((3, 3)) # Cria matriz 3 x 3 somente com zero
m2 = numpy.ones((2,2)) # Cria matriz 2 x 2 somente com um
m3 = numpy.eye(4) # Cria matriz 4 x 4 identidade
m4 = numpy.random.randint(low=0, high=100, size=10).reshape(2, 5) # Cria matriz
2 X 5 com números inteiros
```

```
print('\n numpy.zeros((3, 3)) = \n', numpy.zeros((3, 3)))
print('\n numpy.ones((2,2)) = \n', numpy.ones((2,2)))
print('\n numpy.eye(4) = \n', numpy.eye(4))
print('\n m4 = \n', m4)
```

```
print(f"Soma dos valores em m4 = {m4.sum()}")
print(f"Valor mínimo em m4 = {m4.min()}")
print(f"Valor máximo em m4 = {m4.max()}")
print(f"Média dos valores em m4 = {m4.mean()}")
```

Ver anotações

```
numpy.zeros((3, 3)) =  
[[0. 0. 0.]  
[0. 0. 0.]  
[0. 0. 0.]]
```

0

```
numpy.ones((2,2)) =  
[[1. 1.]  
[1. 1.]]
```

Ver anotações

```
numpy.eye(4) =  
[[1. 0. 0. 0.]  
[0. 1. 0. 0.]  
[0. 0. 1. 0.]  
[0. 0. 0. 1.]]
```

```
m4 =  
[[20 46 25 93 94]  
[ 5 12 19 48 69]]
```

Soma dos valores em m4 = 431

Valor mínimo em m4 = 5

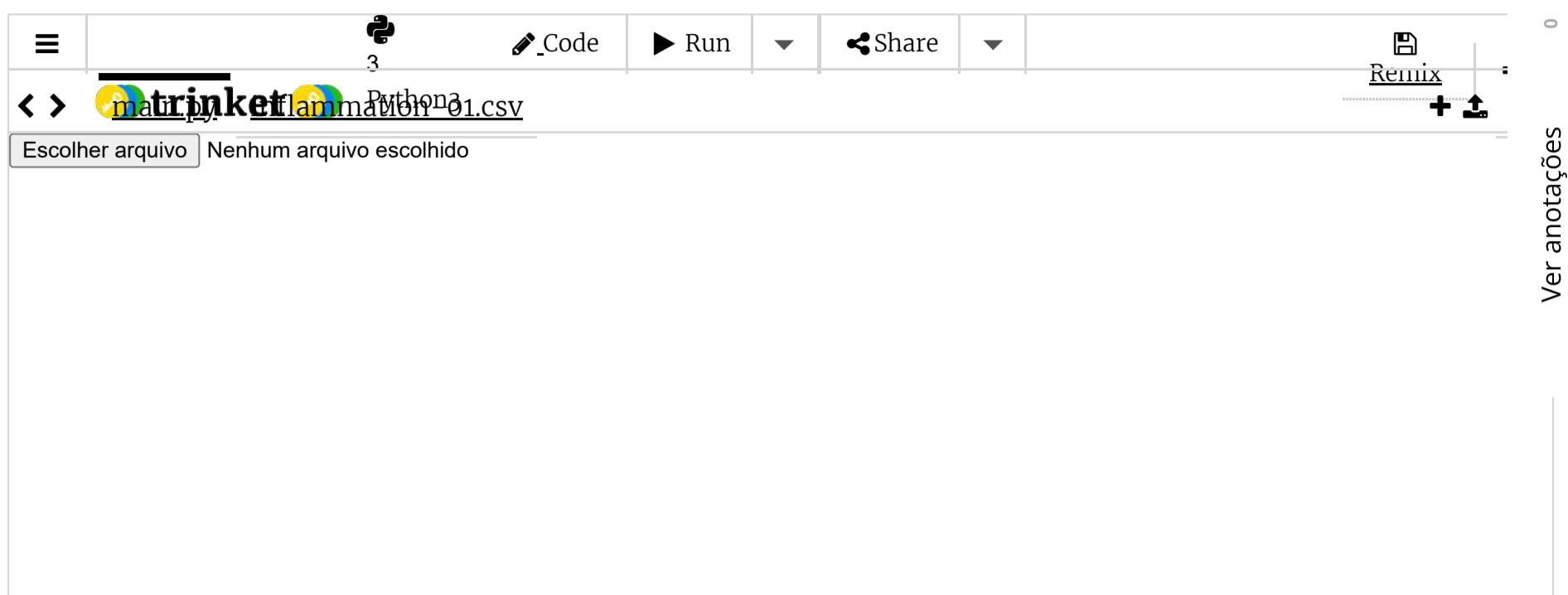
Valor máximo em m4 = 94

Média dos valores em m4 = 43.1

Na entrada 24, criamos matrizes somente com 0, com 1 e matriz identidade (1 na diagonal principal) usando comandos específicos. Por sua vez, a matriz 'm4' foi criada usando a função que gera números inteiros aleatórios. Escolhemos o menor valor como 0 e o maior como 100, e também pedimos para serem gerados 10 números. Em seguida, usamos a função *reshape()* para transformá-los em uma matriz com 2 linhas e 5 colunas. Das linhas 11 a 14, usamos funções que extraem informações estatísticas básicas de um conjunto numérico.

Com essa introdução ao uso da biblioteca NumPy, encerramos esta seção, na qual aprendemos uma variedade de estruturas de dados em Python. Para esse desenvolvimento é importante utilizar o emulador para testar os códigos e

implementar novas possibilidades. Será que conseguimos colocar um array NumPy dentro de um dicionário? Que tal testar?



## REFERÊNCIAS E LINKS ÚTEIS

BANIN, S. L. **Python 3** - conceitos e aplicações: uma abordagem didática. São Paulo: Érica, 2018.

NUMPT.ORG. Getting Started. Página inicial. 2020. Disponível em:  
<https://numpy.org/>. Acesso em: 25 abr. 2020.

PERKOVIC, L. **Introdução à computação usando Python**: um foco no desenvolvimento de aplicações. Rio de Janeiro: LTC, 2016.

PSF - Python Software Fundation. Data model. 2020a. Disponível em:  
<https://bit.ly/3iAcCL4>. Acesso em: 25 abr. 2020.

PSF - Python Software Fundation. Data Structures. 2020b. Disponível em:  
<https://bit.ly/3aqdoY3>. Acesso em: 25 abr. 2020.

PSF - Python Software Fundation. Built-in Types. 2020c. Disponível em:  
<https://bit.ly/3ixH6NE>. Acesso em: 25 abr. 2020.

# FOCO NO MERCADO DE TRABALHO

## ESTRUTURAS DE DADOS EM PYTHON

Vanessa Cadan Scheffer

0

Ver anotações

### UTILIZANDO AS ESTRUTURAS DE DADOS OFERECIDAS EM PYTHON

Em Python não existe somente uma forma de implementar uma solução, uma vez que são oferecidas uma série de estruturas de dados.



Fonte: Shutterstock.

### Deseja ouvir este material?

Áudio disponível no material digital.

### DESAFIO

Muitos sistemas disponibilizam as informações por meio de uma API (interface de programação de aplicações), as quais utilizam arquivos JSON (notação de objetos JavaScript) para disponibilizar os dados. Um arquivo JSON é composto por

elementos na forma chave-valor, o que torna esse formato leve de ser transferido e fácil de ser manipulado.

Como desenvolvedor em uma empresa de consultoria, você foi alocado para atender um cliente que organiza processos seletivos (concurso público, vestibular etc.). Essa empresa possui um sistema web no qual os candidatos de um certo processo seletivo fazem a inscrição e acompanham as informações. Um determinado concurso precisou ser adiado, razão pela qual um processo no servidor começou a enviar e-mails para todos os candidatos inscritos. Porém, em virtude da grande quantidade de acessos, o servidor e a API saíram do ar por alguns segundos, o que gerou um rompimento na criação do arquivo JSON com a lista dos candidatos já avisados da alteração.

Por causa do rompimento do arquivo, foram gerados dois novos arquivos, razão pela qual, desde então, não se sabe nem quem nem quantas pessoas já receberam o aviso. Seu trabalho, neste caso, é criar uma função que, com base nas informações que lhe serão fornecidas, retorne uma lista com os e-mails que ainda precisam ser enviados.

Para esse projeto, você e mais um desenvolvedor foram alocados. Enquanto seu colega trabalha na extração dos dados da API, você cria a lógica para gerar a função. Foi combinado, entre vocês, que o programa receberá dois dicionários referentes aos dois arquivos que foram gerados. O dicionário terá a seguinte estrutura: três chaves (*nome*, *email*, *enviado*), cada uma das quais recebe uma lista de informações; ou seja, as chaves *nome* e *email* estarão, respectivamente, associadas a uma lista de nomes e uma de emails. Por sua vez, a chave *enviado* estará associada a uma lista de valores booleanos (True-False) que indicará se o e-mail já foi ou não enviado.

Veja um exemplo.

```
dict_1 = {
```

```
'nome': ['nome_1'],
```

0

Ver anotações

```
'email': ['email_1'],  
'enviado': [False]  
}
```

0

Ver anotações

Considerando que você receberá duas estruturas, conforme foi mencionado, crie uma função que trate os dados e retorne uma lista com os e-mails que ainda não foram enviados.

## RESOLUÇÃO

Você foi alocado para um projeto no qual precisa implementar uma função que, com base em dois dicionários, retorne uma lista com os e-mails que precisam ser enviados aos candidatos de um concurso. Os dicionários serão fornecidos para você no formato combinado previamente. Ao mesmo tempo em que seu colega trabalha na extração dos dados da API, foi-lhe passado uma amostra dos dados para que você comece a trabalhar na lógica da função. Os dados passados foram:

```
dados_1 = {  
    'nome': ['Sonia Weber', 'Daryl Lowe', 'Vernon Carroll', 'Basil Gilliam',  
    'Mechelle Cobb', 'Edan Booker', 'Igor Wyatt', 'Ethan Franklin', 'Reed  
    Williamson', 'Price Singleton'],  
    'email': ['Lorem.ipsum@cursusvestibulumMauris.com', 'auctor@magnis.org',  
    'at@magnaUttincidunt.org', 'mauris.sagittis@sem.com',  
    'nec.euismod.in@mattis.co.uk', 'egestas@massaMaurisvestibulum.edu',  
    'semper.auctor.Mauris@Crasdolor dolor.edu', 'risus.Quisque@condimentum.com',  
    'Donec@nislMaecenasmalesuada.net', 'Aenean.gravida@atrisus.edu'],  
    'enviado': [False, False, False, False, False, False, False, True, False,  
    False]  
}
```

```
dados_2 = {  
    'nome': ['Travis Shepherd', 'Hoyt Glass', 'Jennifer Aguirre', 'Cassady Ayers',  
    'Colin Myers', 'Herrod Curtis', 'Cecilia Park', 'Hop Byrd', 'Beatrice Silva',  
    'Alden Morales'],  
    'email': ['at@sed.org', 'ac.arcu.Nunc@auctor.edu',  
    'nunc.Quisque.ornare@nibhAliquam.co.uk', 'non.arcu@mauriseu.com',  
    'fringilla.cursus.purus@erategetipsum.ca', 'Fusce.fermentum@tellus.co.uk',  
    'dolor.tempus.non@ipsum.net', 'blandit.congue.In@libero.com',  
    'nec.tempus.mauris@Suspendisse.com', 'felis@urnaconvalliserat.org'],  
    'enviado': [False, False, False, True, True, True, False, True, True, False]  
}
```

Com a estrutura do dicionário definida e com uma amostra dos dados, é possível começar a implementação. Vale lembrar não existe somente uma forma de implementar uma solução, ainda mais se tratando da linguagem Python, que oferece uma série de estruturas de dados. A seguir trazemos uma possível solução.

0

Ver anotações

A função "extrair\_lista\_email" recebe como parâmetro dois dicionários de dados. Para que pudéssemos fazer a extração, criamos a lista\_1 (linha 2), que consiste em uma lista de tuplas, na qual cada uma destas é composta por *nome*, *email*, *enviado*, exatamente nessa sequência. Para construir essa tupla, usamos a função *zip()*, passando com parâmetro a lista de nomes, de e-mails e o status do enviado e transformamos seu resultado em uma lista.

Na linha 3 imprimos uma única tupla construída para que pudéssemos checar a construção.

Na linha 5, criamos uma segunda lista de tuplas, agora usando os dados do segundo dicionário.

Para termos a lista completa de dados, foi necessário juntar as duas construções, fizemos isso na linha 7, simplesmente usando o '+' para concatenar as duas listas.

Na linha 12 fizemos a "mágica" de extrair somente os e-mails usando uma *list comprehension*. Vamos entender: "dados" é uma lista de tuplas, conforme amostra que imprimimos. Cada item dessa lista é uma tupla. Quando selecionamos o item[1], estamos pegando o valor que ocupa a posição 1 da tupla, ou seja, o e-mail. Fazendo isso, iterando sobre todos os dados, teremos uma lista com todos os e-mails. No entanto, queremos somente os e-mails que ainda não foram enviados, razão pela qual o valor da posição 2 na tupla tem que ser *Falso*. Para fazer esse filtro, incluímos na listcomp uma estrutura condicional (if) que adiciona somente "if not item[2]", ou seja, somente se o item[2] não tiver valor *True*. Com essa construção, extraímos exatamente o que precisamos.

Na linha 14 retornamos a lista.

Veja que as funções e os objetos em Python facilitaram o trabalho. Com a função *zip()* conseguimos extrair cada registro do dicionário, depois, com uma simples concatenação, juntamos todos os dados e, com uma única linha, usando a *list comprehension*, criamos a lista com os critérios que foram estabelecidos.

Ver anotações

In [25]:

0

Ver anotações

```
def extrair_lista_email(dict_1, dict_2):  
    lista_1 = list(zip(dict_1['nome'], dict_1['email'], dict_1['enviado']))  
    print(f"Amostra da lista 1 = {lista_1[0]}")  
  
    lista_2 = list(zip(dict_2['nome'], dict_2['email'], dict_2['enviado']))  
  
    dados = lista_1 + lista_2  
  
    print(f"\nAmostra dos dados = \n{dados[:2]}\n\n")  
  
    # Queremos uma lista com o email de quem ainda não recebeu o aviso  
    emails = [item[1] for item in dados if not item[2]]  
  
    return emails
```

```
dados_1 = {  
    'nome': ['Sonia Weber', 'Daryl Lowe', 'Vernon Carroll', 'Basil Gilliam',  
    'Mechelle Cobb', 'Edan Booker', 'Igor Wyatt', 'Ethan Franklin', 'Reed  
    Williamson', 'Price Singleton'],  
    'email': ['Lorem.ipsum@cursusvestibulumMauris.com', 'auctor@magnis.org',  
    'at@magnaUttincidunt.org', 'mauris.sagittis@sem.com',  
    'nec.euismod.in@mattis.co.uk', 'egestas@massaMaurisvestibulum.edu',  
    'semper.auctor.Mauris@Crasdolor dolor.edu', 'risus.Quisque@condimentum.com',  
    'Donec@nislMaecenasmalesuada.net', 'Aenean.gravida@atrisus.edu'],  
    'enviado': [False, False, False, False, False, False, False, True, False,  
    False]  
}
```

```
dados_2 = {  
    'nome': ['Travis Shepherd', 'Hoyt Glass', 'Jennifer Aguirre', 'Cassady  
    Ayers', 'Colin Myers', 'Herrod Curtis', 'Cecilia Park', 'Hop Byrd', 'Beatrice  
    Silva', 'Alden Morales'],  
    'email': ['at@sed.org', 'ac.arcu.Nunc@auctor.edu',  
    'nunc.Quisque.ornare@nibhAliquam.co.uk', 'non.arcu@mauriseu.com'],
```

0

Ver anotações

```
'fringilla.cursus.purus@erategetipsum.ca', 'Fusce.fermentum@tellus.co.uk',  
'dolor.tempus.non@ipsum.net', 'blandit.congue.In@libero.com',  
'nec.tempus.mauris@Suspendisse.com', 'felis@urnaconvalliserat.org'],  
    'enviado': [False, False, False, True, True, True, False, True, True, False]  
}
```

```
emails = extrair_lista_email(dict_1=dados_1, dict_2=dados_2)  
print(f"E-mails a serem enviados = \n {emails}")
```

```
Amostra da lista 1 = ('Sonia Weber', 'Lorem.ipsum@cursusvestibulumMauris.com',  
False)
```

```
Amostra dos dados =  
[('Sonia Weber', 'Lorem.ipsum@cursusvestibulumMauris.com', False), ('Daryl  
Lowe', 'auctor@magnis.org', False)]
```

```
E-mails a serem enviados =  
['Lorem.ipsum@cursusvestibulumMauris.com', 'auctor@magnis.org',  
'at@magnaUttincidunt.org', 'mauris.sagittis@sem.com',  
'nec.euismod.in@mattis.co.uk', 'egestas@massaMaurisvestibulum.edu',  
'semper.auctor.Mauris@Crasdolor dolor.edu', 'Donec@nis1Maecenasmalesuada.net',  
'Aenean.gravida@atrisus.edu', 'at@sed.org', 'ac.arcu.Nunc@auctor.edu',  
'nunc.Quisque.ornare@nibhAliquam.co.uk', 'dolor.tempus.non@ipsum.net',  
'felis@urnaconvalliserat.org']
```

## DESAFIO DA INTERNET

Na página 116 do Capítulo 4 da seguinte obra, você encontra o exercício 6 (uso de listas). Nele, o autor propõe a construção de uma progressão aritmética com o uso de listas em Python.

BANIN, S. L. Python 3 - conceitos e aplicações: uma abordagem didática. São Paulo: Érica, 2018. Disponível em: <https://bit.ly/2Njnf0w>. Acesso em: 30 jun. 2020.

Ver anotações

Utilize o emulador a seguir, para resolver o desafio!

The screenshot shows a Jupyter Notebook interface. At the top, there is a toolbar with icons for file operations, code editing, running cells, sharing, and remixing. Below the toolbar, the title bar displays the file path: 'matrinxerflammation01.csv' and 'Python3'. A message 'Escolher arquivo' (Select file) and 'Nenhum arquivo escolhido' (No file selected) is shown. The main area is a large, empty white space where code would be written and executed.

Ver anotações

NÃO PODE FALTAR

## ALGORITMOS DE BUSCA

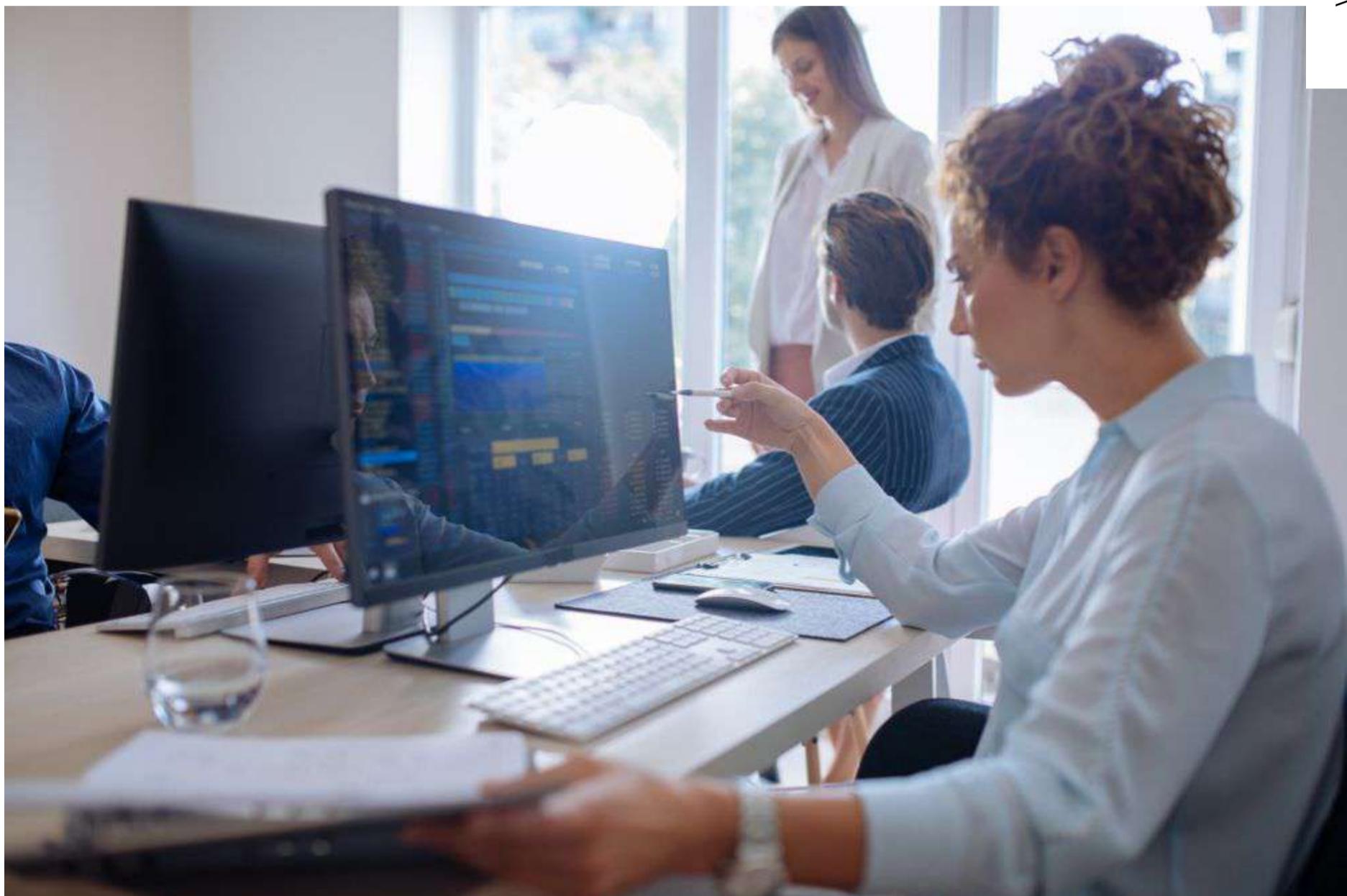
Vanessa Cadan Scheffer

0

Ver anotações

### ANALISANDO OS COMANDOS DOS MECANISMOS DE BUSCAS

Vamos descobrir como se comportam os algoritmos de busca.



Fonte: Shutterstock.

**Deseja ouvir este material?**

Áudio disponível no material digital.

### INTRODUÇÃO

Você já fez alguma pesquisa no Google hoje? Já fez a busca por alguma passagem de avião, de ônibus, ou até mesmo a aquisição de algum item em um site de compras? Já procurou alguém nas redes sociais? Se tivermos o cadastro de clientes

em uma loja, como faremos a busca por esses cliente? Podemos ter o cadastro de uma lista de RG e CPF para realizar essa busca.

Enfim, todas essas aplicações utilizam mecanismos de busca. Um grande diferencial entre uma ferramenta e outra é a velocidade da resposta: quanto mais rápido ela for, mais eficiente e melhor será a aceitação dela. Por trás de qualquer tipo de pesquisa existe um algoritmo de busca sendo executado. Nesta aula vamos aprender dois deles.

0

Ver anotações

## ■ ALGORITMOS DE BUSCAS

"Os algoritmos são o cerne da computação" (TOSCANI; VELOSO, 2012, p. 1).

Com essa afirmação, iniciamos nossa aula, na qual estudaremos algoritmos de busca. Os algoritmos computacionais são desenvolvidos e usados para resolver os mais diversos problemas. Nesse universo, como o nome sugere, os algoritmos resolvem problemas relacionados ao encontro de valores em uma estrutura de dados.

Diversas são as aplicações que utilizam esse mecanismo.

Já nos deparamos, como usuários, com um algoritmo de busca nesta disciplina quando estudamos as estruturas de dados. Em Python, temos a operação "in" ou "not in" usada para verificar se um valor está em uma sequência. Quando utilizamos esses comandos, estamos sendo "usuários" de um algoritmo que alguém escreveu e "encapsulou" neles. No entanto, como profissionais de tecnologia, precisamos saber o que está por trás do comando. Observe o exemplo:

In [1]:

```
nomes = 'João Marcela Sonia Daryl Vernon Eder Mechelle Edan Igor Ethan Reed  
Travis Hoyt'.split()  
  
print('Marcela' in nomes)  
print('Roberto' in nomes)  
True  
False
```

0

Ver anotações

Na entrada 1 (ln [1]), usamos o operador *in* para verificar se dois nomes constavam na lista. No primeiro, obtivemos True; e no segundo, False. O que, no entanto, há "por trás" desse comando *in*? Que algoritmo foi implementado nesse operador? Um profissional de tecnologia, além de saber implementar, precisa conhecer o funcionamento dos algoritmos.

## | BUSCA LINEAR (OU BUSCA SEQUENCIAL)

Como o nome sugere, uma busca linear (ou exaustiva) simplesmente percorre os elementos da sequência procurando aquele de destino, conforme ilustra a Figura 2.1.

Figura 2.1 | Busca sequencial



Fonte: elaborada pela autora.

Veja que a busca começa por uma das extremidades da sequência e vai percorrendo até encontrar (ou não) o valor desejado. Com essa imagem, fica claro que uma pesquisa linear examina todos os elementos da sequência até encontrar o de destino, o que pode ser muito custoso computacionalmente, conforme veremos adiante.

Para implementar a busca linear, vamos precisar de uma estrutura de repetição (for) para percorrer a sequência, e uma estrutura de decisão (if) para verificar se o valor em uma determinada posição é o que procuramos. Vejamos como fazer a implementação em Python.

In [2]:

```
def executar_busca_linear(lista, valor):
    for elemento in lista:
        if valor == elemento:
            return True
    return False
```

In [3]:

```
import random

lista = random.sample(range(1000), 50)
print(sorted(lista))
executar_busca_linear(lista, 10)
```

```
[52, 73, 95, 98, 99, 103, 123, 152, 158, 173, 259, 269, 294, 313, 318, 344, 346,
348, 363, 387, 407, 410, 414, 433, 470, 497, 520, 530, 536, 558, 573, 588, 620,
645, 677, 712, 713, 716, 720, 727, 728, 771, 790, 801, 865, 898, 941, 967, 970,
979]
```

Out[3]:

```
False
```

Ver anotações

Na entrada 2 (In [2]) criamos a função "executar\_busca\_linear", que recebe uma lista e um valor a ser localizado. Na linha 2, criamos a estrutura de repetição, que percorrerá cada elemento da lista pela comparação com o valor buscado (linha 3). Caso este seja localizado, então a função retorna o valor booleano True; caso não seja encontrado, então retorna False.

Para testarmos a função, criamos o código na entrada 3 (In[3]), no qual, por meio da biblioteca random (não se preocupe com a implementação, já que ainda veremos o uso de bibliotecas), criamos uma lista de 50 valores com números inteiros randômicos que variam entre 0 e 1000; e na linha 5 invocamos a função, passando a lista e um valor a ser localizado. Cada execução desse código gerará uma lista diferente, e o resultado poderá alterar.

Nossa função é capaz de determinar se um valor está ou não presente em uma sequência, certo? E se, no entanto, quiséssemos também saber sua posição na sequência? Em Python, as estruturas de dados do tipo *sequência* possuem a função *index()*, que é usada da seguinte forma: *sequencia.index(valor)*. A função *index()* espera como parâmetro o valor a ser procurado na sequência. Observe o código a seguir.

In [4]:

```
vogais = 'aeiou'

resultado = vogais.index('e')
print(resultado)
```

1

Será que conseguimos implementar nossa própria versão de busca por index com utilização da busca linear? A resposta é sim! Podemos iterar sobre a lista e, quando o elemento for encontrado, retornar seu índice. Caso não seja encontrado, então a função deve retornar None. Vale ressaltar a importância dos tipos de valores que

Ver anotações

serão usados para realizar a busca. Dada uma lista numérica, somente podem ser localizados valores do mesmo tipo. O mesmo para uma sequência de caracteres, que só pode localizar letras, palavras ou ainda uma string vazia. O tipo None não pode ser localizado em nenhuma lista – se tentar passar como parâmetro, poderá receber um erro. Observe a função "procurar\_valor" a seguir.

Ver anotações

In [5]:

```
def procurar_valor(lista, valor):
    tamanho_lista = len(lista)
    for i in range(tamanho_lista):
        if valor == lista[i]:
            return i
    return None
```

In [6]:

```
vogais = 'aeiou'

resultado = procurar_valor(lista=vogais, valor='a')

if resultado != None:
    print(f"Valor encontrado na posição {resultado}")
else:
    print("Valor não encontrado")
```

```
Valor encontrado na posição 0
```

Na entrada 5 (In [5]), criamos a função "procurar\_valor", a fim de retornar a posição de um valor, caso ele seja encontrado. Na entrada 6 (In [6]), criamos uma lista com as vogais e invocamos a função "procurar\_valor", passando a lista e um valor a ser procurado. Na linha 5, testamos se existe valor na variável "resultado", já que, caso o valor guardado em "resultado" for None, então o else é acionado.

Vamos testar o código, agora, buscando o valor "o" na lista de vogais. Observe que a lista tem tamanho 5 (número de vogais), sendo representada com os índices 0 a 4. O valor foi alterado para a vogal "o" na linha 19 do código. Teste o código utilizando a ferramenta Python Tutor.

0

Ver anotações

The screenshot shows the Python Tutor interface. On the left, there is a code editor window titled "Python 3.6" containing the following code:

```
1 def procurar_valor(lista, valor):
2     tamanho_lista = len(lista)
3     for i in range(tamanho_lista):
4         if valor == lista[i]:
5             return i
6     return None
7
8
9 def testar_resultado(resultado):
10    if resultado:
11        print(f"Valor encontrado na posição {resultado}")
12    else:
13        print("Valor não encontrado")
14
15
16 vogais = ['a', 'e', 'i', 'o', 'u']
17
```

Below the code editor, there is a legend:

- Green arrow: line that just executed
- Red arrow: next line to execute

At the bottom of the code editor, there are navigation buttons: "< Prev", "Next >", and "Step 1 of 21".

On the right side of the interface, there is a large rectangular box labeled "Print output (drag lower right corner to resize)". Below this box are two tabs: "Frames" and "Objects".

Observe que o valor "o" foi encontrado na posição 3 da lista. Nesse contexto, foi necessário percorrer os índices 0, 1, 2 e 3 até chegar à vogal procurada. Assim funciona a busca linear (sequencial): todas as posições são visitadas até que se encontre o elemento buscado. Caso a busca fosse por um valor que não está na lista, o valor retornado seria `None`.

Acabamos de implementar nossa versão da função `index`. Será que ela é melhor que a função `index()` das sequências em Python? Se tentarmos executar a função `index()`, passando como parâmetro um valor que não está na lista, teremos um erro como resultado: "ValueError". Por sua vez, na nossa versão, caso o valor não esteja na lista, não será retornado um erro, mas simplesmente o valor `None`. Você pode utilizar o emulador para fazer novos testes e aprimorar seus estudos!

The screenshot shows a Jupyter Notebook interface with the following details:

- Header:** Includes icons for file operations (three horizontal lines, back, forward), Python logo (3), Code, Run, Share, and Remix.
- Title Bar:** Shows the title "matrixflammation\_01.csv" and "Python3".
- Cell Content:** A single cell with the text "Escolher arquivo" and "Nenhum arquivo escolhido".
- Right Panel:** Displays a small icon of a document with a plus sign and an upward arrow, followed by a "0" indicating no annotations.

Ver anotações

## PRECISAMOS FALAR SOBRE COMPLEXIDADE

Nossa função *procurar\_valor* tem um comportamento diferente da função *index()* para os valores não encontrados. Será que isso é uma característica que pode ser levada em consideração para determinar que um algoritmo é melhor que o outro? A resposta é não!

Em termos computacionais, um algoritmo é considerado melhor que o outro quando, para a mesma entrada, utiliza menos recursos computacionais em termos de memória e processamento. Um exemplo clássico é a comparação entre os métodos de Cramer e Gauss, usados para resolver equações lineares. O método de Cramer pode levar dezenas de milhões de anos para resolver um sistema matricial de 20 linhas por 20 colunas, ao passo que o método de Gauss pode levar alguns segundos (TOSCANI; VELOSO, 2012). Veja a diferença: para a mesma entrada há um algoritmo que é inviável! Esse estudo da viabilidade de um algoritmo, em termos de espaço e tempo de processamento, é chamado de análise da complexidade do algoritmo.

Para começarmos a compreender intuitivamente, pense na lista de alunos, em ordem alfabética, de toda a rede Kroton/Cogna: estamos falando de cerca de 1,5 milhões de alunos. Dada a necessidade de encontrar um aluno, você acha mais eficiente procurá-lo pela lista inteira, um por um, ou dividir essa lista ao meio e verificar se o nome buscado está na parte superior ou inferior?

A análise da complexidade do algoritmo fornece mecanismos para medir o desempenho de um algoritmo em termos de "tamanho do problema *versus* tempo de execução" (TOSCANI; VELOSO, 2012). A análise da complexidade é feita em duas dimensões: espaço e tempo. Embora ambas as dimensões influenciem na eficiência de um algoritmo, o tempo que ele leva para executar é tido como a característica mais relevante, pois, como vimos na comparação com os métodos de Cramer e Gauss, o primeiro é impossível de ser utilizado.

0

Ver anotações

Essa análise tem de ser feita sem considerar o hardware, pois sabemos que uma solução executada em um processador mais potente certamente levará menos tempo de execução do que se ocorrer em um menos potente. Não é esse tipo de medida que a análise da complexidade está interessada em estudar: o que interessa saber é qual função matemática expressa o comportamento do tempo, principalmente, para grandes entradas de dados. Por exemplo, no caso de nossa função "executar\_busca\_linear", conforme o tamanho da lista aumenta, o tempo necessário para achar um valor pode aumentar, principalmente se este estiver nas posições finais da lista.

o

Ver anotações

Essa reflexão sobre a posição do valor na lista nos leva a outros conceitos da análise da complexidade: a medida do tempo com relação ao melhor caso, ao pior caso e ao caso médio. Usando a busca linear, se o valor procurado estiver nas primeiras posições, temos o melhor cenário de tempo, independentemente do tamanho da lista, uma vez que, assim que o valor for localizado, a execução da função já se encerra. Se o valor estiver no meio da lista, temos um cenário mediano de complexidade, pois o tamanho da lista começa a ter influência no tempo da busca. Se o valor estiver no final da lista, teremos o pior caso, já que o tamanho da lista influenciará totalmente o tempo de execução.

A análise da complexidade está interessada em medir o desempenho de um algoritmo para grandes entradas, ou seja, para o pior caso (TOSCANI; VELOSO, 2012). Podemos, então, concluir que a análise da complexidade de um algoritmo tem como um dos grandes objetivos encontrar o comportamento do algoritmo (a função matemática) em relação ao tempo de execução para o pior caso, ao que chamamos de complexidade assintótica. "A complexidade assintótica é definida pelo crescimento da complexidade para entradas suficientemente grandes. O comportamento assintótico de um algoritmo é o mais procurado, já que, para um volume grande de dados, a complexidade torna-se mais importante. O algoritmo assintoticamente mais eficiente é melhor para todas as entradas, exceto talvez para entradas relativamente pequenas." (TOSCANI; VELOSO, 2012, p. 24).

A complexidade é expressa por uma função matemática  $f(n)$ , em que  $n$  é o tamanho da entrada. Para determinar a complexidade, a função é dividida em um termo dominante e termos de ordem inferior, pois estes últimos e as contantes são excluídos. Por exemplo, vamos supor que a função  $f(n) = an^2 + bn + c$  (função do segundo grau) é a função de tempo de um determinado algoritmo. Para expressar sua complexidade, identifica-se qual é o termo dominante (nesse caso, parâmetro com maior crescimento é  $n^2$ ). Ignoram-se os termos de ordem inferior (nesse caso  $n$ ) e ignoram-se as constantes ( $a, b, c$ ), razão pela qual ficamos, então com a função  $f(n) = n^2$ . Portanto, a função de complexidade para esse algoritmo é denotada pela notação assintótica  $O(n^2)$ , chamada de Big-Oh (Grande-O).

Toda essa simplificação é embasada em conceitos matemáticos (limites) que não fazem parte do escopo desta disciplina. A análise assintótica é um método usado para descrever o comportamento de limites, razão pela qual é adotada pela análise da complexidade. Para saber mais profundamente sobre essa importante área da computação, recomendamos a leitura de: CORMEN et al. **Introduction to Algorithms**. Cambridge: The MIT Press, 2001.

[Ver anotações](#)

#### EXEMPLIFICANDO

Agora que já conhecemos os conceitos básicos da análise da complexidade de algoritmos, vamos fazer a análise do algoritmo de busca sequencial, considerando nossa função *procurar\_valor*. A Figura 2.2 ilustra a análise que queremos fazer.

Figura 2.2 | Análise da complexidade da busca linear

```

1 def executar_busca_linear(lista, valor):
2     for elemento in lista:
3         if valor == elemento: ← 1 operação x N vezes
4             return True
5     return False
  
```

N vezes = tamanho da lista

$f(N) = 1 * N \rightarrow O(N)$

Fonte: elaborada pela autora.

Para isso, vamos considerar as operações em alto nível, ou seja, não vamos nos preocupar com as operações no nível de máquina. Por exemplo, na linha 4 temos uma operação de alto nível, que será executada N vezes (sendo N o tamanho da lista). Portanto, a função que determina o tempo de execução é  $f(N) = 1 * N$ . Porém, queremos representar a complexidade assintótica usando a notação Big-Oh, razão pela qual, ao excluirmos os termos de menor ordem e as constantes da equação, obtemos  $O(N)$  como a complexidade do algoritmo de busca linear (STEPHENS, 2013).

A notação  $O(N)$  representa uma complexidade linear. Ou seja, o tempo de execução aumentará de forma linear com o tamanho da entrada. Outras complexidades que são comumente encontradas são:  $O(\log N)$ ,  $O(N^2)$ ,  $O(N^3)$ . Vale ressaltar que, em termos de eficiência, teremos que:  $O(1) < O(\log N) < O(N) < O(N^2) < O(N^3) < O(2^N)$ , ou seja, um algoritmo com complexidade  $O(N)$  é mais eficiente que  $O(N^2)$ .

Ver anotações

## BUSCA BINÁRIA

Outro algoritmo usado para buscar um valor em uma sequência é o de busca binária. A primeira grande diferença entre o algoritmo de busca linear e o algoritmo de busca binária é que, com este último, os valores precisam estar ordenados. A lógica é a seguinte:

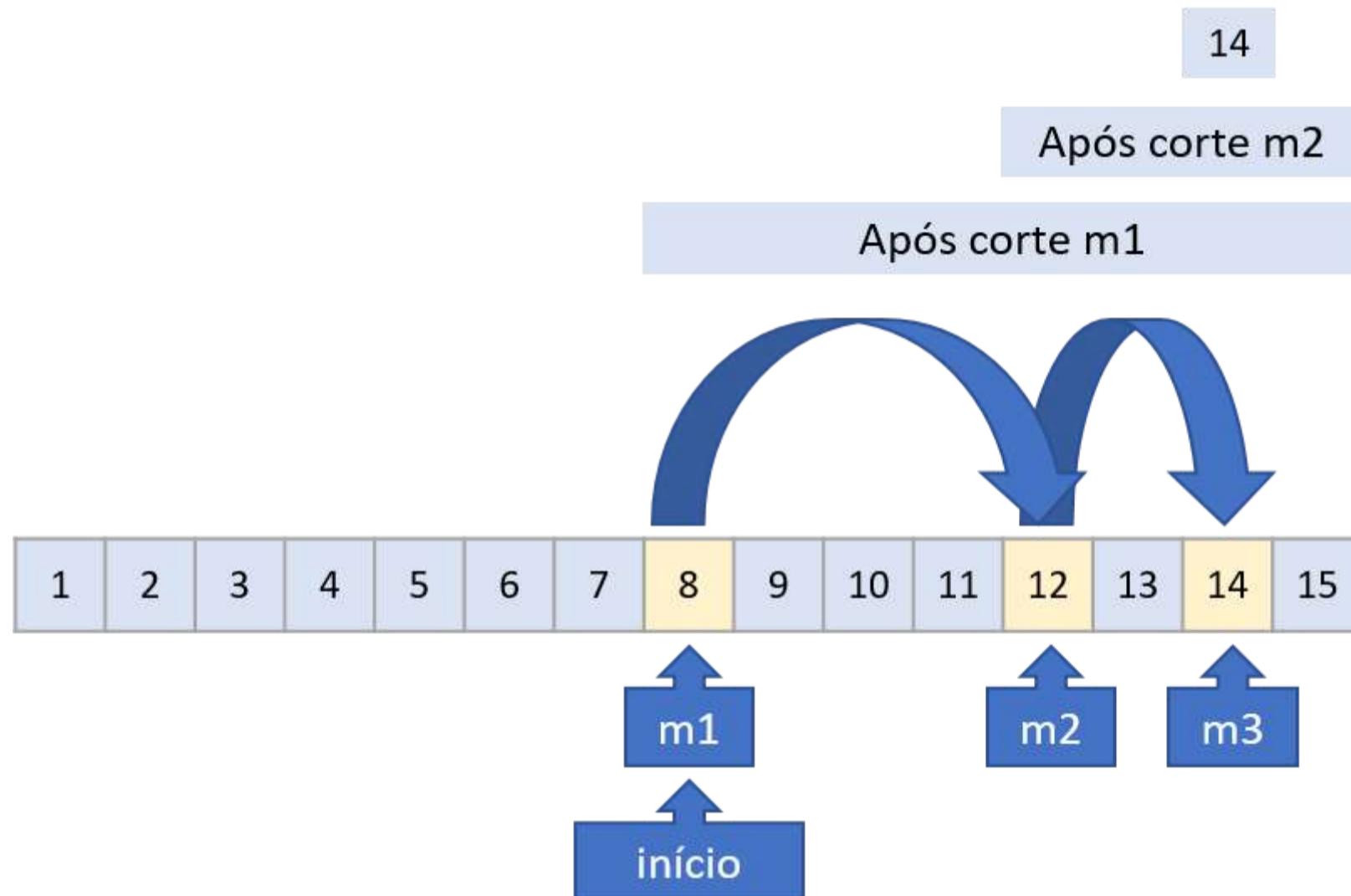
1. Encontra o item no meio da sequência (meio da lista).
2. Se o valor procurado for igual ao item do meio, a busca se encerra.
3. Se não for, verifica-se se o valor buscado é maior ou menor que o valor central.
4. Se for maior, então a busca acontecerá na metade superior da sequência (a inferior é descartada); se não for, a busca acontecerá na metade inferior da sequência (a superior é descartada).

Veja que o algoritmo, ao encontrar o valor central de uma sequência, a divide em duas partes, o que justifica o nome de busca binária. A Figura 2.3 ilustra o funcionamento do algoritmo na busca pelo número 14 em uma certa sequência numérica.

0

Ver anotações

Figura 2.3 | Busca binária



Fonte: elaborada pela autora.

Veja que o algoritmo começa encontrando o valor central, ao qual damos o nome de m1. Como o valor buscado não é o central, sendo maior que ele, então a busca passa a acontecer na metade superior. Dado o novo conjunto, novamente é localizado o valor central, o qual chamamos de m2, que também é diferente do valor buscado, sendo menor que este. Mais uma vez a metade superior é considerada e, ao localizar o valor central, agora sim trata-se do valor procurado, razão pela qual o algoritmo encerra. Veja que usamos retângulos para representar os conjuntos de dados após serem feitos os testes com os valores centrais. Como pode ser observado, cada vez mais o conjunto a ser procurado diminui.

o

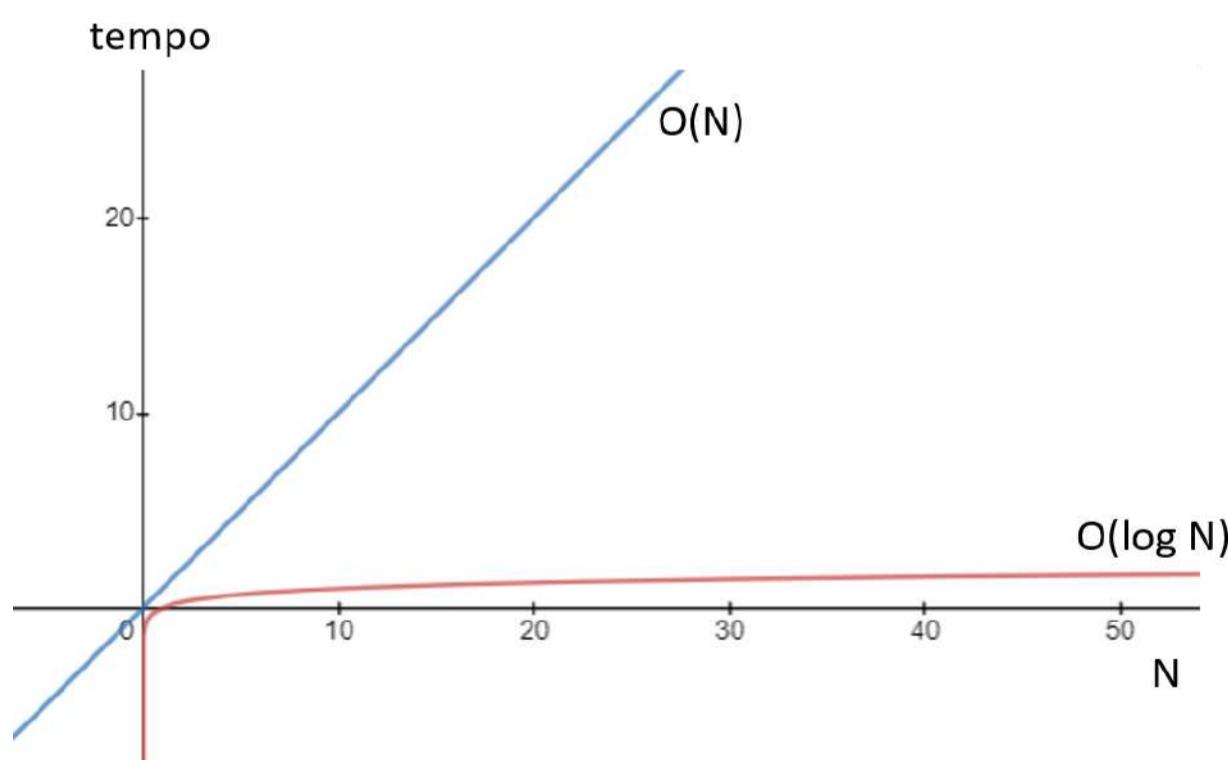
Ver anotações

Os retângulos usados na Figura 2.3 para ilustrar o conjunto de dados após encontrar o meio, checar se o valor é maior ou menor e excluir uma parte demonstram a seguinte situação:

Suponha que tenhamos uma lista com 1024 elementos. Na primeira iteração do loop, ao encontrar o meio e excluir uma parte, a lista a ser buscada já é diminuída para 512. Na segunda iteração, novamente ao encontrar o meio e excluir uma parte, restam 256 elementos. Na terceira iteração, restam 128. Na quarta, restam 64. Na quinta, restam 32. Na sexta, restam 16. Na sétima 8. Na oitava 4. Na nona Na décima iteração resta apenas 1 elemento. Ou seja, para 1024 elementos, no pior caso, o loop será executado apenas 10 vezes, diferentemente da busca linear, na qual a iteração aconteceria 1024 vezes.

A busca binária possui complexidade  $O(\log_2 N)$  (STEPHENS, 2013). Isso significa que, para valores grandes de  $N$  (listas grandes), o desempenho desse algoritmo é melhor se comparado à busca sequencial, que tem complexidade  $O(N)$ . Para ficar claro o que isso significa, observe a Figura 2.4.

Figura 2.4 | Comportamento da complexidade  $O(N)$  e  $O(\log N)$



Fonte: elaborada pela autora.

Nela construímos um gráfico com a função  $f(N) = N$ , que representa o comportamento da complexidade  $O(N)$ , e outro gráfico com a função  $f(N) = \log N$ , que representa, por sua vez,  $O(\log N)$ . Veja como o tempo da complexidade  $O(N)$

cresce muito mais rápido que  $O(\log N)$ , quando  $N$  aumenta.

Em Stephens (2013, p. 219), podemos encontrar o seguinte pseudocódigo para a busca binária:

```
0
Integer: BinarySearch(Data values[], Data target)

    Integer: min = 0

    Integer: max = - 1

    While (min <= max)

        // Find the dividing item.

        Integer: mid = (min + max) / 2

        // See if we need to search the left or right half.

        If (target < values[mid]) Then max = mid - 1

            Else If (target > values[mid]) Then min = mid + 1

            Else Return mid

    End While

    // If we get here, the target is not in the array.

    Return -1

End BinarySearch
```

Pois bem, agora que já conhecemos o algoritmo, basta escolhermos uma linguagem de programação e implementarmos. Veja como fica a busca binária em Python.

In [7]:

0

Ver anotações

```
def executar_busca_binaria(lista, valor):  
    minimo = 0  
    maximo = len(lista) - 1  
    while minimo <= maximo:  
        # Encontra o elemento que divide a lista ao meio  
        meio = (minimo + maximo) // 2  
        # Verifica se o valor procurado está a esquerda ou direita do valor  
        central  
        if valor < lista[meio]:  
            maximo = meio - 1  
        elif valor > lista[meio]:  
            minimo = meio + 1  
        else:  
            return True # Se o valor for encontrado para aqui  
    return False # Se chegar até aqui, significa que o valor não foi encontrado
```

In [8]:

```
lista = list(range(1, 50))  
  
print(lista)  
  
print('\n',executar_busca_binaria(lista=lista, valor=10))  
print('\n', executar_busca_binaria(lista=lista, valor=200))
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22,  
23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42,  
43, 44, 45, 46, 47, 48, 49]
```

```
True
```

```
False
```

Na entrada 7 (In [7]), implementamos o algoritmo da busca binária.

0

Ver anotações

Nas linhas 2 e 3, inicializamos as variáveis que contêm o primeiro e o último índice da lista. No começo da execução, esses valores são o índice 0 para o mínimo e o último índice, que é o tamanho da lista menos 1. Essas variáveis serão atualizadas dentro do loop, conforme condição.

0

Ver anotações

- Na linha 4 usamos o while como estrutura de repetição, pois não sabemos quantas vezes a repetição deverá ser executada. Esse while fará com que a execução seja feita para todos os casos binários.
- Na linha 6, usamos uma equação matemática (a média estatística) para encontrar o meio da lista.
- Na linha 8, checamos se o valor que estamos buscando é menor que o valor encontrado no meio da lista.
- Caso seja, então vamos para a linha 9, na qual atualizamos o índice máximo. Nesse cenário, vamos excluir a metade superior da lista original.
- Caso o valor não seja menor que o meio da lista, então vamos para a linha 10, na qual testamos se ele é maior. Se for, então atualizamos o menor índice, excluindo assim a metade inferior.
- Se o valor procurando não for nem menor nem maior e ainda estivermos dentro do loop, então ele é igual, e o valor True é retornado pelo comando na linha 13.
- Porém, se já fizemos todos os testes e não encontramos o valor, então é retornado False na linha 14.

Na entrada 8 (In [8]), testamos a função `executar_busca_binaria`. Veja que usamos a função `range()` para criar uma lista numérica de 50 valores ordenados. Nas linhas 5 e 6 testamos a função, procurando um valor que sabemos que existe e outro que não existe.

Como podemos alterar nossa função para que, en vez de retornar True ou False, retorne a posição que o valor ocupa da sequência? A lógica é a mesma. No entanto, agora vamos retornar a variável "meio", já que esta, se o valor for encontrado, será a posição. Observe o código a seguir.

0

In [9]:

```
def procurar_valor(lista, valor):
    minimo = 0
    maximo = len(lista) - 1
    while minimo <= maximo:
        meio = (minimo + maximo) // 2
        if valor < lista[meio]:
            maximo = meio - 1
        elif valor > lista[meio]:
            minimo = meio + 1
        else:
            return meio
    return None
```

Ver anotações

In [10]:

```
vogais = ['a', 'e', 'i', 'o', 'u']

resultado = procurar_valor(lista=vogais, valor='z')

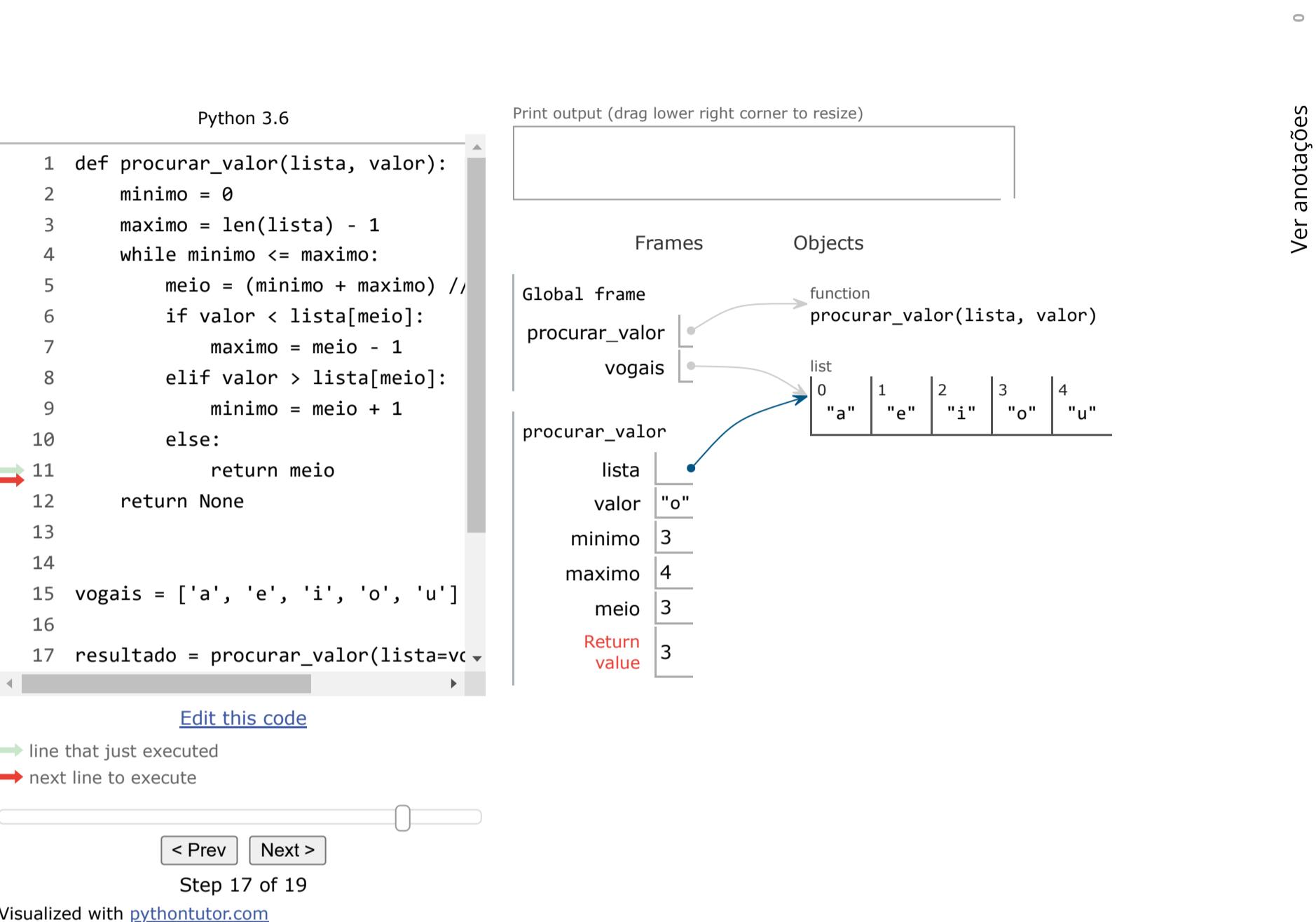
if resultado:
    print(f"Valor encontrado na posição {resultado}")
else:
    print("Valor não encontrado")
```

Valor não encontrado

Na entrada 9 (In [9]), alteramos a função *executar\_busca\_binaria* para *procurar\_valor*. Esta função fará a busca com o uso do algoritmo binário. Veja que, nas linhas 11 e 12, alteramos o retorno da função. Na entrada 10, testamos o

funcionamento do algoritmo.

Vamos buscar a vogal "o" utilizando o algoritmo de busca binária. Teste o código utilizando a ferramenta Python tutor.



The screenshot shows the Python Tutor interface with the following details:

- Python 3.6** code editor on the left containing the `procurar_valor` function and its execution steps.
- Print output** window on the right showing an empty box.
- Frames** and **Objects** panes on the right side.
- Global frame** contains the `procurar_valor` function and a `vogais` list object.
- Objects** pane shows the `vogais` list as a sequence of elements: 0 "a", 1 "e", 2 "i", 3 "o", 4 "u".
- Variables** pane shows local variables: `lista`, `valor`, `minimo`, `maximo`, `meio`, and `Return value`.
- Execution step 17 of 19 is highlighted.
- Legend: green arrow for executed line, red arrow for next line to execute.
- Navigation buttons: < Prev, Next >, Step 17 of 19.
- Footer: Visualized with [pythontutor.com](http://pythontutor.com)

Observe que é verificado se o valor procurado é igual ao valor da posição do meio da lista. Se o valor procurado é menor, o processo se repete considerando que a lista possui a metade do tamanho, razão pela qual inicia na posição seguinte do meio. Todavia, se o valor da posição for menor, o processo é repetido, considerando que a lista tem a metade do tamanho e inicia da posição anterior. No caso do teste apresentado, em que se busca a vogal "o", a metade da lista está na posição 2, correspondendo ao valor "i", caso no qual será buscado no próximo elemento a partir da metade da lista. Assim, o valor "o" é encontrado na posição 3.

Que tal utilizar o emulador a seguir para testar as funções e praticar mais?

The screenshot shows a Jupyter Notebook interface with the following elements:

- Header:** Includes icons for file operations (three horizontal lines), Python logo (3), Code, Run, Share, and Remix.
- Title Bar:** Displays "matrinxflammation\_01.csv" and "Python3".
- Code Cell:** Contains the following Python code:

```
3
< > matrinxflammation_01.csv
```
- File Selection:** A button labeled "Escolher arquivo" with the sub-label "Nenhum arquivo escolhido".
- Bottom Right:** A small "0" icon and the text "Ver anotações".

## REFERÊNCIAS E LINKS ÚTEIS

BANIN, S. L. **Python 3** - conceitos e aplicações: uma abordagem didática. São Paulo: Érica, 2018.

CORMEN et. al. **Introduction to algorithms**. 2. ed. Cambridge: The MIT Press, 2001.

PSF - Python Software Fundation. Built-in Functions. 2020d. Disponível em:  
<https://bit.ly/2XTVJmm>. Acesso em: 10 mai. 2020.

STEPHENS, R. **Essential algorithms**: a practical approach to computer algorithms. [S. l.]: John Wiley & Sons, 2013.

TOSCANI, L. V; VELOSO, P. A. S. **Complexidade de algoritmos**: análise, projeto e métodos. 3. ed. Porto Alegre: Bookman, 2012.

0

Ver anotações

# FOCO NO MERCADO DE TRABALHO

## ALGORITMOS DE BUSCA

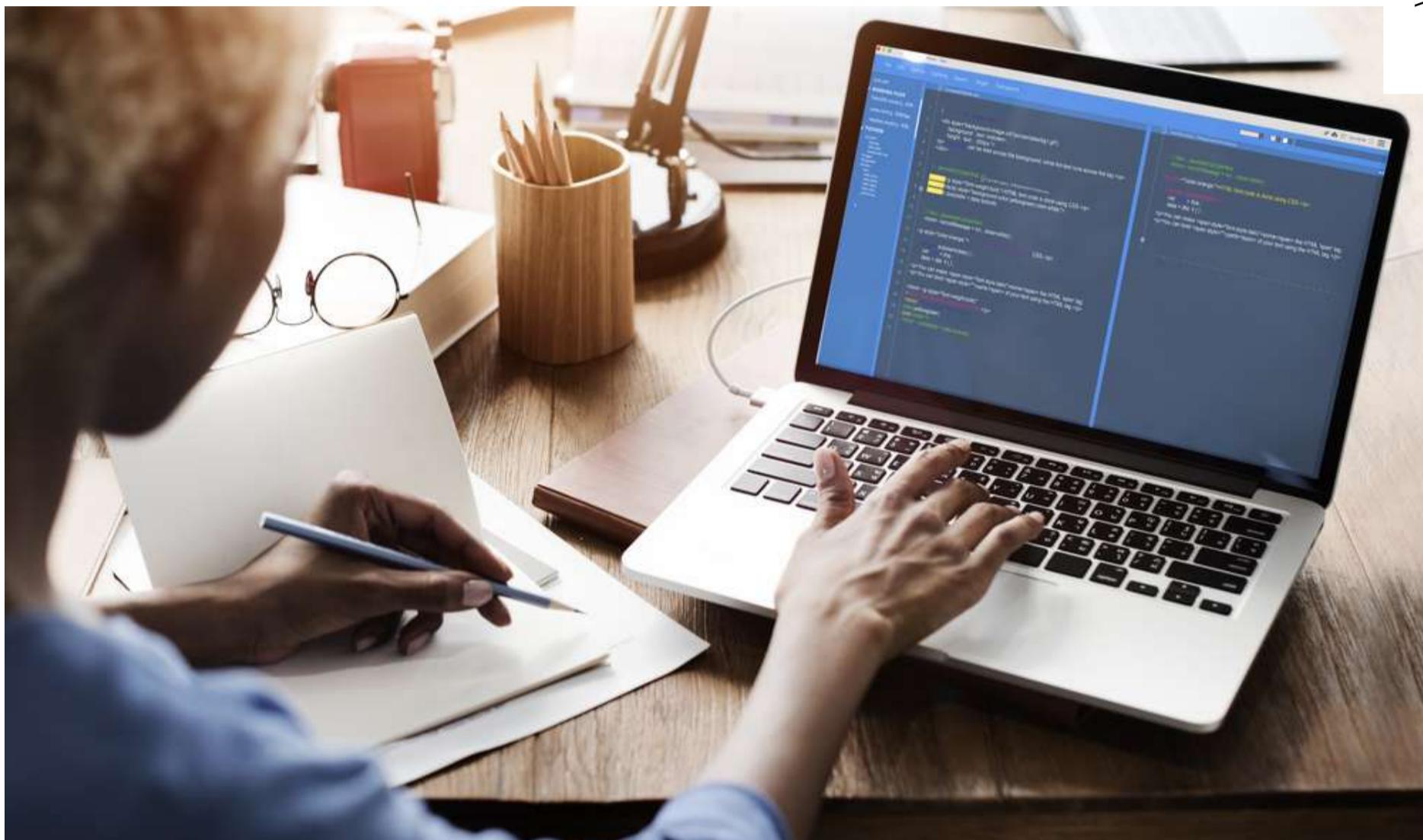
Vanessa Cadan Scheffer

0

Ver anotações

### A ESCOLHA DE UM ALGORITMO DE BUSCA

Por meio dos algoritmos de busca, podemos tratar dados de diversas maneiras.



Fonte: Shutterstock.

### Deseja ouvir este material?

Áudio disponível no material digital.

### DESAFIO

Empresas têm investido na contratação de profissionais com conhecimento na área de análise de dados. Entre as diversas especialidades que envolvem essa área, os engenheiros de dados são responsáveis por capturar, tratar e disponibilizar os dados para as áreas de análise e negócio. Entre os vários

tratamentos que precisam ser feitos nos dados, o *dedup* é um deles. *Dedup* é a abreviação para deduplicação de dados, que consiste em eliminar registros duplicados dos dados.

Como desenvolvedor em uma empresa de consultoria, você foi alocado para atender um cliente que precisa fazer a ingestão de dados de uma nova fonte e tratá-los. Alocado em uma equipe ágil, você se responsabilizou por implementar uma solução que recebe uma lista de CPFs e a retorna com a transformação de dedup. Além de fazer o dedup, também foi solicitado a você retornar somente a parte numérica do CPF e verificar se eles possuem 11 dígitos. Se não possuírem, o CPF deve ser eliminado da lista.

Como você fará o dedup usando um algoritmo de busca? É mais apropriado usar uma busca linear ou binária? Como vai remover os caracteres *ponto* e *traço* que costumam aparecer nos CPFs? Como vai verificar se os CPFs possuem 11 dígitos?

## RESOLUÇÃO

Você foi alocado em um projeto no qual precisa implementar uma função que faça a transformação de dedup em uma lista de CPFs. Além do dedup, você também precisa remover os caracteres *ponto* e *traço* do CPF, deixando somente números, e verificar se eles possuem 11 dígitos [caso contrário, o CPF não deve ser incluído na lista de CPFs válidos].

Para essa implementação, vamos utilizar um algoritmo de busca linear, uma vez que não podemos garantir que os CPFs virão ordenados, o que é uma exigência da busca binária. Poderíamos, internamente, fazer essa ordenação, mas qual seria o custo de um algoritmo de ordenação? Como não temos essa resposta, vamos adotar a busca linear. Uma vez que os CPFs podem ter números, traço e ponto, eles devem ser tratados como string, e, como vantagem, ganhamos a função *replace()*, que troca um caractere por outro. Podemos usar a função *len()* para verificar se o CPF tem 11 dígitos. Veja uma possível implementação para a solução

In [11]:

```
# Parte 1 - Implementar o algoritmo de busca linear
def executar_busca_linear(lista, valor):
    tamanho_lista = len(lista)
    for i in range(tamanho_lista):
        if valor == lista[i]:
            return True
    return False
```

0

Ver anotações

In [12]:

```
# Parte 2 - Criar a função que faz o dedup e os tratamentos no cpf
def criar_lista_dedup(lista):
    lista_dedup = []
    for cpf in lista:
        cpf = str(cpf)
        cpf = cpf.replace("-", "").replace(".", "")
        if len(cpf) == 11:
            if not executar_busca_linear(lista_dedup, cpf):
                lista_dedup.append(cpf)

    return lista_dedup
```

In [13]:

```
# Parte 3 - Criar uma função de teste
def testar_funcao(lista_cpfs):
    lista_dedup = criar_lista_dedup(lista_cpfs)
    print(lista_dedup)

lista_cpfs = ['111.111.111-11', '11111111111', '222.222.222-22', '333.333.333-33', '2222222222', '444.44444']
testar_funcao(lista_cpfs)
```

['11111111111', '2222222222', '3333333333']

Implementamos nossa solução em três etapas. As etapas 1 e 2 são, de fato, parte da solução. A etapa 3 é uma boa prática de programação, já que, para toda solução, é viável ter uma ou mais funções de teste. Na primeira parte, implementamos a função de busca linear, a qual vai receber uma lista e um valor a ser procurado. Lembre-se de que, nesse algoritmo, toda a lista é percorrida até encontrar (ou não) o valor.

o

Ver anotações

Na segunda parte da solução, implementamos a função que faz ajustar o CPF, que valida seu tamanho e que faz o dedup. Vamos entender o processo.

- Linha 3 - criamos uma estrutura de dados, do tipo lista, vazia. Essa lista armazenará os valores únicos dos CPFs.
- Linha 4 - criamos a estrutura de repetição, que percorrerá cada CPF da lista original.
- Linha 5 - fazemos o cast (conversão forçada) do CPF para o tipo string. Quando percorremos uma lista, cada elemento pode ter um tipo diferente. Convidamos você a fazer um teste: antes de fazer o cast, peça para imprimir o tipo do dado `print(type(cpf))`. Você verá que, nos casos em que o CPF só tem número, o tipo é inteiro.
- Linha 6 - usamos um encadeamento da função `replace` para substituir o traço por "nada". Quando usamos `replace("-", "")`, o primeiro elemento é o que queremos substituir e o segundo é o que queremos colocar no lugar. Nesse caso, veja que não há nada dentro das aspas, ou seja, o traço será apenas removido. O mesmo acontece para o ponto.
- Linha 7 - checamos se o tamanho do CPF é 11. Se não for, nada acontece, e o loop passa para a próxima iteração. Se for, então passamos para a linha 8.
- Linha 8 - aqui está a "mágica" do dedup. Invocamos a função de busca, passando como parâmetro a `lista_dedup` (que é a lista final que queremos) e o CPF. Essa função vai procurar, na `lista_dedup`, o valor. Caso o encontre, retornará `True`; caso não o encontre, retornará `False`. Queremos justamente os

casos falsos, pois isso quer dizer que o CPF ainda não está na lista correta, razão pela qual usamos `if not`. Caso não esteja, então passamos a linha 9.

- Linha 9 - adicionamos à `lista_dedup` o CPF, já validado, transformado e com a garantia de que não está duplicado.

Na terceira parte da solução, apenas fazemos um teste. Dada uma sequência fictícia de CPFs, veja que o resultado é exatamente o que queremos: CPFs somente numéricos sem duplicações. Veja como o algoritmo foi capaz de lidar com CPFs que contêm traços e pontos da mesma forma que o fez com os somente numéricos.

0

Ver anotações

## DESAFIO DA INTERNET

O site <https://www.hackerrank.com/> é uma ótima opção para quem deseja treinar as habilidades de programação. Nesse portal, é possível encontrar vários desafios, divididos por categoria e linguagem de programação. Na página inicial, você encontrará a opção para empresas e para desenvolvedores. Escolha a segunda e faça seu cadastro.

Após fazer o cadastro, faça login para ter acesso ao dashboard (quadro) de desafios. Navegue até a opção de algoritmos e clique nela. Uma nova página será aberta, do lado direito da qual você deve escolher o subdomínio "search" para acessar os desafios pertinentes aos algoritmos de busca. Tente resolver o desafio "Missing Numbers"!

Você deve estar se perguntando: " por que eu deveria fazer tudo isso?".

Acreditamos que o motivo a seguir pode ser bem convincente: algumas empresas utilizam o site HackerRank como parte do processo seletivo. Então, é com você!

NÃO PODE FALTAR

# ALGORITMOS DE ORDENAÇÃO

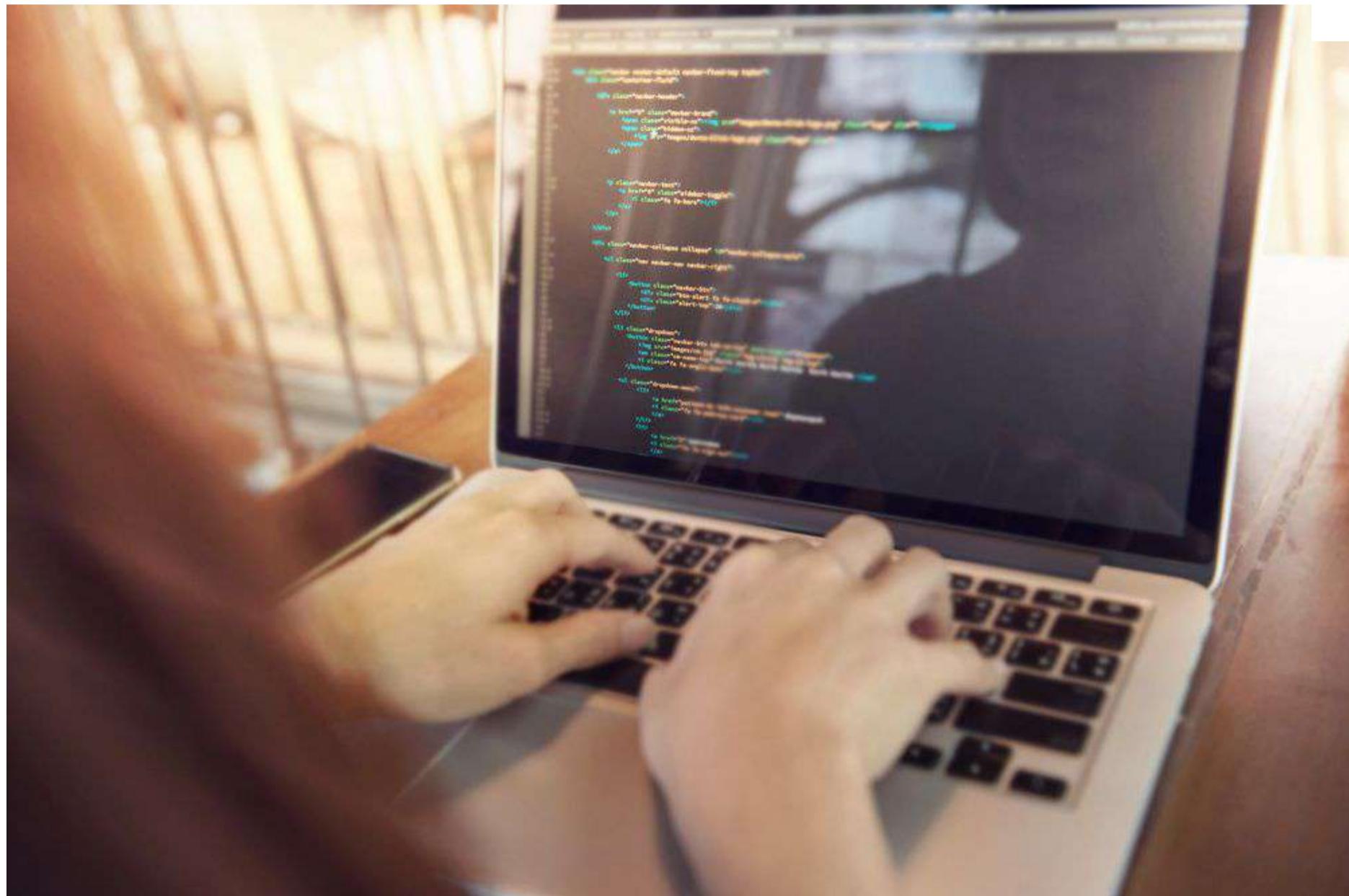
Vanessa Cadan Scheffer

0

Ver anotações

## QUAIS SÃO AS FINALIDADES DOS ALGORITMOS DE ORDENAÇÃO?

Vamos estudar esta classe de algoritmos que vai permitir explorar várias técnicas de programação em Python.



Fonte: Shutterstock.

## Deseja ouvir este material?

Áudio disponível no material digital.

## INTRODUÇÃO

"Quase todos os dados são mais úteis quando estão ordenados." (STEPHENS, 2013, p. 175)

Com essa afirmação, iniciamos esta etapa de estudos, na qual estudaremos cinco algoritmos de ordenação. Além de ser útil trabalhar com dados ordenados, estudar essa classe de algoritmos permite explorar várias técnicas de programação, tais como a recursão e o problema de dividir para conquistar. Com esse conhecimento podemos arriscar dizer que você, desenvolvedor, passará de um nível de conhecimento mais "usuário" para um nível mais técnico, mais profundo, o que lhe capacitará a entender vários mecanismos que estão por trás de funções prontas que encontramos no dia a dia.

Ver anotações

## APLICAÇÕES DOS ALGORITMOS DE BUSCA

São inúmeras as aplicações que envolvem a necessidade de dados ordenados. Podemos começar mencionando o algoritmo de busca binária, que exige que os dados estejam ordenados. No dia a dia, podemos mencionar a própria lista de chamadas, que possui dados ordenados que facilitam a localização dos nomes. Levando em consideração o *hype* da área de dados, podemos enfatizar a importância de algoritmos de ordenação, visto que alguns algoritmos de aprendizado de máquina exigem que os dados estejam ordenados para que possam fazer a previsão de um resultado, como a regressão linear.

Em Python, existem duas formas já programadas que nos permitem ordenar uma sequência: a função *built-in sorted()* e o método *sort()*, presente nos objetos da classe *list*. Observe o código a seguir.

In [1]:

```
lista = [10, 4, 1, 15, -3]

lista_ordenada1 = sorted(lista)
lista_ordenada2 = lista.sort()

print('lista = ', lista, '\n')
print('lista_ordenada1 = ', lista_ordenada1)
print('lista_ordenada2 = ', lista_ordenada2)
print('lista = ', lista)
```

```
lista = [-3, 1, 4, 10, 15]

lista_ordenada1 = [-3, 1, 4, 10, 15]
lista_ordenada2 = None
lista = [-3, 1, 4, 10, 15]
```

Na entrada 1, temos dois recursos que ordenam uma lista, mas com diferentes comportamentos (RAMALHO, 2014). A função *built-in sorted()* cria uma nova lista para ordenar e a retorna, razão pela qual, como resultado da linha 7, há uma lista ordenada, guardada dentro da variável *lista\_ordenada1*. Por sua vez, o método *sort()*, da classe *list*, não cria uma nova lista, uma vez que faz a ordenação "inplace" (ou seja, dentro da própria lista passada como parâmetro). Isso justifica o resultado obtido pela linha 8, que mostra que, dentro da variável *lista\_ordenada2*, está o valor *None* e, também, o resultado da linha 9, em que pedimos para imprimir a lista que agora está ordenada! Em Python, por padrão, quando uma função faz a alteração *inplace*, ela deve retornar o valor *None* (RAMALHO, 2014). Logo, concluímos que: 1) a função *built-in sorted()* não altera a lista original e faz a ordenação em uma nova lista; e 2) o método *sort()* faz a ordenação na lista original com retorno *None*.

o

Ver anotações

Como "programadores usuários", temos essas duas opções para ordenar uma lista em Python, certo? E como profissionais de tecnologia que entendem o algoritmo que está por trás de uma função? Para alcançarmos esse mérito, vamos então conhecer cinco algoritmos de ordenação.

A essência dos algoritmos de ordenação consiste em comparar dois valores, verificar qual é menor e colocar na posição correta. O que vai mudar, neste caso, como e quando a comparação é feita. Para que possamos começar a entender a essência dos algoritmos de ordenação, observemos o código a seguir.

0

Ver anotações

In [2]:

```
lista = [7, 4]

if lista[0] > lista[1]:
    aux = lista[1]
    lista[1] = lista[0]
    lista[0] = aux

print(lista)
```

```
[4, 7]
```

O código na entrada 2 mostra como fazer a ordenação em uma lista com dois valores. Veja, na linha 3, que a ordenação consiste em comparar um valor e seu vizinho. Caso o valor da posição mais à frente seja menor, então deve ser feita a troca de posições. Para fazer a troca, usamos uma forma mais clássica, na qual uma variável auxiliar é criada para guardar, temporariamente, um dos valores (no nosso caso estamos guardando o menor). Na linha 5, colocamos o valor maior na posição da frente e, na linha 6, resgatamos o valor da variável auxiliar colocando-a na posição correta.

A sintaxe adotada para fazer a troca no código da entrada 2 funciona para qualquer linguagem de programação. No entanto, como você já deve ter notado, em Python há "mágicas". Podemos fazer a troca usando a atribuição múltipla. Nesse caso, a atribuição é feita de maneira posicional: o primeiro valor após o símbolo de igualdade vai para a primeira variável, e assim por diante. Veja, a seguir, o mesmo exemplo, agora com uma notação pythônica.

o

Ver anotações

In [3]:

```
lista = [5, -1]

if lista[0] > lista[1]:
    lista[0], lista[1] = lista[1], lista[0]

print(lista)
```

```
[-1, 5]
```

Pois bem, ordenar significa comparar e escolher o menor. Fizemos isso para uma lista com dois valores. E se tivéssemos três deles, como faríamos a comparação e ordenação? É justamente esse "como" que dá origem aos diversos algoritmos de ordenação. Para nosso estudo, vamos dividir os algoritmos nestes dois grupos, de acordo com a complexidade:

o

Ver anotações

1. **Complexidade  $O(N^2)$ :** neste grupo estão os algoritmos *selection sort*, *bubble sort* e *insertion sort*. Esses algoritmos são lentos para ordenação de grandes listas, porém são mais intuitivos de entender e possuem uma codificação mais simples.
2. **Complexidade  $O(N \log N)$ :** deste grupo, vamos estudar os algoritmos *merge sort* e *quick sort*. Tais algoritmos possuem performance superior, porém são um pouco mais complexos de serem implementados.

## SELECTION SORT (ORDENAÇÃO POR SELEÇÃO)

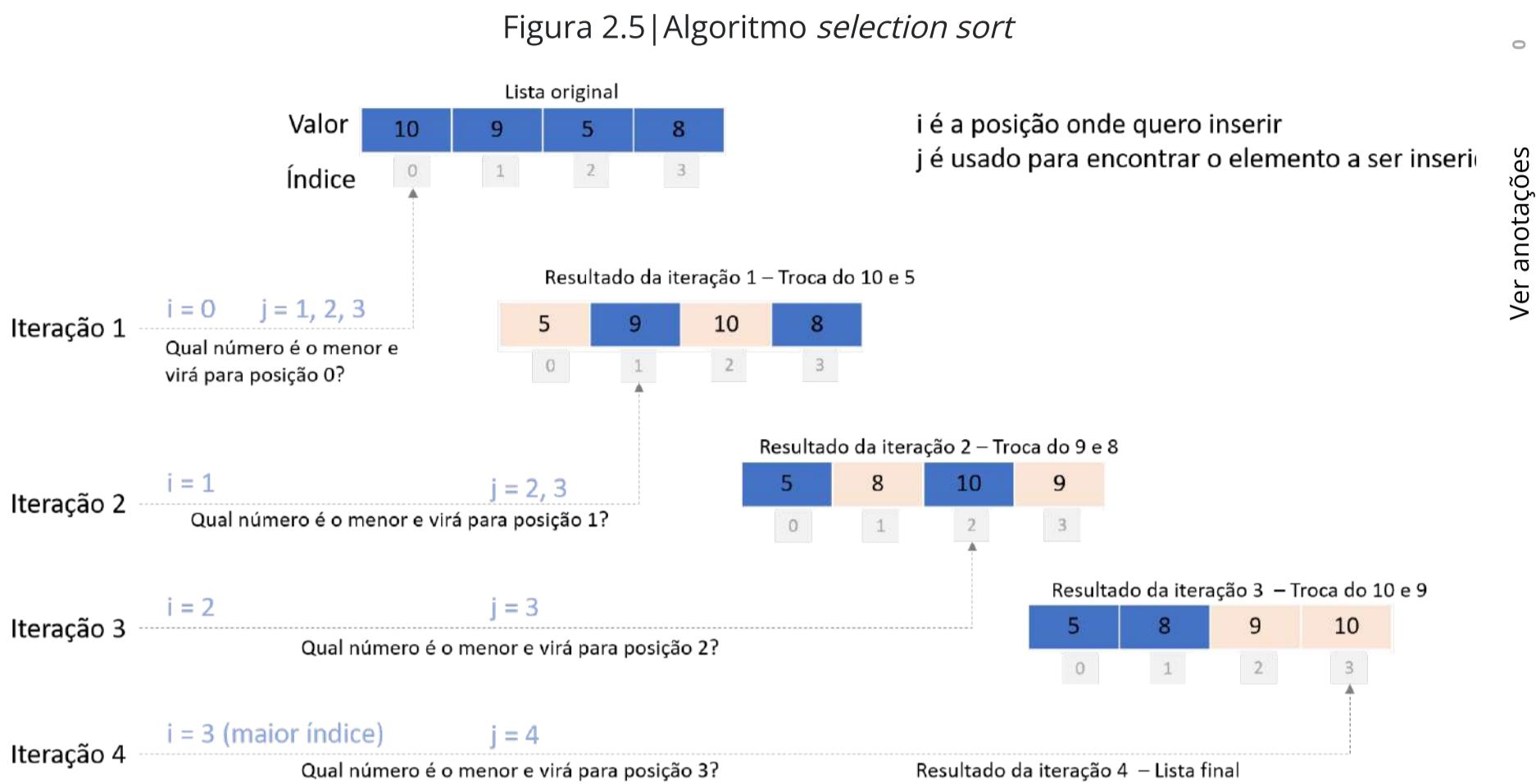
O algoritmo *selection sort* recebe esse nome, porque faz a ordenação sempre escolhendo o menor valor para ocupar uma determinada posição. Para entendermos como funciona o algoritmo, suponha que exista uma fila de pessoas, mas que, por algum motivo, elas precisem ser colocadas por ordem de tamanho, do menor para o maior. Essa ordenação será feita por um supervisor. Segundo o algoritmo *selection sort*, esse supervisor olhará para cada uma das pessoas na fila e procurará a menor delas. Quando encontrá-la, essa pessoa trocará de posição com a primeira. Veja que, agora, a primeira pessoa da fila está na posição correta, pois é a menor. Em seguida, o supervisor precisa olhar para as demais e escolher a segunda menor pessoa; quando encontrá-la, a pessoa troca de lugar com a segunda. Agora as duas primeiras pessoas da fila estão ordenadas. Esse mecanismo é feito até que todos estejam em suas devidas posições. Portanto, a lógica do algoritmo é a seguinte:

- Iteração 1: percorre toda a lista, procurando o menor valor para ocupar a posição 0.
- Iteração 2: a partir da posição 1, percorre toda a lista, procurando o menor valor para ocupar a posição 1.
- Iteração 3: a partir da posição 2, percorre toda a lista, procurando o menor valor para ocupar a posição 2.
- Esse processo é repetido  $N-1$  vezes, sendo  $N$  o tamanho da lista.

0

Ver anotações

A Figura 2.5 ilustra o funcionamento do algoritmo, ordenando a lista [10, 9, 5, 8].



Fonte: elaborada pela autora.

Vamos considerar a variável  $i$  como a posição onde queremos inserir o menor valor e usaremos  $j$  para percorrer a lista a procurar o menor valor. Na iteração 1,  $i$  é 0 (posição 0) e  $j$  precisa percorrer as posições 1, 2 e 3 comparando cada valor com o da posição 0. Como 5 é o menor valor da lista, então ele passa a ocupar a posição 0 ao passo que o valor 10 vai para a posição 2. Na segunda iteração, queremos inserir o menor valor, do restante da lista, na posição 1, razão pela qual  $j$  terá que percorrer as posições 2 e 3. Como 8 é o menor valor, então 8 passa a ocupar a posição 1; e 9, a posição 3. Na terceira iteração, queremos inserir o menor valor na posição 2, razão pela qual  $j$  precisa comparar com a última posição – como 9 é menor que 10, então acontece a troca. Após essa troca,  $i$  passa a valer 3, o maior índice, razão pela qual não há mais comparações a serem feitas.

Agora que sabemos como o algoritmo *selection sort* funciona, vamos implementá-lo na linguagem Python. Observe o código a seguir.

---

In [4]:

```
def executar_selection_sort(lista):
    n = len(lista)

    for i in range(0, n):
        index_menor = i

        for j in range(i+1, n):
            if lista[j] < lista[index_menor]:
                index_menor = j

        lista[i], lista[index_menor] = lista[index_menor], lista[i]

    return lista
```

```
lista = [10, 9, 5, 8, 11, -1, 3]
```

```
executar_selection_sort(lista)
```

Out[4]:

```
[-1, 3, 5, 8, 9, 10, 11]
```

No código da entrada 4, temos uma variável que guarda o tamanho da lista (n). Precisamos de duas estruturas de controle para iterar, tanto para ir atualizando a posição de inserção quanto para achar o menor valor da lista. Usamos a variável *i* para controlar a posição de inserção e a variável *j* para iterar sobre os valores da lista, procurando o menor valor. A busca pelo menor valor é feita com o auxílio de uma variável com a qual, quando o menor valor for encontrado, a variável *index\_menor* guardará a posição para a troca dos valores. Quando o valor na posição *i* já for o menor, então *index\_menor* não se atualiza pelo *j*. Na linha 9, usamos a atribuição múltipla para fazer a troca dos valores, quando necessário.

Para ajudar na compreensão, veja essa versão do selection sort, na qual criamos uma lista vazia e, dentro de uma estrutura de repetição, usamos a função *built-in min()* para, a cada iteração, encontrar o menor valor da sequência e adicioná-lo na lista\_ordenada. Veja que, a cada iteração, o valor adicionado à nova lista é removido da lista original.

0

Ver anotações

In [2]:

```
def executar_selection_sort_2(lista):
    lista_ordenada = []
    while lista:
        minimo = min(lista)
        lista_ordenada.append(minimo)
        lista.remove(minimo)
    return lista_ordenada
```

0

[Ver anotações](#)

```
lista = [10, 9, 5, 8, 11, -1, 3]
executar_selection_sort_2(lista)
```

Out[2]:

```
[-1, 3, 5, 8, 9, 10, 11]
```

Para enriquecer seu conhecimento e dar-lhe a oportunidade de explorar o código, fizemos uma implementação, com impressões de informação, por meio da ferramenta Python Tutor. Explore a execução, vendo-a passo a passo.

Python 3.6

Print output (drag lower right corner to resize)

```
→ 1 def executar_selection_sort(lista):
2     n = len(lista)
3
4     for i in range(0, n):
5         print('\n----- i = ',
6             index_menor = i
7             for j in range(i+1, n):
8
9                 print(f'\nlista[{j}] []')
10
11                if lista[j] < lista[index_menor]:
12                    index_menor = j
13                    print('index_menor')
14
15                lista[i], lista[index_menor]
16
17                print(f'\nLista ao final da')

```

[Edit this code](#)

→ line that just executed  
→ next line to execute

< Prev    Next >

Step 1 of 75

# BUBBLE SORT (ORDENAÇÃO POR "BOLHA")

O algoritmo bubble sort (algoritmo da bolha) recebe esse nome porque faz a ordenação sempre a partir do ínicio da lista, comparando um valor com seu vizinho. Para entendermos como funciona o algoritmo, suponha que exista uma fila de pessoas, mas que, por algum motivo, elas precisem ser colocadas por ordem de tamanho, do menor para o maior. Segundo o algoritmo bubble sort, a primeira pessoa da fila (pessoa A), perguntará para o segundo a altura – se o segundo for menor, então eles trocam de lugar. Novamente, a pessoa A perguntará para seu próximo vizinho qual é a altura deste – se esta for menor, eles trocam. Esse processo é repetido até que a pessoa A encontre alguém que é maior, contexto no qual essa nova pessoa vai percorrer a fila até o final fazendo a mesma pergunta. Esse processo é repetido até que todas as pessoas estejam na posição correta. A lógica do algoritmo é a seguinte:

- Iteração 1: seleciona o valor na posição 0 e o compara com seu vizinho – se for menor, há troca; se não for, seleciona o próximo e compara, repentinamente o

processo.

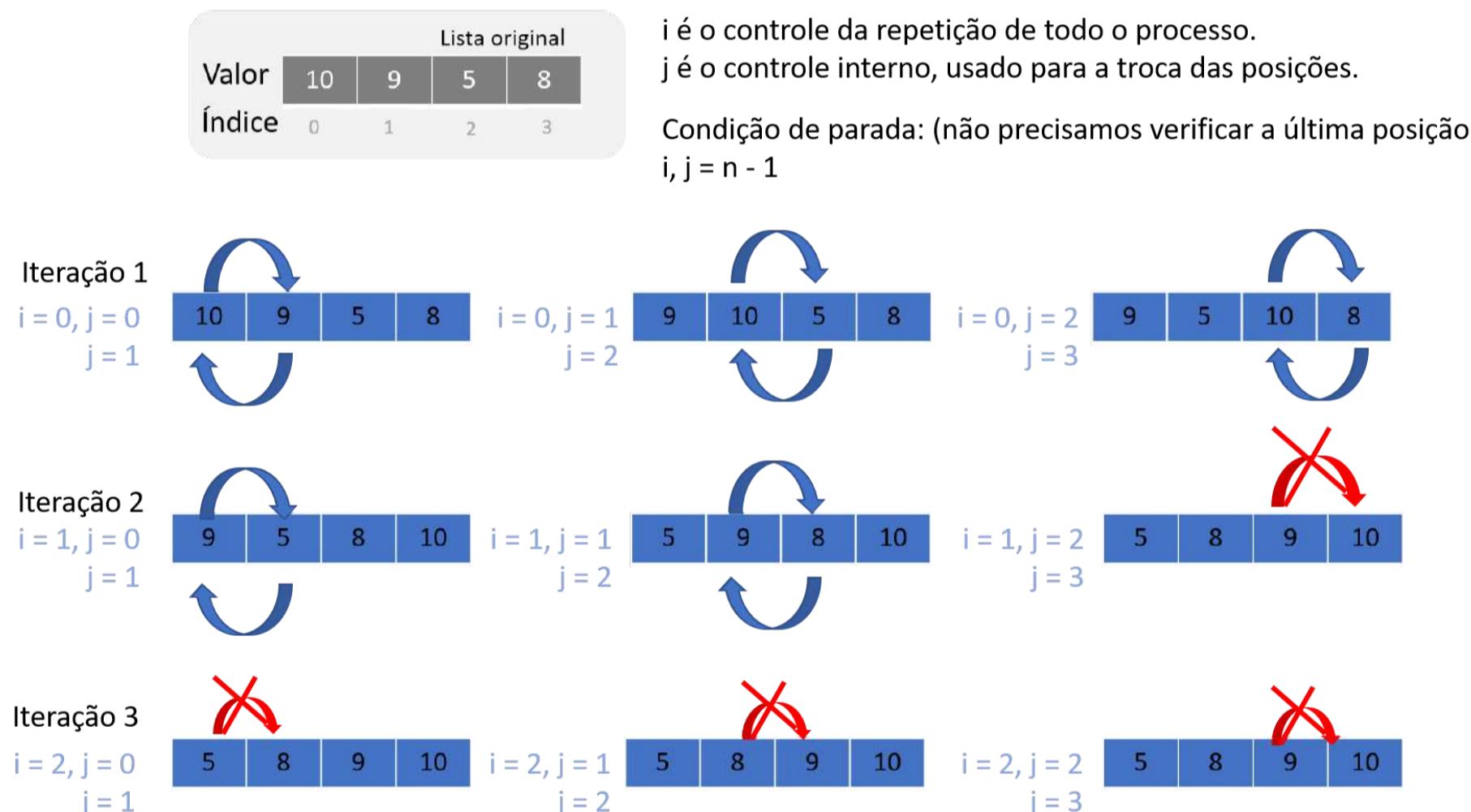
- Iteração 2: seleciona o valor na posição 0 e compara ele com seu vizinho, se for menor troca, senão seleciona o próximo e compara, repetindo o processo.
- Iteração N - 1: seleciona o valor na posição 0 e o compara com seu vizinho – se for menor, há troca; se não for, seleciona o próximo e compara, repetindo o processo.

0

[Ver anotações](#)

A Figura 2.6 ilustra o funcionando do algoritmo que ordena a lista [10, 9, 5, 8].

Figura 2.6 | Algoritmo bubble sort



Fonte: elaborada pela autora.

A variável i é usada para controlar a repetição de todo processo; e a variável j é usada para fazer as comparações entre os elementos. Na iteração 1, o valor 10 ocupará a posição 0, sendo, então, comparado ao valor da posição 1 (9) – como 10 é maior, então é feita a troca. Em seguida, o valor 10 é comparado a seu novo vizinho (5) – como é maior, é feita a troca. Em seguida, o valor 10 é comparado ao novo vizinho (8) – como é menor, é feita a troca. Em seguida, como não há mais vizinhos, a primeira iteração encerra. A segunda iteração, começa novamente com a comparação do valor das posições 0 e 1 – novamente, o valor 9 (posição 0) é maior que 5 (posição 1), então é feita a troca. Em seguida, 9 é comparado ao próximo vizinho 8 – como é maior, é feita a troca. Em seguida, 9 é comparado ao seu novo vizinho 10, como é menor, não é feita a troca. A terceira iteração começa novamente comparando o valor das posições 0 e 1, porém, agora, 5 é menor que 8, razão pela qual não se faz a troca. Na sequência, o valor 8 é comparado ao seu vizinho 9 – como é menor, não é feita a troca. Na sequência, o valor 9 é comparado ao seu vizinho 10 – como é menor, não é feita a troca, quando, então, se encerra o algoritmo.

Agora que sabemos como o algoritmo bubble sort funciona, vamos implementá-lo na linguagem Python. Observe o código a seguir.

In [5]:

```
def executar_bubble_sort(lista):
    n = len(lista)
    for i in range(n-1):
        for j in range(n-1):
            if lista[j] > lista[j + 1]:
                lista[j], lista[j + 1] = lista[j + 1], lista[j]
    return lista

lista = [10, 9, 5, 8, 11, -1, 3]
executar_bubble_sort(lista)
```

Ver anotações

Out[5]:

```
[-1, 3, 5, 8, 9, 10, 11]
```

No código da entrada 5, temos uma variável que guarda o tamanho da lista (n).

Precisamos de duas estruturas de controle para iterar, tanto para controlar a bolha, que é a quantidade de vezes que a comparação voltará para o início, quanto para controlar as comparações entre vizinhos. Usamos a variável *i* para o controle da bolha e a *j* para fazer as comparações entre vizinhos. Veja que, na linha 5, fazemos a comparação – caso o valor antecessor seja maior que o de seu sucessor, então é feita a troca de posições na linha 6; caso contrário, o valor de *j* é incremento, e uma nova comparação é feita, até que tenham sido comparados todos valores.

Para facilitar o entendimento, fizemos uma implementação, com impressões de informação, utilizando a ferramenta Python Tutor. Explore a execução, vendo-a passo a passo.

The screenshot shows the Python Tutor interface with the following details:

- Python 3.6**: The code editor window where the bubble sort algorithm is implemented.
- Print output (drag lower right corner to resize)**: A large text area where the program's output is displayed.
- Frames** and **Objects**: Buttons for navigating between different parts of the visualization.
- Code Content**:

```
1 def executar_bubble_sort(lista):
2     n = len(lista)
3     for i in range(n-1):
4
5         print('\n----- i = ', )
6
7         for j in range(n-1):
8
9             print(f'\nlista[{j}] {')
10
11            if lista[j] > lista[j + 1]:
12                print(f'troca posição')
13
14                lista[j], lista[j + 1] = lista[j + 1], lista[j]
15
16    return lista
17
```
- Execution Indicators**: A legend at the bottom left indicates:
  - Green arrow: line that just executed
  - Red arrow: next line to execute
- Navigation**: Buttons for **< Prev** and **Next >**, and a progress bar showing "Step 1 of 77".
- Footer**: The text "Visualized with [pythontutor.com](http://pythontutor.com)".

## INSERTION SORT (ORDENAÇÃO POR INSERÇÃO)

O algoritmo insertion sort (algoritmo de inserção) recebe esse nome porque faz a ordenação pela simulação da inserção de novos valores na lista. Para entender como funciona o algoritmo, imagine um jogo de cartas para a execução do qual cada jogador começa com cinco cartas e, a cada rodada, deve pegar e inserir uma nova carta na mão. Para facilitar, o jogador opta por deixar as cartas ordenadas em sua mão, razão pela qual, a cada nova carta que precisar inserir, ele olha a sequência da esquerda para a direita, procurando a posição exata para fazer a inserção. A lógica do algoritmo é a seguinte:

Ver anotações

- Início: parte-se do princípio de que a lista possui um único valor e, consequentemente, está ordenada.
- Iteração 1: parte-se do princípio de que um novo valor precisa ser inserido na lista; nesse caso, ele é comparado com o valor já existente para saber se precisa ser feita uma troca de posição.
- Iteração 2: parte-se do princípio de que um novo valor precisa ser inserido na lista; nesse caso, ele é comparado com os valores já existentes para saber se precisam ser feitas trocas de posição.
- Iteração N: parte-se do princípio de que um novo valor precisa ser inserido na lista; nesse caso, ele é comparado com todos os valores já existentes (desde o início) para saber se precisam ser feitas trocas de posição.

A Figura 2.7 ilustra o funcionamento do algoritmo que ordena a lista [10, 9, 5, 8].

Figura 2.7 | Algoritmo insertion sort

i guarda a posição do elemento que quero inserir  
j é usado para encontrar a posição a ser inserida

Parte-se do princípio que a lista tem 1 elemento e está ordenada!

Iteração 1    i = 1    j = 0

Queremos inserir o 9, ele é menor ou maior que seu antecessor?



- 1 - Guarda o 9 em uma variável auxiliar.
- 2 - Copia o 10 para a posição do 9.
- 3 - Coloca o 9 na posição certa.

Resultado da iteração 1



Iteração 2    i = 2    j = 1, 0

Queremos inserir o 5, ele é menor ou maior que seus antecessores?



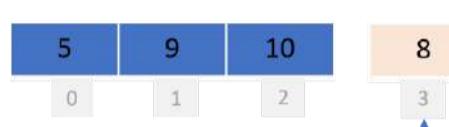
- 1 - Guarda o 5 em uma variável auxiliar.
- 2 - Copia o 10 para a posição do 5.
- 3 - Copia o 9 para posição do 10.
- 4 - Coloca o 5 na posição certa.

Resultado da iteração 2



Iteração 3    i = 3    j = 2, 1, 0

Queremos inserir o 8, ele é menor ou maior que seus antecessores?



- 1 - Guarda o 8 em uma variável auxiliar.
- 2 - Copia o 10 para a posição do 8.
- 3 - Copia o 9 para posição do 10.
- 4 - Coloca o 8 na posição certa.

Resultado da iteração 3



Fonte: Elaborada pela autora.

A variável i é usada para guardar a posição do elemento que queremos inserir, e a variável j é usada para encontrar a posição correta a ser inserida. Ou seja, j será usada para percorrer as posições já preenchidas. O primeiro valor da lista é 10, razão pela qual se parte do princípio de que a lista só possui esse valor e está ordenada. Na primeira iteração, em que o valor 9 precisa ser inserido na lista, ele deve ser inserido na posição 0 ou 1? Como é menor, é feita a troca com o 10. Na segunda iteração, queremos inserir o valor 5, razão pela qual ele é comparado a todos seus antecessores para encontrar a posição correta – como é menor que todos eles, então 5 é alocado na posição 0 e todos os demais são deslocados "para frente". Na terceira iteração, queremos inserir o valor 8, razão pela qual ele é comparado a todos seus antecessores para encontrar a posição correta e, então, é alocado para a posição 1. Todos os outros valores são deslocados "para frente".

Agora que sabemos como o algoritmo insertion sort funciona, vamos implementá-lo na linguagem Python. Observe o código a seguir.

In [6]:

Ver anotações

```
def executar_insertion_sort(lista):
    n = len(lista)
    for i in range(1, n):
        valor_inserir = lista[i]
        j = i - 1

        while j >= 0 and lista[j] > valor_inserir:
            lista[j + 1] = lista[j]
            j -= 1

        lista[j + 1] = valor_inserir

    return lista

lista = [10, 9, 5, 8, 11, -1, 3]
executar_insertion_sort(lista)
```

Out[6]:

```
[-1, 3, 5, 8, 9, 10, 11]
```

0

Ver anotações

No código da entrada 6, temos uma variável que guarda o tamanho da lista (n). Precisamos de duas estruturas de controle para iterar. Na primeira estrutura (linha 3), usamos o *for* para controlar a variável *i*, que representa a posição do valor a ser inserido. Uma vez que sabemos exatamente quantas vezes iterar, o *for* pode ser usado. Observe que o *for* começa na posição 1, pois o algoritmo parte do princípio de que a lista possui um valor e um novo precisa ser inserido. Na linha 5, inicializamos a variável *j*, com a posição anterior ao valor a ser inserido. Na linha 7 criamos a segunda estrutura de repetição com *while*, pois não sabemos quantas casas vamos ter de percorrer até encontrar a posição correta de inserção. Veja que o loop acontecerá enquanto houver elementos para comparar ( $j \geq 0$ ) e o valor da posição anterior (*lista[j]*) for maior que o valor a ser inserido. Enquanto essas condições acontecerem, os valores já existentes vão sendo "passados para frente" (linha 8) e *j* vai decrementando (linha 9). Quando a posição for encontrada, o valor é inserido (linha 10).

Para facilitar o entendimento, fizemos uma implementação, com impressões de informação, usando a ferramenta Python Tutor. Explore a execução, vendo-a passo a passo.

Python 3.6

```
→ 1 def executar_insertion_sort(lista):
    2     n = len(lista)
    3     for i in range(1, n):
    4         valor_inserir = lista[i] #
    5         j = i - 1 # calcula a posição correta
    6
    7         print(f'Inserir o {valor_inserir} na posição {j+1}')
    8
    9         while j >= 0 and lista[j] > valor_inserir:
   10             lista[j + 1] = lista[j]
   11
   12             print(f'No while: Valor {valor_inserir} inserido na posição {j+1}')
   13             j -= 1 # diminui o índice para inserir
   14             lista[j + 1] = valor_inserir
   15
   16         print(f'Pós while: posição {j+1} = {valor_inserir}')
   17         print('-----')
```

[Edit this code](#)

→ line that just executed  
→ next line to execute

< Prev    Next >

Step 1 of 59

Visualized with [pythontutor.com](https://pythontutor.com)

Ver anotações

Para todas as ordenações que fizemos até o momento, o tempo de execução foi instantâneo, pois as listas eram pequenas. E, se tivéssemos que ordenar uma lista com muitos valores, certamente perceberíamos uma lentidão na performance.

Para suprir essa necessidade, existem algoritmos que, embora performem melhor, demandam um pouco mais de complexidade na implementação. Vamos, então, conhecer os algoritmos *merge sort* e *quick sort*.

## MERGE SORT (ORDENAÇÃO POR JUNÇÃO)

O algoritmo merge sort recebe esse nome porque faz a ordenação em duas etapas: (i) divide a lista em sublistas; e (ii) junta (merge) as sublistas já ordenadas. Esse algoritmo é conhecido por usar a estratégia "dividir para conquistar" (STEPHENS, 2013). Essa estratégia é usada por algoritmos de estrutura recursiva: para resolver um determinado problema, eles se chamam recursivamente uma ou mais vezes para lidar com subproblemas intimamente relacionados. Esses algoritmos geralmente seguem uma abordagem de dividir e conquistar: eles dividem o problema em vários subproblemas semelhantes ao problema original, mas menores em tamanho, resolvem os subproblemas recursivamente e depois combinam essas soluções para criar uma solução para o problema original (CORMEN *et al.*, 2001). O paradigma de dividir e conquistar envolve três etapas em cada nível da recursão: (i) dividir o problema em vários subproblemas; (ii) conquistar os subproblemas, resolvendo-os recursivamente – se os tamanhos dos subproblemas forem pequenos o suficiente, apenas resolva os subproblemas de maneira direta; (iii) combine as soluções dos subproblemas na solução do problema original.

Ver anotações

- Etapa de divisão:
  1. Com base na lista original, encontre o meio e separe-a em duas listas: esquerda\_1 e direita\_2.
  2. Com base na sublista *esquerda\_1*, se a quantidade de elementos for maior que 1, encontre o meio e separe-a em duas listas: *esquerda\_1\_1* e *direta\_1\_1*.
  3. Com base na sublista *esquerda\_1\_1*, se a quantidade de elementos for maior que 1, encontre o meio e separe-a em duas listas: *esquerda\_1\_2* e *direta\_1\_2*.
  4. Repita o processo até encontrar uma lista com tamanho 1.
  5. Chame a etapa de merge.
  6. Repita o processo para todas as sublistas.

Para entender a etapa de junção do merge sort, suponha que você está vendendo duas fileiras de crianças em ordem de tamanho. Ao olhar para a primeira criança de cada fila, é possível identificar qual é a menor e, então, colocá-la na posição correta. Fazendo isso sucessivamente, teremos uma fila ordenada com base nas duas anteriores.

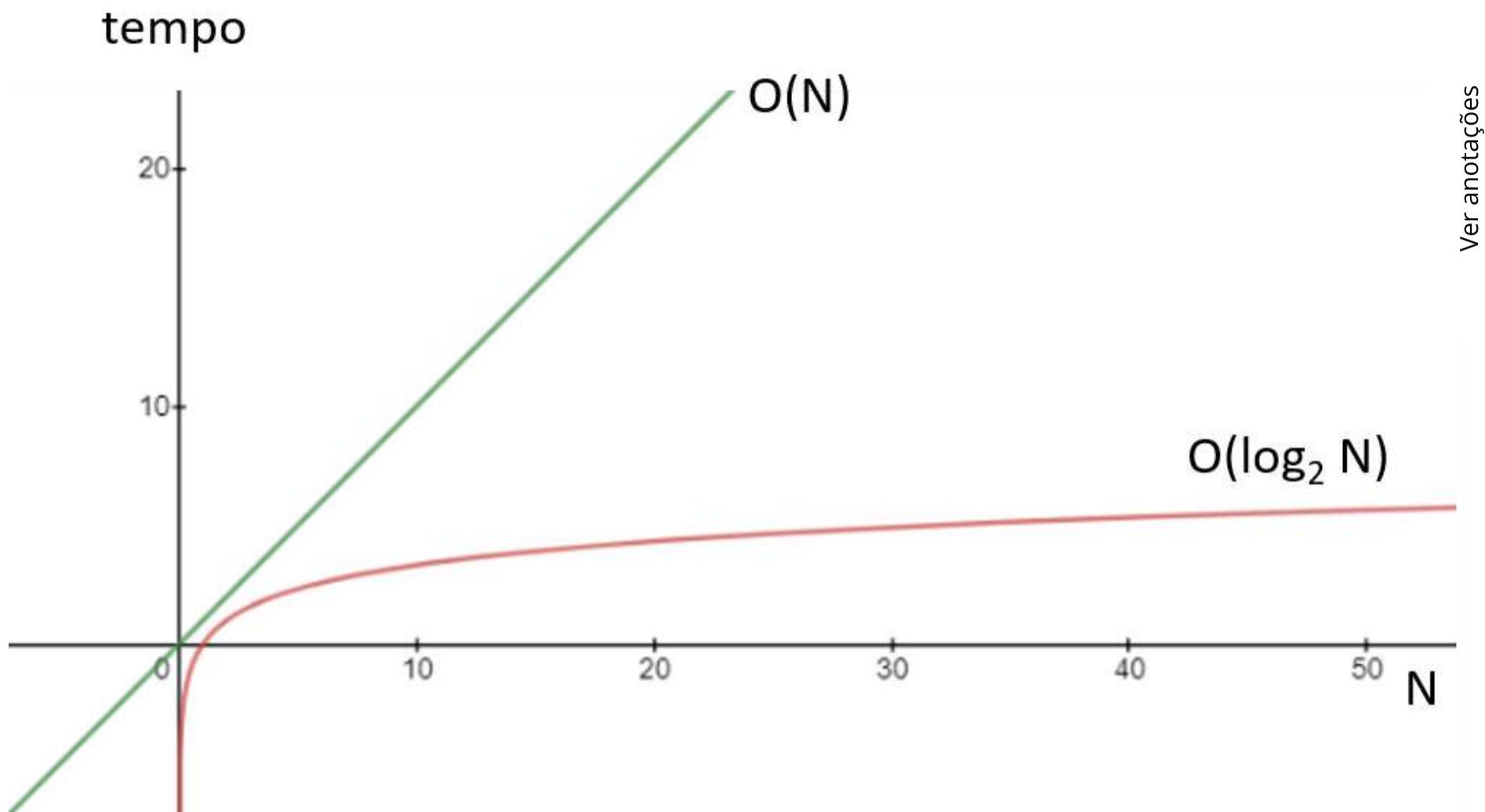
- Etapa de merge:
  1. Dadas duas listas, cada uma das quais contém 1 valor – para ordenar, basta comparar esses valores e fazer a troca, gerando uma sublista com dois valores ordenados.
  2. Dadas duas listas, cada uma das quais contém 2 valores – para ordenar, basta ir escolhendo o menor valor em cada uma e gerar uma sublista com quatro valores ordenados.
  3. Repita o processo de comparação e junção dos valores até chegar à lista original, agora ordenada

0

[Ver anotações](#)

A Figura 2.8 ilustra o funcionamento do algoritmo, ordenando a lista [10, 9, 5, 8].

Figura 2.8 | Algoritmo merge sort



Fonte: elaborada pela autora.

Começando pela etapa de divisão, é encontrado o meio da lista e é feita uma divisão, o que resulta nas sublistas [10, 9] do lado esquerdo e [5, 8] do lado direito. Como as sublistas possuem mais que um valor, então é feita uma quebra novamente – a sublista da esquerda gera duas novas listas [10] e [9]; e a sublista da direita, as novas listas [5] e [8]. Alcançado o tamanho mínimo da lista, é hora de começar o processo de merge, que fará a junção de forma ordenada. Começando pela esquerda, as listas [10] e [9] são comparadas, gerando a sublista [9, 10]. Do outro lado, as listas [5] e [8] são comparadas, gerando a sublista [5, 8]. Agora é hora de fazer o novo merge entre essas sublistas de tamanho 2. Para facilitar a compreensão, podemos pensar nessas listas como pilhas ordenadas, no topo das quais está o menor valor de cada uma. Então, ao olhar para o topo de duas pilhas, vemos os valores 9 e 5 – como 5 é menor, ele é o primeiro escolhido para ocupar a posição 0. Olhando novamente para essa pilha, agora temos no topo os valores 9 e 8 – como 8 é menor, então ele é o segundo escolhido para compor a nova lista.

Olhando novamente, agora temos somente uma pilha, em cujo topo está o valor 9, razão pela qual ele é escolhido e, por fim, o valor 10 é selecionado. Como não existem mais merges para serem feitos, o algoritmo encerra.

Agora que sabemos como o algoritmo merge sort funciona, vamos implementá-lo na linguagem Python. Como já mencionamos, esse algoritmo possui mais complexidade de código. Primeiro ponto: vamos precisar de duas funções, uma que divide e outra que junta. Segundo ponto: a divisão é feita de maneira lógica, seja, as sublistas são "fatiamentos" da lista original. O algoritmo de merge vai sempre receber a lista inteira, mas tratará de posições específicas. Terceiro ponto: na etapa de divisão, serão feitas sucessivas subdivisões aplicando-se a mesma regra, tarefa para cuja realização vamos usar a técnica de recursão, fazendo chamadas recursivas a função de divisão. Caso você não se lembre do funcionamento das funções recursivas, recomendamos os seguintes vídeos:

- No canal Me Salva!, em vídeo de 5 minutos: *Programação em C - PLC20 - Recursão*, a partir do minuto 1:23 é exibido um exemplo clássico da recursão. A partir do minuto 2:54, é explicado como a função recursiva se comporta, fazendo chamadas sucessivas até encontrar o caso base, que dará o retorno para as funções. O vídeo está disponível no endereço <https://bit.ly/3ijla9N>.
- No canal Bóson Treinamentos, em vídeo de 16 minutos: *29 - Curso de Python - Funções - Recursividade*, é feita a explicação da técnica de recursão, além de serem feitas duas implementações, uma com e outra sem recursão, para o mesmo problema. O vídeo está disponível no endereço <https://bit.ly/3dNib5T>.

Agora que já relembramos a técnica de recursão, observe o código a seguir, que implementa o merge sort.

In [7]:

0

Ver anotações

```
def executar_merge_sort(lista):
    if len(lista) <= 1: return lista
    else:
        meio = len(lista) // 2
        esquerda = executar_merge_sort(lista[:meio])
        direita = executar_merge_sort(lista[meio:])
        return executar_merge(esquerda, direita)
```

0

Ver anotações

```
def executar_merge(esquerda, direita):
    sub_lista_ordenada = []
    topo_esquerda, topo_direita = 0, 0
    while topo_esquerda < len(esquerda) and topo_direita < len(direita):
        if esquerda[topo_esquerda] <= direita[topo_direita]:
            sub_lista_ordenada.append(esquerda[topo_esquerda])
            topo_esquerda += 1
        else:
            sub_lista_ordenada.append(direita[topo_direita])
            topo_direita += 1
    sub_lista_ordenada += esquerda[topo_esquerda:]
    sub_lista_ordenada += direita[topo_direita:]
    return sub_lista_ordenada
```

```
lista = [10, 9, 5, 8, 11, -1, 3]
executar_merge_sort(lista)
```

Out[7]:

```
[-1, 3, 5, 8, 9, 10, 11]
```

Na entrada 7, criamos nossa função de ordenação por merge sort, que, na verdade, são duas funções, uma para dividir as listas (`executar_merge_sort`) e outra para fazer o merge (`executar_merge`). A função que faz a divisão recebe como parâmetro a lista a ser ordenada. Na linha 2, se o tamanho da lista é menor ou igual 1, isso significa que a sublista só tem 1 valor e está ordenada, razão pela qual

seu valor é retornado; caso não seja, então é encontrado o meio da lista e feita a divisão entre sublistas da direita e da esquerda. Esse processo é feito recursivamente até que se tenha sublistas de tamanho 1.

A função de junção, ao receber duas listas, percorre cada uma delas pelo while na linha 13, e, considerando cada valor, o que for menor é adicionado à sublista ordenada.

A fim de facilitar o entendimento, utilize a ferramenta Python Tutor para explorar execução, vendo-a passo a passo.

The screenshot shows the Python Tutor interface with the following details:

- Python 3.6**: The code editor pane containing the merge sort implementation.
- Frames**: A pane showing the execution frames, with the first frame visible.
- Objects**: A pane showing the state of objects at each step of the execution.
- Code:**

```
1 def executar_merge_sort(lista):
2     if len(lista) <= 1: return lista
3     else:
4         meio = len(lista) // 2
5         esquerda = executar_merge_s
6         direita = executar_merge_s
7         return executar_merge(esque
8
9
10 def executar_merge(esquerda, direit
11     sub_lista_ordenada = []
12     topo_esquerda, topo_direita = 0, 0
13     while topo_esquerda < len(esque
14         if esquerda[topo_esquerda] <
15             sub_lista_ordenada.append(
16                 esquerda[topo_esquerda]
17             topo_esquerda += 1
18         else:
19             sub_lista_ordenada.append(
20                 direita[topo_direita]
21             topo_direita += 1
22     return sub_lista_ordenada
```
- Execution Status:** Step 1 of 167.
- Annotations:** A green arrow points to the line just executed (line 13), and a red arrow points to the next line to execute (line 14).
- Buttons:** < Prev, Next >, Edit this code.
- Legend:** line that just executed (green arrow), next line to execute (red arrow).
- Footer:** Visualized with [pythontutor.com](https://pythontutor.com).

## QUICKSORT (ORDENAÇÃO RÁPIDA)

Dado um valor em uma lista ordenada, à direita desse número existem somente números maiores que ele; e à esquerda, somente os menores. Esse valor, chamado de pivô, é a estratégia central no algoritmo *quicksort*. O algoritmo quicksort também trabalha com a estratégia de dividir para conquistar, pois, a partir do pivô, quebrará uma lista em sublistas (direita e esquerda) – a cada

escolha do pivô e a cada quebra da lista, o processo de ordenação vai acontecendo. Para entendermos como o algoritmo funciona, suponha uma fila de pessoas que, por algum motivo, precisam ser ordenadas por ordem de tamanho. Pede-se para uma pessoa da fila se voluntariar para ter seu tamanho comparado (esse é o pivô). Com base nesse voluntário, todos que são menores que ele devem se dirigir para a esquerda e todos que são maiores para a direita. O voluntário está na posição ordenada. Vamos repetir o mesmo procedimento para os menores e maiores. A cada passo estamos dobrando o número de pessoas na posição final. Lógica é a seguinte:

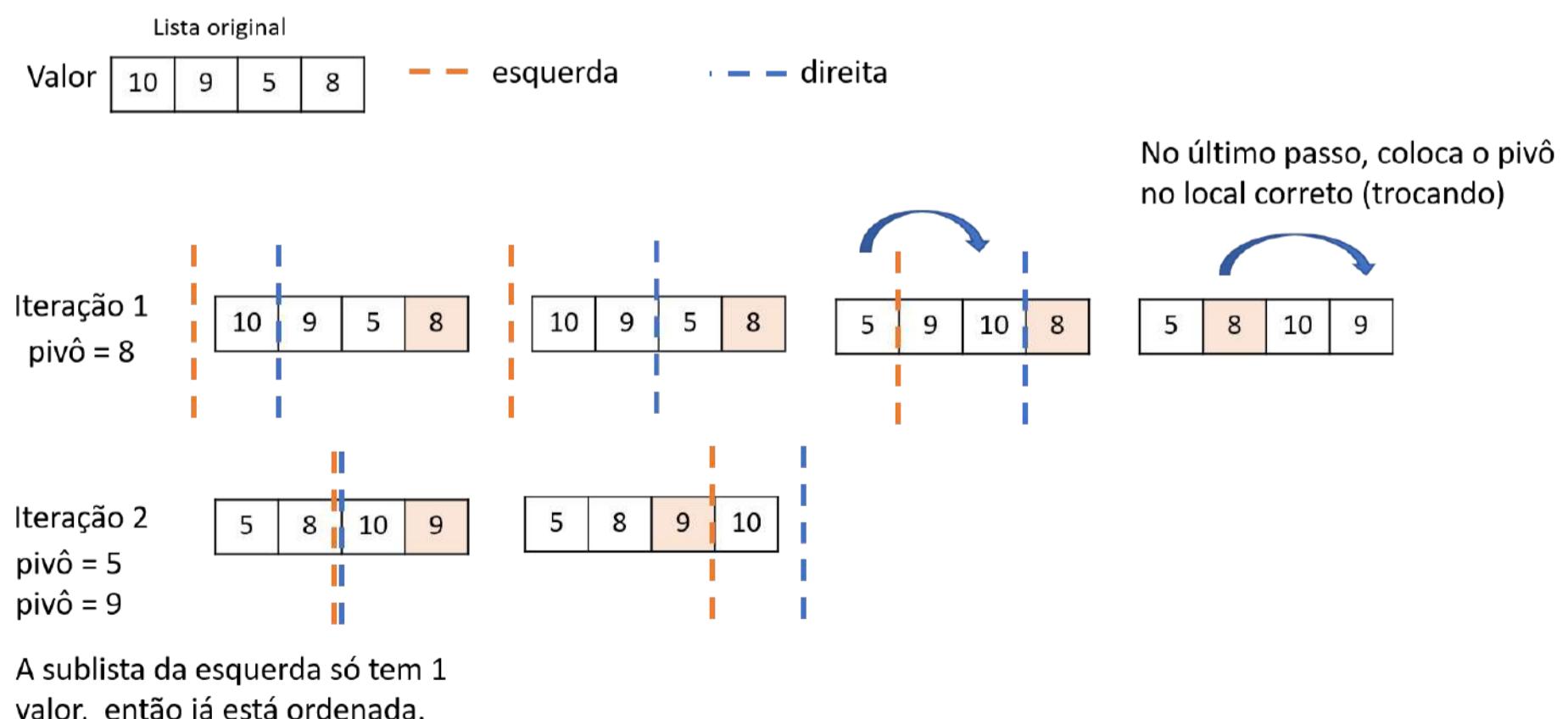
o

Ver anotações

1. Primeira iteração: a lista original será quebrada através de um valor chamado de pivô. Após a quebra, os valores que são menores que o pivô devem ficar à sua esquerda e os maiores à sua direita. O pivô é inserido no local adequado, trocando a posição com o valor atual.
2. Segunda iteração: agora há duas listas, a da direita e a da esquerda do pivô. Novamente são escolhidos dois novos pivôs e é feito o mesmo processo, de colocar à direita os menores e à esquerda os maiores. Ao final os novos pivôs ocupam suas posições corretas.
3. Terceira iteração: olhando para as duas novas sublistas (direita e esquerda), repete-se o processo de escolha dos pivôs e separação.
4. Na última iteração, a lista estará ordenada, como resultado dos passos anteriores.

Para nosso exemplo e implementação, vamos escolher o pivô como sempre o último valor da lista e das sublistas.

Figura 2.9 | Algoritmo quicksort



Fonte: elaborada pela autora.

o

Ver anotações

A Figura 2.9 ilustra o funcionamento do algoritmo quicksort. Dada uma lista a ser ordenada [10, 9, 5, 8], na primeira iteração o pivô é escolhido, o qual, no nosso caso, é o valor 8. Agora, para cada valor da lista, o pivô será comparado e acontecerá uma separação da lista em duas sublistas: à esquerda do pivô, os valores menores; e à direita, os valores maiores. No passo 1, movemos a linha que representa a sublista da direita, pois 10 é maior que o pivô. No passo 2, também movemos essa linha, pois 9 é maior que o pivô. No passo 3, movemos a linha da direita (porque sempre vamos fazer a comparação com os elementos da frente), mas movemos também a linha da esquerda, visto que 5 é menor que o pivô. Veja que, nesse movimento, os valores das posições são trocados: 5 passa a ocupar a posição da linha da esquerda, e o valor que ali estava vai para a posição da direita. Como não há mais elementos para comparar, uma vez que alcançamos o pivô, então ele é colocado no seu devido lugar, que está representado pela linha da esquerda. Para essa inserção, é feita a troca do valor que ali está com o valor do pivô. Veja que 8 passa a ocupar a posição 1, ao passo que 9 vai para a posição do pivô. Agora, todo o processo precisa se repetir, considerando-se, agora, as sublistas da esquerda e da direita. Na esquerda, temos uma sublista de tamanho 1 (valor 5), razão pela qual não há nada a ser feito. Por sua vez, na direita, temos a sublista [10, 9], razão pela qual se adota 9 como o pivô dela. Como o pivô é menor que o primeiro elemento dessa sublista, então o marcado da direita avança, mas o da esquerda também, fazendo a troca dos valores. Como não há mais comparações a serem feitas, o algoritmo encerra com a lista ordenada.

Agora que sabemos como o algoritmo quicksort funciona, vamos implementá-lo na linguagem Python. Assim como o merge sort, o quicksort será implementado em duas funções, uma vez que o mesmo processo vai se repetir várias vezes. Observe o código a seguir.

In [8]:

```
def executar_quicksort(lista, inicio, fim):
    if inicio < fim:
        pivo = executar_particao(lista, inicio, fim)
        executar_quicksort(lista, inicio, pivo-1)
        executar_quicksort(lista, pivo+1, fim)
    return lista

def executar_particao(lista, inicio, fim):
    pivo = lista[fim]
    esquerda = inicio
    for direita in range(inicio, fim):
        if lista[direita] <= pivo:
            lista[direita], lista[esquerda] = lista[esquerda], lista[direita]
            esquerda += 1
    lista[esquerda], lista[fim] = lista[fim], lista[esquerda]
    return esquerda

lista = [10, 9, 5, 8, 11, -1, 3]
executar_quicksort(lista, inicio=0, fim=len(lista)-1)
```

Out[8]:

```
[-1, 3, 5, 8, 9, 10, 11]
```

Na entrada 8 implementamos nosso quicksort em duas funções:

*executar\_quicksort* e *executar\_particao*. A função *executar\_quicksort* é responsável por criar as sublistas, cada uma das quais, no entanto, deve ser criada com base em um pivô. Por isso, caso a posição de início da lista seja menor que o fim (temos mais que 1 elemento), então é chamada a função *executar\_particao*, que de fato faz a comparação e, quando necessário, troca os valores de posição, além de retornar o índice correto para o pivô. Na linha 10, fazemos a definição do pivô como o último valor da lista (e mesmo da sublista). Na linha 11, criamos a variável que controla a separação da lista da esquerda, ou seja, a lista que guardará os valores menores que o pivô. Usamos a estrutura de repetição para ir comparando

0

Ver anotações

o pivô com todos os valores da lista à direita. A cada vez que um valor menor que o pivô é encontrado (linha 13), é feita a troca dos valores pelas posições (linha 14), e a delimitação da lista dos menores (esquerda) é atualizada (linha 15). Na linha 16, o pivô é colocado na sua posição (limite da lista esquerda), fazendo a troca com o valor que ali está. Por fim, a função retorna o índice do pivô.

o

Ver anotações

Para ajudar na compreensão do quicksort, veja esta outra implementação do algoritmo, na qual usamos list comprehensions para criar uma lista de pivôs (agora o pivô é o primeiro valor da lista), uma lista com os valores menores e uma com os valores maiores. Esse processo é chamado recursivamente até que a lista esteja ordenada.

In [2]:

```
def executar_quicksort_2(lista):
    if len(lista) <= 1: return lista
    pivo = lista[0]
    iguais = [valor for valor in lista if valor == pivo]
    menores = [valor for valor in lista if valor < pivo]
    maiores = [valor for valor in lista if valor > pivo]
    return executar_quicksort_2(menores) + iguais +
executar_quicksort_2(maiores)

lista = [10, 9, 5, 8, 11, -1, 3]
executar_quicksort_2(lista)
```

Out[2]:

```
[-1, 3, 5, 8, 9, 10, 11]
```

Utilize a ferramenta Python Tutor e explore a execução, observando o resultado a cada passo.

0

Ver anotações

Python 3.6
Frames
Objects

---

```

→ 1 def executar_quicksort_2(lista):
  2     if len(lista) <= 1: return lista
  3     pivo = lista[0]
  4     iguais = [valor for valor in li:
  5     menores = [valor for valor in li:
  6     maiores = [valor for valor in li:
  7     return executar_quicksort_2(menor
  8
  9
10 lista = [10, 9, 5, 8, 11, -1, 3]
11 executar_quicksort_2(lista)

```

<
▶

[Edit this code](#)

→ line that just executed  
→ next line to execute

< Prev
Next >

Step 1 of 140

Visualized with [pythontutor.com](#)

[Move and hide objects](#)

Para finalizar nossa aula, vale ressaltar que a performance de cada algoritmo vai depender da posição do valor na lista e do seu tamanho. Por exemplo, no algoritmo *insertion sort*, dado que o lado esquerdo é conhecido (e ordenado), se o valor a ser inserido é grande, o algoritmo terá uma boa performance. Por outro lado, o algoritmo *selection sort*, uma vez que verifica o mínimo para frente, que não se conhece, sempre está no pior caso.

## REFERÊNCIAS E LINKS ÚTEIS

BANIN, S. L. **Python 3** - conceitos e aplicações: uma abordagem didática. São Paulo: Érica, 2018.

CORMEN, T. *et al.* **Introduction to algorithms**. Cambridge: The MIT Press, 2001.

KHAN ACADEMY. Algoritmos de divisão e conquista. Google Sala de Aula, 2020.

Disponível em: <https://bit.ly/3is319Z>. Acesso em: 2 jul. 2020.

PSF - Python Software Foundation. Built-in Functions. 2020d. Disponível em: <https://bit.ly/2Bwp1zw>. Acesso em: 10 mai. 2020.

RAMALHO, L. **Fluent Python**. Gravenstein: O'Reilly Media, 2014.

STEPHENS, R. Essential algorithms: a practical approach to computer algorithms. [S. I.]: John Wiley & Sons, 2013.

TOSCANI, L. V; VELOSO, P. A. S. **Complexidade de algoritmos**: análise, projeto e métodos. 3. ed. Porto Alegre: Bookman, 2012.

0

Ver anotações

# FOCO NO MERCADO DE TRABALHO

## ALGORITMOS DE ORDENAÇÃO

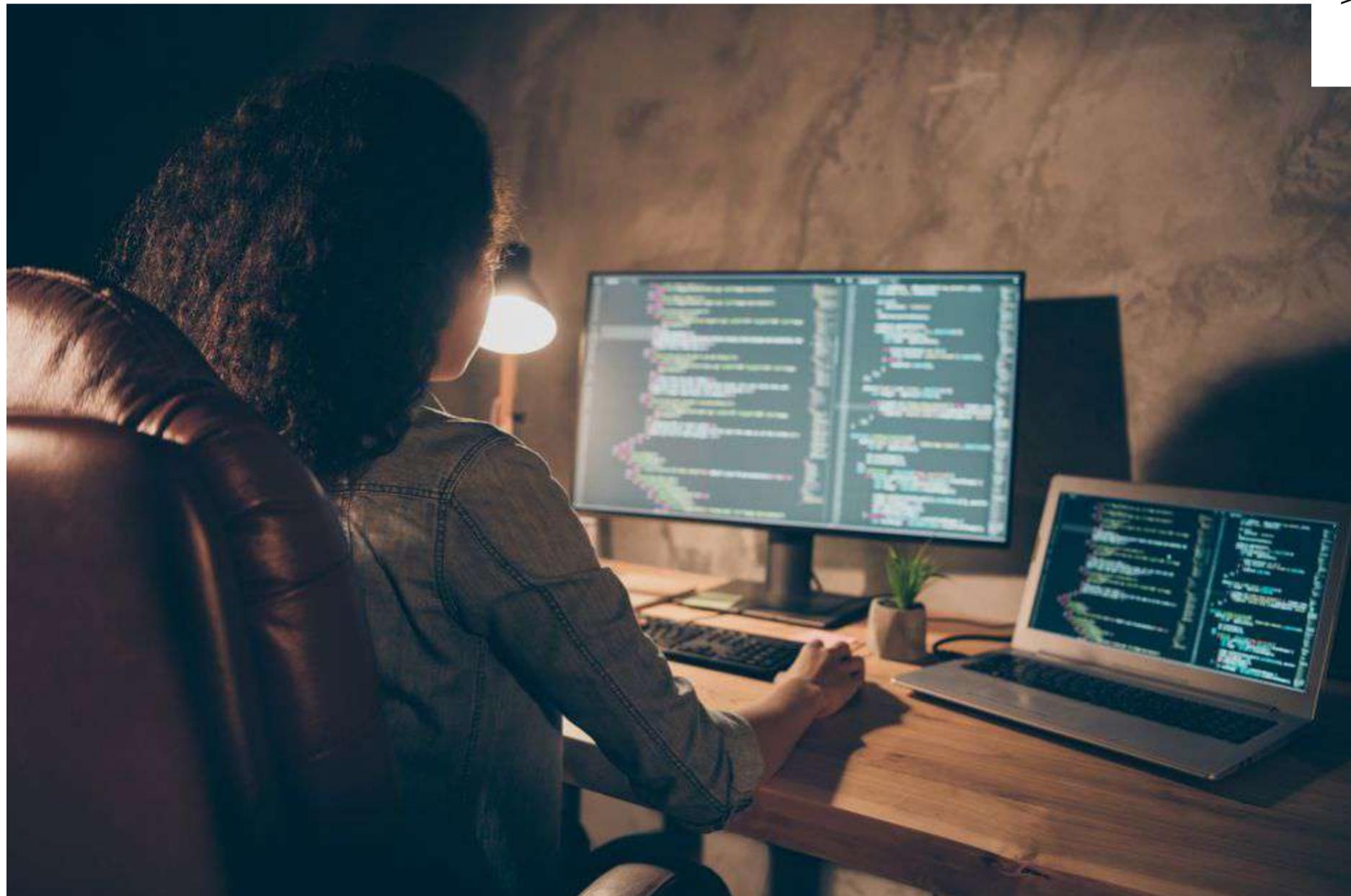
Vanessa Cadan Scheffer

0

Ver anotações

### COMO APRESENTAR OS DADOS DE FORMA ORDENADA?

Implementando algoritmos de ordenação.



Fonte: Shutterstock.

### Deseja ouvir este material?

Áudio disponível no material digital.

### DESAFIO

Como desenvolvedor em uma empresa de consultoria, você continua alocado para atender um cliente que precisa fazer a ingestão de dados de uma nova fonte e tratá-las. Você já fez uma entrega na qual implementou uma solução que faz o

dedup em uma lista de CPFs, retorna somente a parte numérica do CPF e verifica se eles possuem 11 dígitos.

Os clientes aprovaram a solução, mas solicitaram que a lista de CPFs válidos fosse entregue em ordem crescente para facilitar o cadastro. Eles enfatizaram a necessidade de ter uma solução capaz de fazer o trabalho para grandes volumes de dados, no melhor tempo possível. Uma vez que a lista de CPFs pode crescer exponencialmente, escolher os algoritmos mais adequados é importante nesse caso.

0

Ver anotações

Portanto, nesta nova etapa, você deve tanto fazer as transformações de limpeza e validação nos CPFs (remover ponto e traço, verificar se tem 11 dígitos e não deixar valores duplicados) quanto fazer a entrega em ordem crescente. Quais algoritmos você vai escolher para implementar a solução? Você deve apresentar evidências de que fez a escolha certa!

## RESOLUÇÃO

Alocado em um novo desafio, é hora de escolher e implementar os melhores algoritmos para a demanda que lhe foi dada. Consultando a literatura sobre algoritmos de busca, você encontrou que a busca binária performa melhor que a busca linear, embora os dados precisem estar ordenados. No entanto, agora você já sabe implementar algoritmos de ordenação!

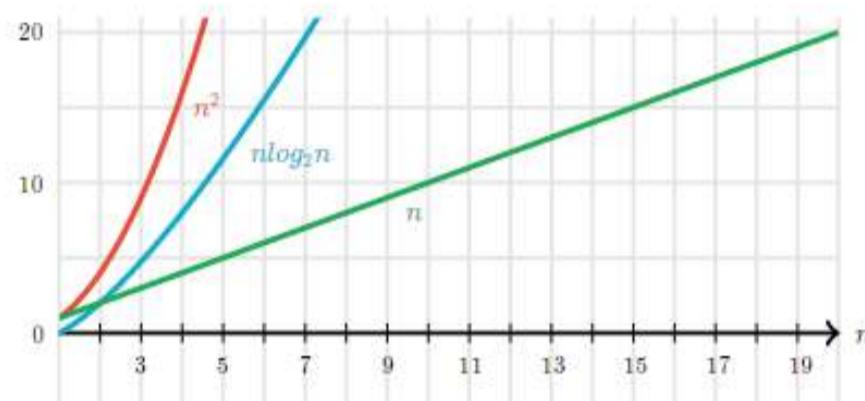
O cliente enfatizou que a quantidade de CPFs pode aumentar exponencialmente, o que demanda algoritmos mais eficazes. Ao pesquisar sobre a complexidade dos algoritmos de ordenação, você encontrou no portal Khan Academy (2020) um quadro comparativo (Figura 2.10) dos principais algoritmos. Na Figura 2.10, fica evidente que o algoritmo de ordenação *merge sort* é a melhor opção para o cliente, visto que, para o pior caso, é o que tem menor complexidade de tempo.

Figura 2.10 | Comparaçāo de algoritmos de ordenaçāo

Algoritmo	Tempo de execução no pior caso	Tempo de execução no melhor caso	Tempo de execução médio
ordenação por seleção (selection sort)	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Insertion sort	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n^2)$
Ordenação por combinação (merge sort)	$\Theta(n \lg n)$	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)$	$\Theta(n \lg n)$

0

Ver anotações



Fonte: Khan Academy (2020, [s.p.]).

Agora que já decidiu quais algoritmos implementar, é hora de colocar a mão na massa!

In [9]:

```
# Parte 1 - Implementar o algoritmo de ordenação merge sort
```

```
def executar_merge_sort(lista, inicio=0, fim=None):
```

```
    if not fim:
```

```
        fim = len(lista)
```

```
    if fim - inicio > 1:
```

```
        meio = (inicio + fim) // 2
```

```
        executar_merge_sort(lista, inicio, meio)
```

```
        executar_merge_sort(lista, meio, fim)
```

```
        executar_merge(lista, inicio, meio, fim)
```

```
    return lista
```

```
def executar_merge(lista, inicio, meio, fim):
```

```
    esquerda = lista[inicio:meio]
```

```
    direita = lista[meio:fim]
```

```
    topo_esquerda = topo_direita = 0
```

```
    for p in range(inicio, fim):
```

```
        if topo_esquerda >= len(esquerda):
```

```
            lista[p] = direita[topo_direita]
```

```
            topo_direita += 1
```

```
        elif topo_direita >= len(direita):
```

```
            lista[p] = esquerda[topo_esquerda]
```

```
            topo_esquerda += 1
```

```
        elif esquerda[topo_esquerda] < direita[topo_direita]:
```

```
            lista[p] = esquerda[topo_esquerda]
```

```
            topo_esquerda += 1
```

```
        else:
```

```
            lista[p] = direita[topo_direita]
```

```
            topo_direita += 1
```

0

Ver anotações

In [10]:

```
# Parte 2 - Implementar o algoritmo de busca binária
```

```
def executar_busca_binaria(lista, valor):  
    minimo = 0  
    maximo = len(lista) - 1  
    while minimo <= maximo:  
        meio = (minimo + maximo) // 2  
        if valor < lista[meio]:  
            maximo = meio - 1  
        elif valor > lista[meio]:  
            minimo = meio + 1  
        else:  
            return True  
    return False
```

0

Ver anotações

In [11]:

```
# Parte 3 - Implementar a função que faz a verificação do cpf, o dedup e devolve  
o resultado esperado
```

```
def criar_lista_dedup_ordenada(lista):  
    lista = [str(cpf).replace('.', '').replace('-', '') for cpf in lista]  
    lista = [cpf for cpf in lista if len(cpf) == 11]  
    lista = executar_merge_sort(lista)  
  
    lista_dedup = []  
    for cpf in lista:  
        if not executar_busca_binaria(lista_dedup, cpf):  
            lista_dedup.append(cpf)  
    return lista_dedup
```

In [12]:

```
# Parte 4 - Criar uma função de teste

def testar_funcao(lista_cpfs):
    lista_dedup = criar_lista_dedup_ordenada(lista_cpfs)
    print(lista_dedup)

lista_cpfs = ['444444444444', '111.111.111-11', '11111111111', '222.222.222-22'
              '333.333.333-33', '2222222222', '444.44444']
testar_funcao(lista_cpfs)
['11111111111', '2222222222', '3333333333', '44444444444']
```

0

Ver anotações

Implementamos os algoritmos de ordenação (merge sort) e de busca (binária), conforme aprendemos nas aulas. No algoritmo de ordenação, fazemos um tratamento na variável `fim` para que não precise ser passada explicitamente na primeira chamada. Vamos focar na função `criar_lista_dedup_ordenada`. Na linha 4, usamos uma list comprehension para remover o ponto e o traço dos CPFs. Na linha 5, criamos novamente uma list comprehension, agora para guardar somente os CPFs que possuem 11 dígitos. Em posse dos CPFs válidos, chamamos a função de ordenação. Agora que temos uma lista ordenada, podemos usar a busca binária para verificar se o valor já está na lista; caso não estiver, então ele é adicionado. Na quarta parte da nossa solução, implementamos uma função de teste para checar se não há nenhum erro e se a lógica está correta.

## DESAFIO DA INTERNET

O site <https://www.hackerrank.com/> é uma ótima opção para quem deseja treinar as habilidades de programação. Nesse portal, é possível encontrar vários desafios, divididos por categoria e linguagem de programação. Na página inicial, você encontra a opção para empresas e para desenvolvedores. Escolha a segunda e faça seu cadastro.

Após fazer o cadastro, faça login para ter acesso ao dashboard (quadro) de desafios. Navegue até a opção de algoritmos e clique nela. Uma nova página será aberta, do lado direito da qual você deve escolher o subdomínio "**sort**" para acessar os desafios pertinentes aos algoritmos de busca. Tente resolver o desafio "Insertion Sort - Part 1"!

Você deve estar se perguntando, por que eu deveria fazer tudo isso? Acredito que o motivo a seguir pode ser bem convincente: algumas empresas utilizam o site Hacker Rank como parte do processo seletivo. Então, é com você!

o

Ver anotações

NÃO PODE FALTAR

## CLASSES E MÉTODOS EM PYTHON

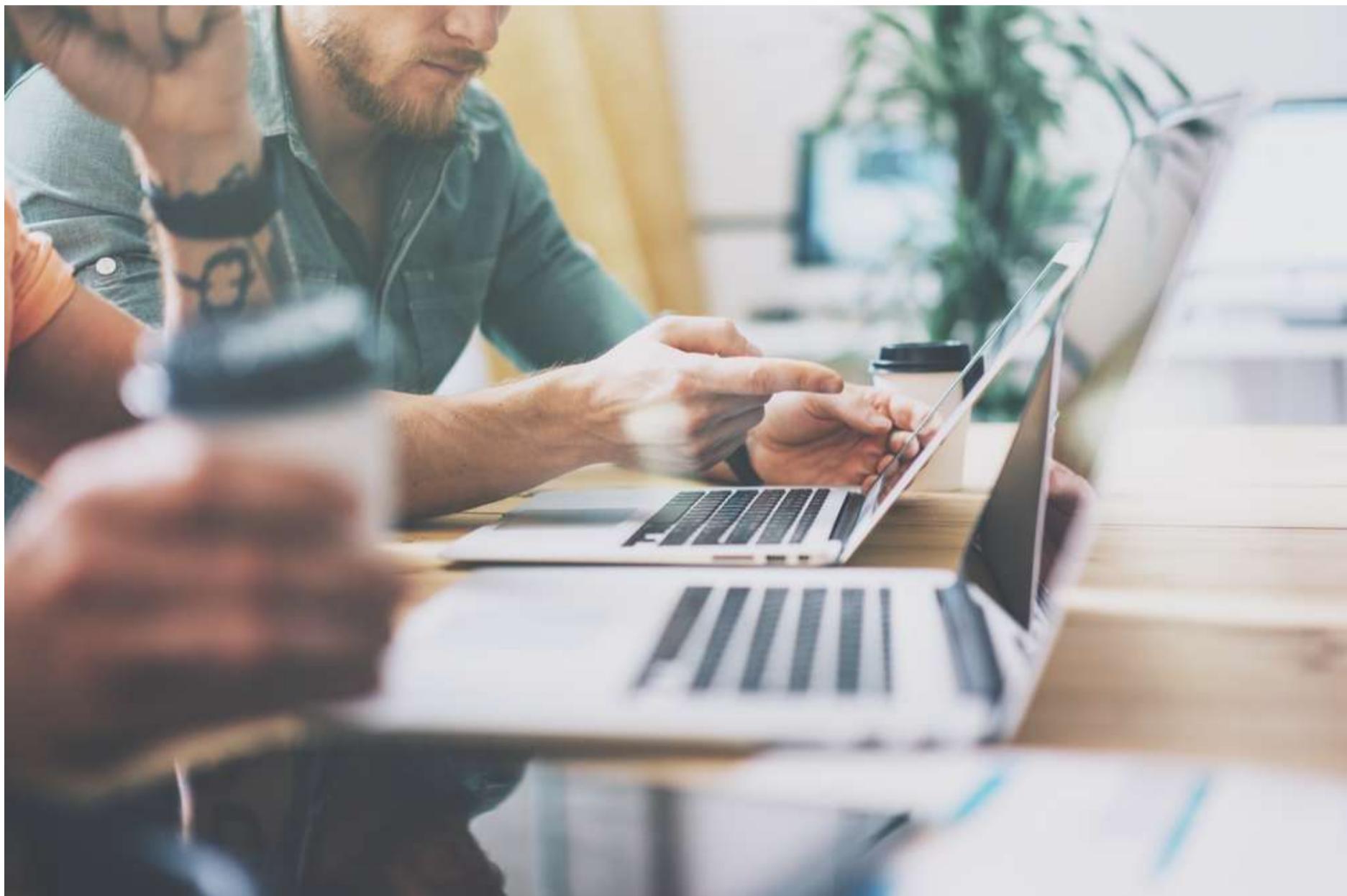
Vanessa Cadan Scheffer

0

Ver anotações

### O QUE SÃO OBJETOS E O QUE AS CLASSES TÊM A VER COM ELES?

Uma classe é uma abstração que descreve entidades do mundo real e, quando instanciadas, dão origem a objetos com características similares. Portanto, a classe é o modelo e o objeto é uma instância.



Fonte: Shutterstock.

**Deseja ouvir este material?**

Áudio disponível no material digital.

INTRODUÇÃO A ORIENTAÇÃO A OBJETOS

Vamos começar esta seção com alguns conceitos básicos do paradigma de programação orientado a objetos. Como nos lembra Weisfeld (2013), as tecnologias mudam muito rapidamente na indústria de software, ao passo que os conceitos evoluem. Portanto, é preciso conhecer os conceitos para então implementá-los na tecnologia adotada pela empresa (e essa tecnologia pode mudar rapidamente).

0

Ver anotações

O desenvolvimento de software orientado a objetos (OO) existe desde o início dos anos 1960, mas, segundo Weisfeld (2013), foi somente em meados da década de 1990 que o paradigma orientado a objetos começou a ganhar impulso. De acordo com o mesmo autor, uma linguagem é entendida como *orientada a objetos* se ela aplica o conceito de abstração e suporta a implementação do encapsulamento, da herança e do polimorfismo. Mas, afinal, o que são objetos e o que as classes têm a ver com eles?

## | ABSTRAÇÃO - CLASSES E OBJETOS

**Objetos** são os componentes de um programa OO. Um programa que usa a tecnologia OO é basicamente uma coleção de objetos. Uma **classe** é um modelo para um objeto. Podemos considerar uma classe uma forma de organizar os dados (de um objeto) e seus comportamentos (PSF, 2020a). Vamos pensar na construção de uma casa: antes do "objeto casa" existir, um arquiteto fez a planta, determinando tudo que deveria fazer parte daquele objeto. Portanto, a **classe** é o modelo e o **objeto** é uma instância. Entende-se por instância a existência física, em memória, do objeto.

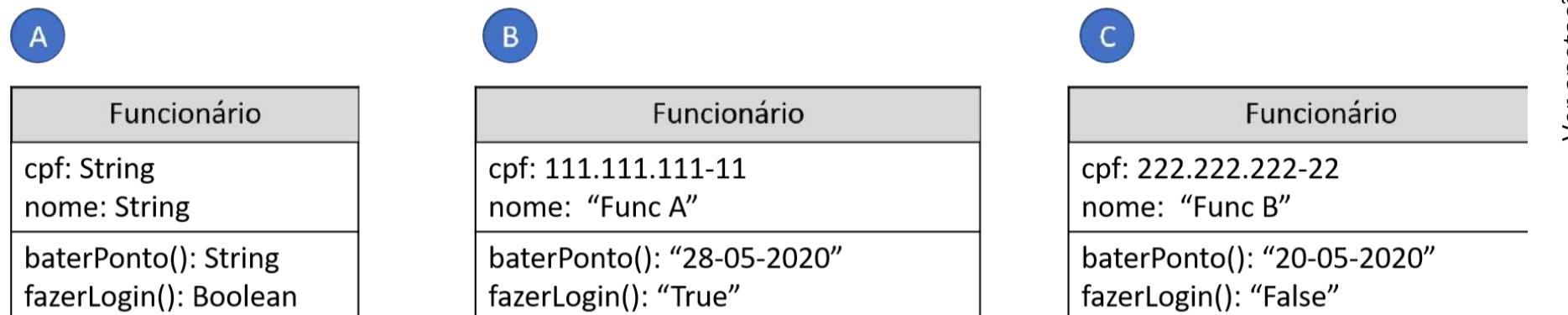
Para compreendermos a diferença entre classe e objeto, observe a Figura 3.1. Usamos a notação gráfica oferecida pela UML (*Unified Modeling Language*) para criarmos um diagrama de classe. Cada diagrama de classes é definido por três seções separadas: o próprio nome da classe, os dados e os comportamentos. Na Figura 3.1(A), em que temos a classe *funcionário*, são especificados o que um funcionário deve ter. No nosso caso, como dados, ele deve ter um CPF e um nome

e, como comportamento, ele deve bater ponto e fazer login. Agora veja a Figura 3.1(B) e a Figura 3.1(C) – esses dados estão "preenchidos", ou seja, foram instanciados e, portanto, são objetos.

0

Ver anotações

Figura 3.1 - Diagrama de classe: classe *versus* objeto



Fonte: elaborada pela autora.

Com base na Figura 3.1, conseguimos definir mais sobre as classes e objetos: eles são compostos por dados e comportamento.

## ATRIBUTOS

Os dados armazenados em um objeto representam o estado do objeto. Na terminologia de programação OO, esses dados são chamados de **atributos**. Os atributos contêm as informações que diferenciam os vários objetos – os funcionários, neste caso.

## MÉTODOS

O comportamento de um objeto representa o que este pode fazer. Nas linguagens procedurais, o comportamento é definido por procedimentos, funções e subrotinas. Na terminologia de programação OO, esses comportamentos estão contidos nos **métodos**, aos quais você envia uma mensagem para invocá-los.

## ENCAPSULAMENTO

O ato de combinar os atributos e métodos na mesma entidade é, na linguagem OO, chamado de **encapsulamento** (Weisfeld, 2013), termo que também aparece na prática de tornar atributos privados, quando estes são encapsulados em

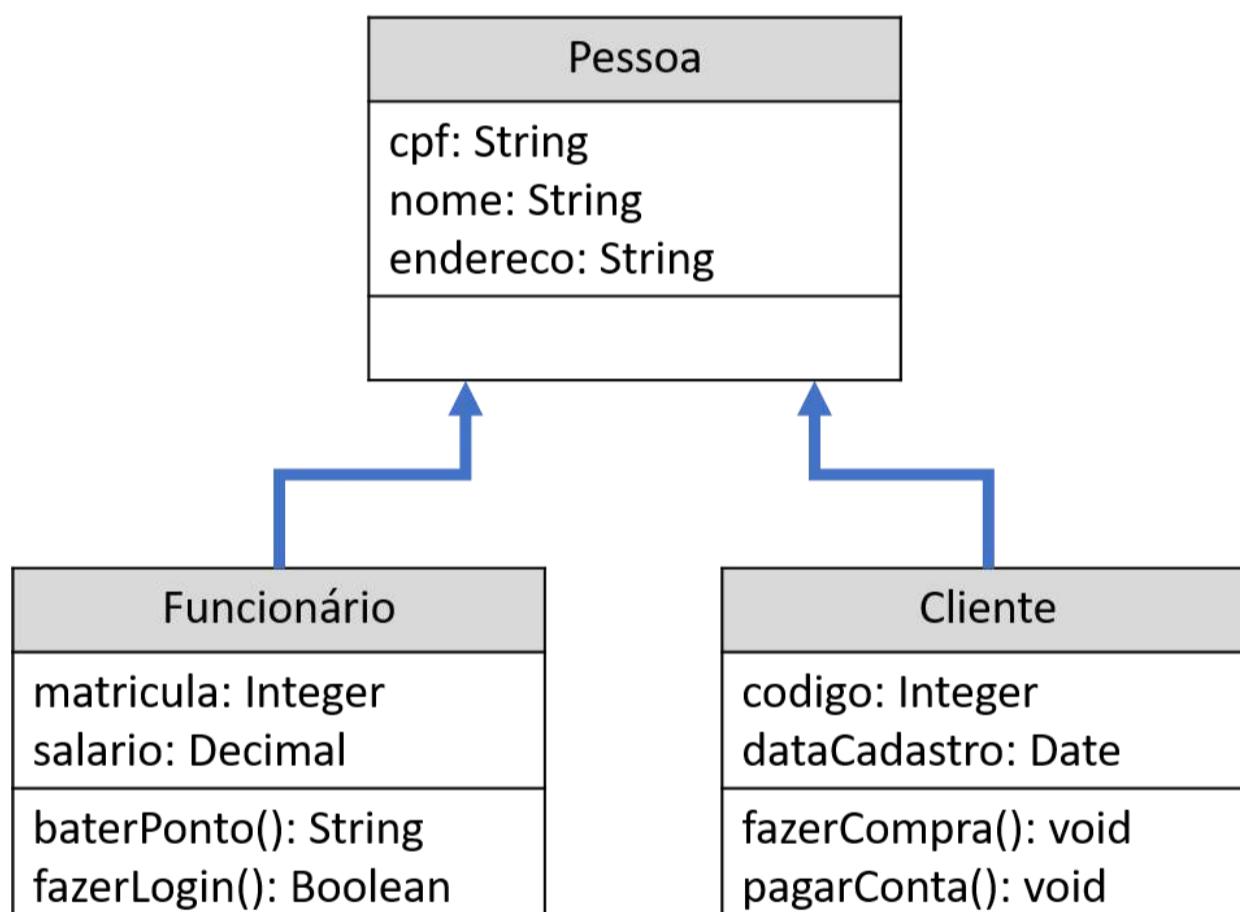
métodos para guardar e acessar seus valores.

## HERANÇA

A Figura 3.2 utiliza um diagrama de classes para ilustrar outro importante conceito da OO, a herança. Por meio desse mecanismo, é possível fazer o reúso de código criando soluções mais organizadas. A herança permite que uma classe herde os atributos e métodos de outra classe. Observe, na Figura 3.2, que as classes *funcionário* e *cliente* herdam os atributos da classe *pessoa*. A classe *pessoa* pode ser chamada de classe-pai, classe-base, superclasse, ancestral; por sua vez, as classes derivadas são as classes-filhas, subclasses.

Ver anotações

Figura 3.2 | Diagrama de classe: herança



Fonte: elaborada pela autora.

## POLIFORMISMO

Para finalizar nossa introdução ao paradigma OO, vamos falar do polimorfismo.

*Polimorfismo* é uma palavra grega que, literalmente, significa *muitas formas*.

Embora o polimorfismo esteja fortemente associado à herança, é frequentemente citado separadamente como uma das vantagens mais poderosas das tecnologias

orientadas a objetos (WEISFELD, 2013). Quando uma mensagem é enviada para um objeto, este deve ter um método definido para responder a essa mensagem. Em uma hierarquia de herança, todas as subclasses herdam as interfaces de sua superclasse. No entanto, como toda subclass é uma entidade separada, cada uma delas pode exigir uma resposta separada para a mesma mensagem.

o

Ver anotações

## A CLASSES EM PYTHON

Uma vez que suporta a implementação do encapsulamento, da herança e do polimorfismo, Python é uma linguagem com suporte ao paradigma orientado a objetos. A Figura 3.3 ilustra a sintaxe básica necessária para criar uma classe em Python. Utiliza-se a palavra reservada *class* para indicar a criação de uma classe, seguida do nome e dois pontos. No bloco indentado devem ser implementados os atributos e métodos da classe.

0

Ver anotações

Figura 3.3 | Sintaxe de classe em Python



Fonte: PSF (2020a, [s.p.]).

Veja a seguir a implementação da nossa primeira classe em Python.

In [1]:

```
class PrimeiraClasse:

    def imprimir_mensagem(self, nome): # Criando um método
        print(f"Olá {nome}, seja bem vindo!")
```

In [2]:

```
objeto1 = PrimeiraClasse() # Instanciando um objeto do tipo PrimeiraClasse
objeto1.imprimir_mensagem('João') # Invocando o método
```

```
Olá João, seja bem vindo!
```

Na entrada 1, criamos nossa primeira classe. Na linha 1 temos a declaração, e na linha 3 criamos um método para imprimir uma mensagem. **Todo método em uma classe deve receber como primeiro parâmetro uma variável que indica a referência à classe; por convenção, adota-se o parâmetro *self*.** Conforme

veremos, o parâmetro *self* será usado para acessar os atributos e métodos dentro da própria classe. Além do parâmetro obrigatório para métodos, estes devem receber um parâmetro que será usado na impressão da mensagem. Assim como acontece nas funções, um parâmetro no método é tratado como uma variável local.

o

Ver anotações

Na entrada 2, criamos uma instância da classe na linha 1 (criamos nosso primeiro objeto!). A variável "objeto1" agora é um objeto do tipo *PrimeiraClasse*, razão pela qual pode acessar seus atributos e métodos. Na linha 2, invocamos o método *imprimir\_mensagem()*, passando como parâmetro o nome *João*. Por que passamos somente um parâmetro se o método escrito na entrada 1 espera dois parâmetros? O parâmetro *self* é a própria instância da classe e é passado de forma implícita pelo objeto. Ou seja, só precisamos passar explicitamente os demais parâmetros de um método.

Para acessarmos os recursos (atributos e métodos) de um objeto, após instanciá-lo, precisamos usar a seguinte sintaxe: *objeto.recurso*. Vale ressaltar que os atributos e métodos estáticos não precisam ser instanciados, mas esse é um assunto para outra hora.

#### EXEMPLIFICANDO

Vamos construir uma classe *Calculadora*, que implementa como métodos as quatro operações básicas matemáticas, além da operação de obter o resto da divisão.

In [3]:

```
class Calculadora:

    def somar(self, n1, n2):
        return n1 + n2

    def subtrair(self, n1, n2):
        return n1 - n2

    def multiplicar(self, n1, n2):
        return n1 * n2

    def dividir(self, n1, n2):
        return n1 / n2

    def dividir_resto(self, n1, n2):
        return n1 % n2
```

0

Ver anotações

In [4]:

```
calc = Calculadora()

print('Soma:', calc.somar(4, 3))
print('Subtração:', calc.subtrair(13, 7))
print('Multiplicação:', calc.multiplicar(2, 4))
print('Divisão:', calc.dividir(16, 5))
print('Resto da divisão:', calc.dividir_resto(7, 3))
```

```
Soma: 7
Subtração: 6
Multiplicação: 8
Divisão: 3.2
Resto da divisão: 1
```

Na entrada 3, em que implementamos os cinco métodos, veja que todos possuem o parâmetro *self*, pois são métodos da instância da classe. Cada um deles ainda recebe dois parâmetros que são duas variáveis locais nos

métodos. Na entrada 4, instanciamos um objeto do tipo *Calculadora* e, nas linhas 3 a 7, acessamos seus métodos.

0

Ver anotações

## CONSTRUTOR DA CLASSE `__INIT__()`

Até o momento criamos classes com métodos, os quais utilizam variáveis locais. E os atributos das classes?

Nesta seção, vamos aprender a criar e utilizar **atributos de instância**, também chamadas de variáveis de instâncias. Esse tipo de atributo é capaz de receber um valor diferente para cada objeto. Um atributo de instância é uma variável precedida com o parâmetro `self`, ou seja, a sintaxe para criar e utilizar é `self.nome_atributo`.

Ver anotações

Ao instanciar um novo objeto, é possível determinar um estado inicial para variáveis de instâncias por meio do método construtor da classe. Em Python, o método construtor é chamado de `__init__()`. Veja o código a seguir.

In [5]:

```
class Televisao:  
    def __init__(self):  
        self.volume = 10  
  
    def aumentar_volume(self):  
        self.volume += 1  
  
    def diminuir_volume(self):  
        self.volume -= 1
```

In [6]:

```
tv = Televisao()  
print("Volume ao ligar a tv = ", tv.volume)  
tv.aumentar_volume()  
print("Volume atual = ", tv.volume)
```

Volume ao ligar a tv = 10

Volume atual = 11

Na entrada 5, criamos a classe *Televisao*, que possui um atributo de instância e três métodos, o primeiro dos quais é (*\_init\_*), aquele que é invocado quando o objeto é instanciado. Nesse método construtor, instanciamos o atributo *volume* com o valor 10, ou seja, todo objeto do tipo *Televisao* será criado com *volume* = 1. Veja que o atributo recebe o prefixo *self.*, que o identifica como variável de instância. Esse tipo de atributo pode ser usado em qualquer método da classe, uma vez que é um atributo do objeto, eliminando a necessidade de passar parâmetros para os métodos. Nos métodos *aumentar\_volume* e *diminuir\_volume*, alteramos esse atributo sem precisar passá-lo como parâmetro, já que é uma variável do objeto, e não uma variável local.

o

Ver anotações

## VARIÁVEIS E MÉTODOS PRIVADOS

Em linguagens de programação OO, como Java e C#, as classes, os atributos e os métodos são acompanhados de modificadores de acesso, que podem ser: *public*, *private* e *protected*. Em Python, não existem modificadores de acesso e todos os recursos são públicos. Para simbolizar que um atributo ou método é privado, por convenção, usa-se um sublinhado “*\_*” antes do nome; por exemplo, *\_cpf*, *\_calcular\_desconto()* (PSF, 2020a).

Conceitualmente, dado que um atributo é privado, ele só pode ser acessado por membros da própria classe. Portanto, ao declarar um atributo privado, precisamos de métodos que acessem e recuperam os valores ali guardados. Em Python, além de métodos para este fim, um atributo privado pode ser acessado por decorators. Embora a explicação sobre estes fuja ao escopo desta seção, recomendamos a leitura do portal Python Course (2020).

Observe o código a seguir, em cuja entrada 7, o atributo *\_saldo* é acessado pelos métodos *depositar()* e *consultar\_saldo()*.

In [7]:

```
class ContaCorrente:  
    def __init__(self):  
        self._saldo = None  
  
    def depositar(self, valor):  
        self._saldo += valor  
  
    def consultar_saldo(self):  
        return self._saldo
```

0

Ver anotações

Na entrada 7, implementamos a classe *ContaCorrente*, que possui dois atributos privados: *\_cpf* e *\_saldo*. Para guardar um valor no atributo *cpf*, deve-se chamar o método *set\_cpf*, e, para recuperar seu valor, usa-se *get\_cpf*. Para guardar um valor no atributo *saldo*, é preciso invocar o método *depositar()*, e, para recuperar seu valor, deve-se usar o método *get\_saldo*. Lembre-se: em Python, atributos e métodos privados são apenas uma convenção, pois, na prática, os recursos podem ser acessados de qualquer forma.

0

Ver anotações

## HERANÇA EM PYTHON

Como vimos, um dos pilares da OO é a reutilização de código por meio da herança, que permite que uma classe-filha herde os recursos da classe-pai. Em Python, uma classe aceita múltiplas heranças, ou seja, herda recursos de diversas classes. A sintaxe para criar a herança é feita com parênteses após o nome da classe: `class NomeClasseFilha(NomeClassePai)`. Se for uma herança múltipla, cada superclasse deve ser separada por vírgula.

Para nosso primeiro exemplo sobre herança em Python, vamos fazer a implementação do diagrama de classes da Figura 3.2. Observe o código a seguir. Na entrada 8, criamos a classe *pessoa* com três atributos que são comuns a todas pessoas da nossa solução. Na entrada 9, criamos a classe *funcionario*, que herda todos os recursos da classe *pessoa*, razão pela qual dizemos que "**um funcionário é uma pessoa**". Na entrada 10, criamos a classe *cliente*, que também herda os recursos da classe *pessoa*, logo, "**um cliente é uma pessoa**".

In [8]:

```
class Pessoa:

    def __init__(self):
        self.cpf = None
        self.nome = None
        self.endereco = None
```

In [9]:

```
class Funcionario(Pessoa):

    def __init__(self):
        self.matricula = None
        self.salario = None

    def bater_ponto(self):
        # código aqui
        pass

    def fazer_login(self):
        # código aqui
        pass
```

0

Ver anotações

In [10]:

```
class Cliente(Pessoa):  
  
    def __init__(self):  
        self.codigo = None  
        self.dataCadastro = None  
  
    def fazer_compra(self):  
        # código aqui  
        pass  
  
    def pagar_conta(self):  
        # código aqui  
        pass
```

0

Ver anotações

In [11]:

```
f1 = Funcionario()  
f1.nome = "Funcionário A"  
print(f1.nome)  
  
c1 = Cliente()  
c1.cpf = "111.111.111-11"  
print(c1.cpf)
```

```
Funcionário A  
111.111.111-11
```

Na entrada 11, podemos ver o resultado da herança. Na linha 1,instanciamos um objeto do tipo *funcionário*, atribuindo à variável *nome* o valor "Funcionário A". O atributo *nome* foi herdado da classe-pai. Veja, no entanto, que o utilizamos normalmente, pois, na verdade, a partir de então esse atributo faz parte da classe mais especializada. O mesmo acontece para o atributo *cpf*, usado no objeto c1 que é do tipo *cliente*.

## MÉTODOS MÁGICOS EM PYTHON

Estamos usando herança desde a criação da nossa primeira classe nesta aula. Não sabíamos disso porque a herança estava sendo feita de modo implícito. Quando uma classe é criada em Python, ela herda, mesmo que não declarado explicitamente, todos os recursos de uma classe-base chamada *object*. Veja o resultado da função *dir()*, que retorna uma lista com os recursos de um objeto. Como é possível observar na saída 12 (Out[12]), a classe *Pessoa*, que explicitamente não tem nenhuma herança, possui uma série de recursos nos quais os nomes estão com *underline* (sublinhado). Todos eles são chamados de métodos mágicos e, com a herança, podem ser sobrescritos, conforme veremos a seguir.

Ver anotações

In [12]:

```
dir(Pessoa())
```

Out[12]:

```
[ '__class__',
  '__delattr__',
  '__dict__',
  '__dir__',
  '__doc__',
  '__eq__',
  '__format__',
  '__ge__',
  '__getattribute__',
  '__gt__',
  '__hash__',
  '__init__',
  '__init_subclass__',
  '__le__',
  '__lt__',
  '__module__',
  '__ne__',
  '__new__',
  '__reduce__',
  '__reduce_ex__',
  '__repr__',
  '__setattr__',
  '__sizeof__',
  '__str__',
  '__subclasshook__',
  '__weakref__',
  'cpf',
  'endereco',
  'nome']
```

0

Ver anotações

## MÉTODO CONSTRUTOR NA HERANÇA E SOBRESCRITA

Na herança, quando adicionamos a função `__init__()`, a classe-filho não herdará o construtor dos pais. Ou seja, o construtor da classe-filho sobrescreve (override) o da classe-pai. Para utilizar o construtor da classe-base, é necessário invocá-lo explicitamente, dentro do construtor-filho, da seguinte forma:

`ClassePai.__init__()`. Para entendermos como funciona a sobreescrita do método construtor e o uso de métodos mágicos, vamos implementar nossa própria versão da classe `'int'`, que é usada para instanciar objetos inteiros. Observe o código a seguir.

In [13]:

```
class int42(int):

    def __init__(self, n):
        int.__init__(n)

    def __add__(a, b):
        return 42

    def __str__(n):
        return '42'
```

In [14]:

```
a = int42(7)
b = int42(13)
print(a + b)
print(a)
print(b)
```

42

42

42

Ver anotações

In [15]:

```
c = int(7)
d = int(13)
print(c + d)
print(c)
print(d)
```

20

7

13

0

Ver anotações

Na entrada 13, temos a classe-filho *int42*, que tem como superclasse a classe *int*. Veja que, na linha 3, definimos o construtor da nossa classe, mas, na linha 4, fazemos com que nosso construtor seja sobreescrito pelo construtor da classe-base, o qual espera receber um valor. O método *construtor* é um método mágico, assim como o *\_add\_* e o *\_str\_*. O primeiro retorna a soma de dois valores, mas, na nossa classe *int42*, ele foi sobreescrito para sempre retornar o mesmo valor. O segundo método, *\_str\_*, retorna uma representação de string do objeto, e, na nossa classe, sobreescrivemos para sempre imprimir 42 como a string de representação do objeto, que será exibida sempre que a função *print()* for invocada para o objeto.

Na entrada 14, podemos conferir o efeito de substituir os métodos mágicos. Nossa classe *int42* recebe como parâmetro um valor, uma vez que estamos usando o construtor da classe-base *int*. O método *\_add\_*, usado na linha 3, ao invés de somar os valores, simplesmente retorna o valor 42, pois o construímos assim. Nas linhas 4 e 5 podemos ver o efeito do método *\_str\_*, que também foi sobreescrito para sempre imprimir 42.

Na entrada 15, usamos a classe original *int* para você poder perceber a diferença no comportamento quando sobreescrivemos os métodos mágicos.

Ao sobreescriver os métodos mágicos, utilizamos outra importante técnica da OO, o **polimorfismo**. Essa técnica, vale dizer, pode ser utilizada em qualquer método, não somente nos mágicos. Construir métodos com diferentes comportamentos pode ser feito sobreescrevendo (**override**) ou sobrecregando (**overload**) métodos. No primeiro caso, a classe-filho sobre escreve um método da classe-base por exemplo, o construtor, ou qualquer outro método. No segundo caso, da sobre carga, um método é escrito com diferentes assinaturas para suportar diferentes comportamentos.

Em Python, graças à sua natureza de interpretação e tipagem dinâmica, a operação de sobre carga não é suportada de forma direta, o que significa que não conseguimos escrever métodos com os mesmos nomes, mas diferentes parâmetros (RAMALHO, 2014). Para contornar essa característica, podemos escrever o método com parâmetros default. Assim, a depender dos que forem passados, mediante estruturas condicionais, o método pode ter diferentes comportamentos ou fazer o overload com a utilização do decorator *functools.singledispatch*, cuja explicação foge ao escopo e propósito desta seção [no entanto, recomendamos, para aprofundamento sobre o assunto, a leitura das páginas 202 a 205 da obra de Ramalho (2014), além da documentação oficial do Python (que você encontra em: <https://docs.python.org/pt-br/3.7/library/functools.html>).

0

Ver anotações

## | HERANÇA MÚLTIPLA

Python permite que uma classe-filha herde recursos de mais de uma superclasse. Para isso, basta declarar cada classe a ser herdada separada por vírgula. A seguir temos uma série de classes. A classe *PCIEthernet* herda recursos das classes *PCI* e *Ethernet*; logo, *PCIEthernet* é uma *PCI* e também uma *Ethernet*. A classe *USBWireless* herda de *USB* e *Wireless*, mas *Wireless* herda de *Ethernet*; logo,

USBWireless é uma USB, uma Wireless e também uma Ethernet, o que pode ser confirmado pelos resultados da função built-in `isinstance(objeto, classe)`, que checa se um objeto é uma instância de uma determinada classe.

0

In [16]:

Ver anotações

```
class Ethernet():

    def __init__(self, name, mac_address):
        self.name = name
        self.mac_address = mac_address

class PCI():

    def __init__(self, bus, vendor):
        self.bus = bus
        self.vendor = vendor

class USB():

    def __init__(self, device):
        self.device = device

class Wireless(Ethernet):

    def __init__(self, name, mac_address):
        Ethernet.__init__(self, name, mac_address)

class PCIEthernet(PCI, Ethernet):

    def __init__(self, bus, vendor, name, mac_address):
        PCI.__init__(self, bus, vendor)
        Ethernet.__init__(self, name, mac_address)

class USBWireless(USB, Wireless):

    def __init__(self, device, name, mac_address):
        USB.__init__(self, device)
        Wireless.__init__(self, name, mac_address)

eth0 = PCIEthernet('pci :0:0:1', 'realtek', 'eth0', '00:11:22:33:44')
```

Ver anotações

```
wlan0 = USBWireless('usb0', 'wlan0', '00:33:44:55:66')

print('PCIEthernet é uma PCI?', isinstance(eth0, PCI))
print('PCIEthernet é uma Ethernet?', isinstance(eth0, Ethernet))
print('PCIEthernet é uma USB?', isinstance(eth0, USB))

print('\nUSBWireless é uma USB?', isinstance(wlan0, USB))
print('USBWireless é uma Wireless?', isinstance(wlan0, Wireless))
print('USBWireless é uma Ethernet?', isinstance(wlan0, Ethernet))
print('USBWireless é uma PCI?', isinstance(wlan0, PCI))
```

PCIEthernet é uma PCI? True

PCIEthernet é uma Ethernet? True

PCIEthernet é uma USB? False

USBWireless é uma USB? True

USBWireless é uma Wireless? True

USBWireless é uma Ethernet? True

USBWireless é uma PCI? False

0

Ver anotações

## EXEMPLIFICANDO

Para treinar tudo o que aprendemos nessa seção, vamos implementar a solução representada no diagrama de classes da Figura 3.4, na qual temos uma classe-base chamada *ContaCorrente* com os seguintes campos:

- Nome, que é do tipo *string* e deve ser público.
- Email, que é do tipo *string* e deve ser público.
- Telefone, que é do tipo *string* e deve ser público.
- Saldo, que é do tipo ponto *flutuante* e deve ser público.

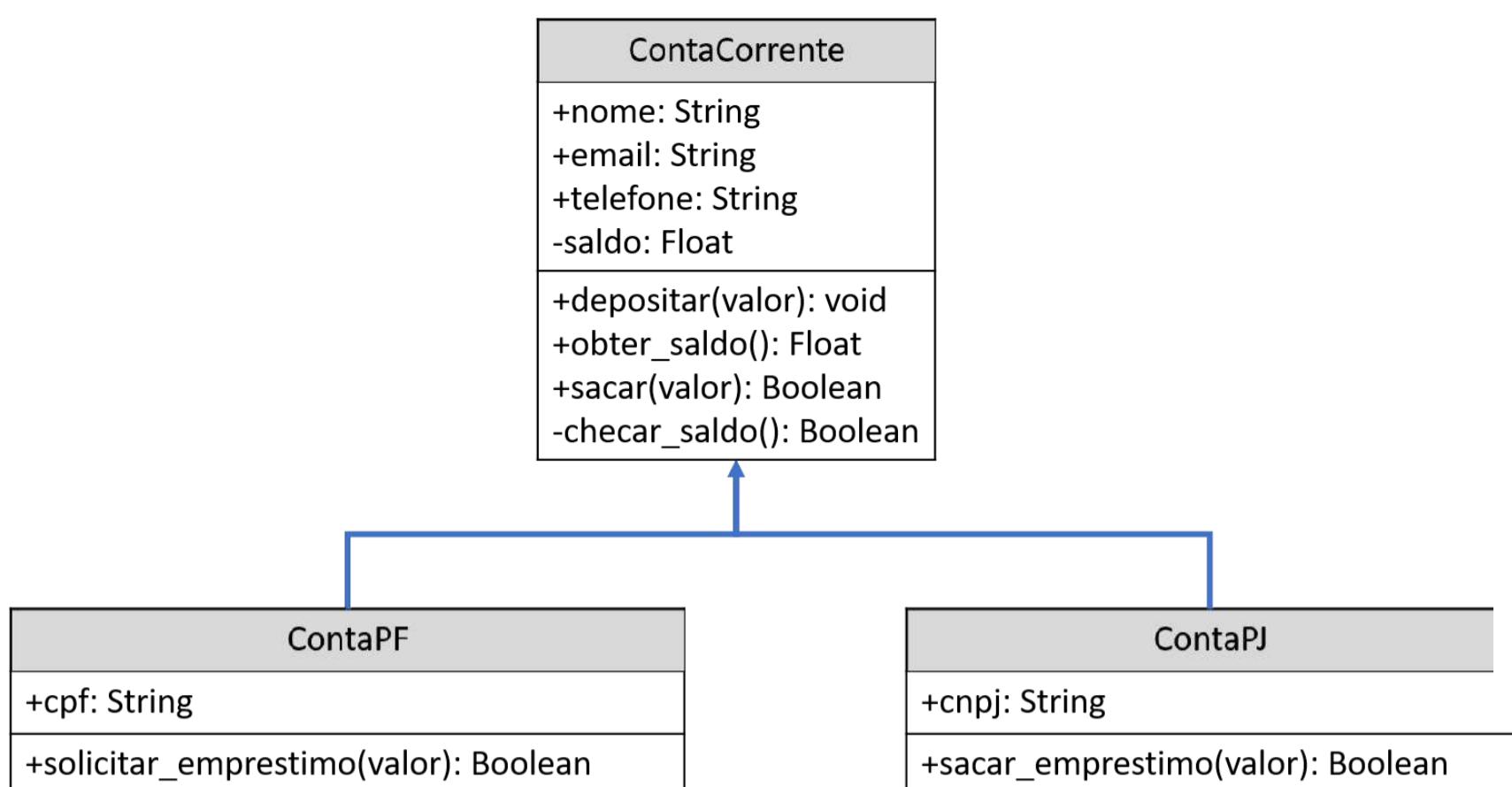
A classe-base conta ainda com os métodos:

- *Depositar*, que recebe como parâmetro um valor, não retorna nada e deve ser público.
- *Obter saldo*, que não recebe nenhum parâmetro, retorna um ponto flutuante e deve ser público.
- *Sacar*, que recebe como parâmetro um valor, retorna se foi possível sacar (booleano) e deve ser público.
- *Checar saldo*, que não recebe nenhum parâmetro, retorna um booleano e deve ser privado, ou seja, será usado internamente pela própria classe.

0

Ver anotações

Figura 3.4 | Diagrama de classe *ContaCorrente*



0

Ver anotações

Fonte: elaborada pela autora.

Na entrada 17, implementamos a classe *ContaCorrente*. Na linha 2, especificamos que a classe deve ser construída recebendo como parâmetro um nome (se o nome não for passado, o objeto não é instanciado). Dentro do construtor, criamos os atributos e os inicializamos. Veja que a variável *\_saldo* recebe um sublinhado (underline) como prefixo para indicar que é uma variável privada e deve ser acessada somente por membros da classe. O valor desse atributo deve ser alterado pelos métodos *depositar* e *sacar*, mas veja que o atributo é usado para *checar o saldo* (linha 9) e também no método de consulta ao saldo (linha 25). O método *\_checar\_saldo()* é privado e, por isso, recebe o prefixo de sublinhado (underline) *"\_"*. Tal método é usado como um dos passos para a realização do saque, pois esta só pode acontecer se houver saldo suficiente. Veja que, na linha 18, antes de fazermos o saque, invocamos o método *self.\_checar\_saldo(valor)* – o *self* indica que o método pertence à classe.

In [17]:

```
class ContaCorrente:

    def __init__(self, nome):
        self.nome = nome
        self.email = None
        self.telefone = None
        self._saldo = 0

    def _checlar_saldo(self, valor):
        return self._saldo >= valor

    def depositar(self, valor):
        self._saldo += valor

    def sacar(self, valor):
        if self._checlar_saldo(valor):
            self._saldo -= valor
            return True
        else:
            return False

    def obter_saldo(self):
        return self._saldo
```

0

Ver anotações

Na entrada 18, criamos a classe *ContaPF*, que herda todas as funcionalidades da classe-pai *ContaCorrente*, inclusive seu construtor (linha 3). Veja, na linha 2, que o objeto deve ser instanciado passando o nome e o cpf como parâmetros – o nome faz parte do construtor-base. Além dos recursos herdados, criamos o atributo *cpfe* o método *solicitar\_emprestimo*, que consulta o saldo para aprovar ou não o empréstimo.

In [18]:

```
class ContaPF(ContaCorrente):  
    def __init__(self, nome, cpf):  
        super().__init__(nome)  
        self.cpf = cpf  
  
    def solicitar_emprestimo(self, valor):  
        return self.obter_saldo() >= 500
```

0

Ver anotações

Na entrada 19, criamos a classe *ContaPJ*, que herda todas as funcionalidades da classe-pai *ContaCorrente*, inclusive seu construtor (linha 3). Veja, na linha 2, que o objeto deve ser instanciado passando o nome e o cnpj como parâmetros – o nome faz parte do construtor-base. Além dos recursos herdados, criamos o atributo *cnpj* e o método *sacar\_emprestimo*, que verifica se o valor solicitado é inferior a 5000. Observe que, para esse valor, não usamos o parâmetro *self*, pois se trata de uma variável local, não um atributo de classe ou instância.

In [19]:

```
class ContaPJ(ContaCorrente):  
    def __init__(self, nome, cnpj):  
        super().__init__(nome)  
        self.cnpj = cnpj  
  
    def sacar_emprestimo(self, valor):  
        return valor <= 5000
```

Na entrada 20, criamos um objeto do tipo *ContaPF* para testar as funcionalidades implementadas. Veja que instanciamos com os valores necessários. Na linha 2, fazemos um depósito, na linha 3, consultamos o saldo e, na linha 4, solicitamos um empréstimo. Como há saldo suficiente, o empréstimo é aprovado. Na segunda parte dos testes, realizamos um saque, pedimos para imprimir o restante e solicitamos novamente um empréstimo, o qual, neste momento, é negado, pois não há saldo suficiente.

In [20]:

```
conta_pf1 = ContaPF("João", '111.111.111-11')
conta_pf1.depositar(1000)
print('Saldo atual é', conta_pf1.obter_saldo())
print('Receberá empréstimo = ', conta_pf1.solicitar_emprestimo(2000))

conta_pf1.sacar(800)
print('Saldo atual é', conta_pf1.obter_saldo())
print('Receberá empréstimo = ', conta_pf1.solicitar_emprestimo(2000))
```

```
Saldo atual é 1000
Receberá empréstimo = True
Saldo atual é 200
Receberá empréstimo = False
```

Ver anotações

Na entrada 21, criamos um objeto do tipo *ContaPJ* para testar as funcionalidades implementadas. Veja que instanciamos com os valores necessários. Na linha 2, consultamos o saldo e na linha 3 solicitamos o saque de um empréstimo. Mesmo não havendo saldo para o cliente, uma vez que a regra do empréstimo não depende desse atributo, o saque é permitido.

In [21]:

```
conta_pj1 = ContaPJ("Empresa A", "11.111.111/1111-11")
print('Saldo atual é', conta_pj1.obter_saldo())
print('Receberá empréstimo = ', conta_pj1.sacar_emprestimo(3000))
```

```
Saldo atual é 0
Receberá empréstimo = True
```

Que tal testar todas essas implementações e funcionalidades na ferramenta Python Tutor? Aproveite a ferramenta e explore novas possibilidades.

The screenshot shows a Python Tutor interface. On the left, a code editor displays Python 3.6 code for a `ContaCorrente` class. The code includes methods for initialization, checking balance, depositing, and withdrawing. A red arrow points to the next line of code to be executed. On the right, a large rectangular area labeled "Print output (drag lower right corner to resize)" is empty. Below it are tabs for "Frames" and "Objects". At the bottom of the code editor, there's an "Edit this code" link, a legend for execution markers, and navigation buttons for "Step 1 of 72". A vertical scrollbar on the right indicates the code is scrollable.

```

Python 3.6
Print output (drag lower right corner to resize)

→ 1 class ContaCorrente:
2     def __init__(self, nome):
3         self.nome = nome
4         self.email = None
5         self.telefone = None
6         self._saldo = 0
7
8     def _checkar_saldo(self, valor):
9         if self._saldo >= valor:
10            return True
11        else:
12            return False
13
14    def depositar(self, valor):
15        self._saldo += valor
16
17    def sacar(self, valor):

```

[Edit this code](#)

→ line that just executed  
→ next line to execute

< Prev Next >  
Step 1 of 72

Visualized with [pythontutor.com](http://pythontutor.com)

## REFERÊNCIAS E LINKS ÚTEIS

LJUBOMIR, P. **Introdução à computação usando Python**: um foco no desenvolvimento de aplicações. Rio de Janeiro: LTC, 2016.

PSF. Python Software Foundation. **Classes**. 2020a. Disponível em: <https://bit.ly/3fSURVx>. Acesso em: 10 maio 2020.

PYTHON COURSE. Properties vs. Getters and Setters. Python 3 Tutorial. 2020. Disponível em: <https://bit.ly/2XWY9Rn>. Acesso em: 14 jun. 2020

RAMALHO, L. **Fluent Python**. Gravenstein: O'Reilly Media, 2014.

WEISFELD, M. A. **The Object-Oriented**: Thought Process. 4. ed. [S.I.]: Addison-Wesley Professional, 2013.

# FOCO NO MERCADO DE TRABALHO

## CLASSES E MÉTODOS EM PYTHON

Vanessa Cadan Scheffer

0

Ver anotações

### DIAGRAMA DE CLASSES E HERANÇA

Um diagrama de classes permite ilustrar o reúso de código através do mecanismo de herança onde uma classe herda os atributos e métodos de outra classe.



Fonte: Shutterstock.

### Deseja ouvir este material?

Áudio disponível no material digital.

### DESAFIO

Como desenvolvedor em uma empresa de consultoria de software, você foi alocado para iniciar a implementação de um sistema de vendas. Nesta primeira etapa, os engenheiros de software fizeram um levantamento de requisitos e

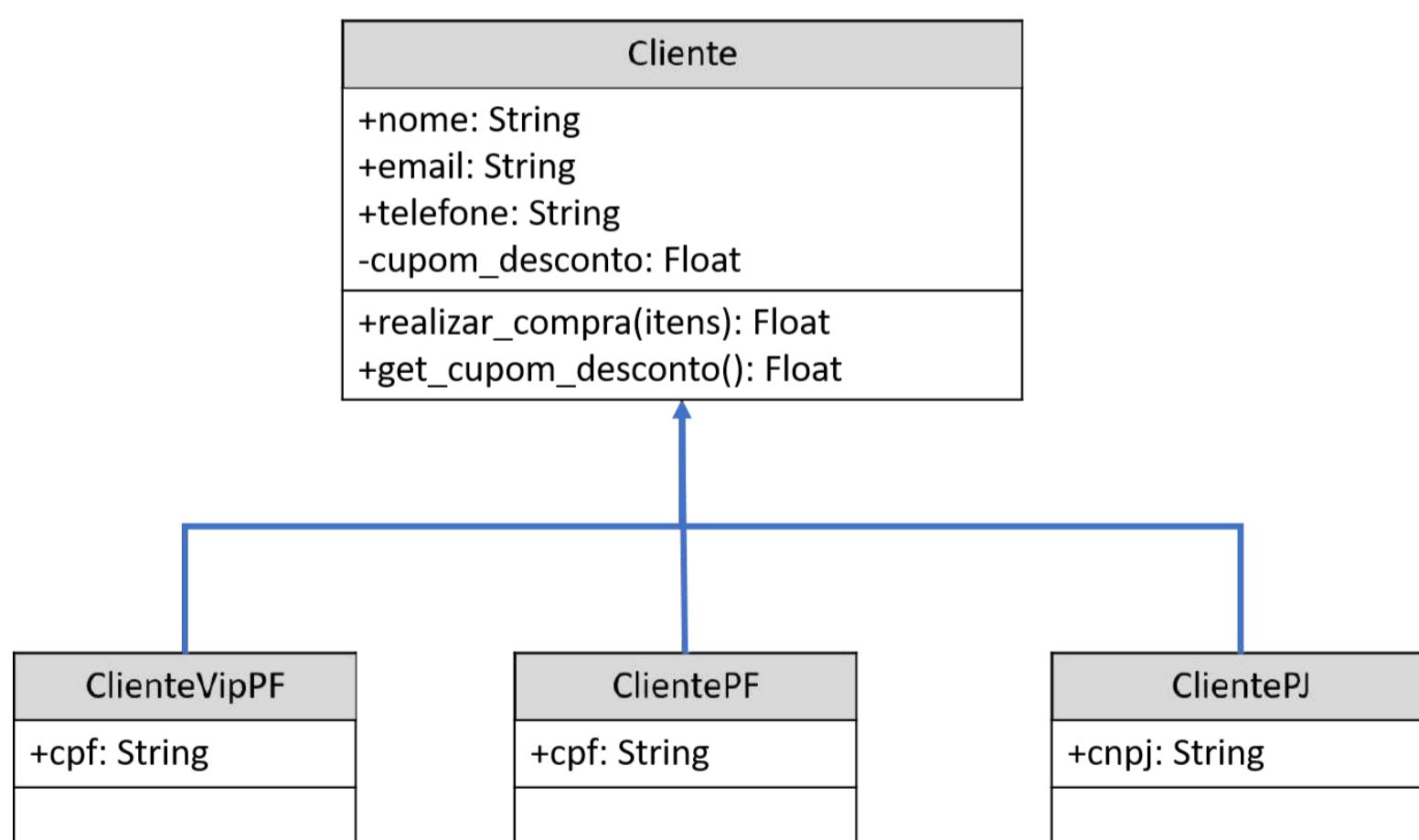
identificaram que o sistema vai atender a três tipos de distintos de clientes: pessoas físicas (PF) que compram com assiduidade, pessoas físicas que compram esporadicamente e pessoas jurídicas (PJ). Os clientes VIPs podem comprar um número ilimitado de itens. Por sua vez, os clientes PF que não são VIPs, só podem comprar até 20 itens, e os clientes PJ podem comprar até 50 itens.

No levantamento de requisitos, também foi determinado que cada tipo de cliente terá um cupom de desconto que será usado para conceder benefícios nas compras. Os clientes VIPs terão um desconto de 20% a cada compra. Os clientes esporádicos (PF) terão um desconto de 5%. Por sua vez, os clientes com CNPJ terão um desconto de 10%. O cupom de desconto deverá ser encapsulado em um método.

Você recebeu a missão de implementar esse primeiro esboço do sistema, tarefa para cuja execução os engenheiros de software lhe entregaram o diagrama de classes representado na Figura 3.5. Agora é com você! Implemente a solução e apresente-a a equipe!

Ver anotações

Figura 3.5 | Esboço inicial do diagrama de classes



Fonte: elaborada pela autora.

## RESOLUÇÃO

Para este desafio, será necessário implementar quatro classes: uma base e três subclasses, que herdam os recursos. Todos os atributos que são comuns às classes devem ficar na classe-base. Por sua vez, os atributos específicos devem ser implementados nas classes-filhas.

A quantidade de itens que cada cliente pode comprar é diferente, razão pela qual o método *realizar\_compra* precisa ser sobreescrito em cada subclasse para atender essa especificidade. O cupom de desconto também varia de acordo com o tipo de cliente. Observe, a seguir, uma possível implementação para a solução.

0  
Ver anotações

In [22]:

```
class Cliente:  
  
    def __init__(self):  
        self.nome = None  
        self.emil = None  
        self.telefone = None  
        self._cupom_desconto = 0  
  
    def get_cupom_desconto(self):  
        return self._cupom_desconto  
  
    def realizar_compras(self, lista_itens):  
        pass
```

In [23]:

```
class ClienteVipPF(Cliente):  
  
    def __init__(self):  
        super().__init__()  
        self._cupom_desconto = 0.2  
  
    def realizar_compras(self, lista_itens):  
        return f"Quantidade total de itens comprados = {len(lista_itens)}"
```

In [24]:

```
class ClientePF(Cliente):  
    def __init__(self):  
        super().__init__()  
        self._cupom_desconto = 0.05  
  
    def realizar_compras(self, lista_itens):  
        if len(lista_itens) <= 20:  
            return f"Quantidade total de itens comprados = {len(lista_itens)}"  
        else:  
            return "Quantidade de itens superior ao limite permitido."
```

In [25]:

```
class ClientePJ(Cliente):  
    def __init__(self):  
        super().__init__()  
        self._cupom_desconto = 0.1  
  
    def realizar_compras(self, lista_itens):  
        if len(lista_itens) <= 50:  
            return f"Quantidade total de itens comprados = {len(lista_itens)}"  
        else:  
            return "Quantidade de itens superior ao limite permitido."
```

In [26]:

```
cli1 = ClienteVipPF()  
cli1.nome = "Maria"  
print(cli1.get_cupom_desconto())  
cli1.realizar_compras(['item1', 'item2', 'item3'])
```

0.2

Out[26]:

```
'Quantidade total de itens comprados = 3'
```

0

Ver anotações

Implementamos as quatro classes necessárias. Veja que em cada classe-filha, determinados o valor do cupom de desconto logo após invocar o construtor da classe-base. Em cada subclasse também sobrescrevemos o método *realizar\_compras* para atender aos requisitos específicos.

Conforme você pode notar, a orientação a objetos permite entregar uma solução mais organizada, com reaproveitamento de código, o que certamente pode facilitar a manutenção e a inclusão de novas funcionalidades. Além disso, utilizar o diagrama de classes para se comunicar com outros membros da equipe tem grande valor no projeto de desenvolvimento de um software.

Utilize o emulador a seguir para testar a implementação e fazer novos testes!

0

Ver anotações

Python 3.6

```
→ 1 class Cliente:
  2     def __init__(self):
  3         self.nome = None
  4         self.emil = None
  5         self.telefone = None
  6         self._cupom_desconto = 0
  7
  8     def get_cupom_desconto(self):
  9         return self._cupom_desconto
 10
 11    def realizar_compras(self, lista):
 12        pass
 13
 14
 15 class ClienteVipPF(Cliente):
 16     def __init__(self):
 17         super().__init__()
```

Print output (drag lower right corner to resize)

Frames Objects

Edit this code

## DESAFIO DA INTERNET

Ganhar habilidade em programação exige estudo e treino (muito treino).

Acesse a biblioteca virtual no endereço: <http://biblioteca-virtual.com/> e busque pelo seguinte livro:

**LJUBOMIR, P. Introdução à computação usando Python:** um foco no desenvolvimento de aplicações. Rio de Janeiro: LTC, 2016.

Na página 301 do Capítulo 8 (*Programação orientada a objetos*) da referida obra,

você é convidado, no exercício 8.20, a criar uma classe chamada *minhaLista*. Essa classe deve se comportar como a classe list, exceto pelo método *sort*. Que tal tentar resolver esse desafio?!

0

Ver anotações

NÃO PODE FALTAR

## BIBLIOTECAS E MÓDULOS EM PYTHON

Vanessa Cadan Scheffer

0

Ver anotações

### O QUE SÃO MÓDULOS?

Um módulo pode ser uma biblioteca de códigos, possui diversas funções (matemáticas, sister operacional. etc.), as quais possibilitam a reutilização de código de uma forma elegante e eficiente.



Fonte: Shutterstock.

**Deseja ouvir este material?**

Áudio disponível no material digital.

# INTRODUÇÃO

Implementamos algoritmos, nas diversas linguagens de programação, para automatizar soluções e acrescentar recursos digitais, como interfaces gráficas e processamento em larga escala. Uma solução pode começar com algumas linhas de códigos, mas, em pouco tempo, passar a ter centenas, milhares e até milhões delas. Nesse cenário, trabalhar com um único fluxo de código se torna inviável, razão pela qual surge a necessidade de técnicas de implementação para organizar a solução.

Ver anotações

## MÓDULOS E BIBLIOTECAS EM PYTHON

Uma opção para organizar o código é implementar **funções**, contexto em que cada bloco passa a ser responsável por uma determinada funcionalidade. Outra forma é utilizar a orientação a objetos e criar classes que encapsulam as características e os comportamentos de um determinado objeto. Conseguimos utilizar ambas as técnicas para melhorar o código, mas, ainda assim, estamos falando de toda a solução agrupada em um arquivo Python (.py). Considerando a necessidade de implementar uma solução, o mundo ideal é: fazer a separação em funções ou classes e ainda realizar a separação em vários arquivos .py, o que é chamado de *modular uma solução* (PSF, 2020b). Segundo a documentação oficial do Python, é preferível implementar, em um arquivo separado, uma funcionalidade que você possa reutilizar, criando assim um módulo.

“

Um módulo é um arquivo contendo definições e instruções Python. O nome do arquivo é o nome do módulo acrescido do sufixo .py.  
— (PSF, 2020b, [s.p.])

A Figura 3.6 ilustra essa ideia, com base na qual uma solução original implementada em um único arquivo .py é transformada em três módulos que, inclusive, podem ser reaproveitados, conforme aprenderemos.

Figura 3.6 | Contínuo versus módulo

```

1 def fib(n): # escreve a sequência de Fibonacci até n
2     a, b = 0, 1
3     while a < n:
4         print(a, end=' ')
5         a, b = b, a+b
6     print()
7
8 def fib2(n): # retorno da sequência de Fibonacci até n
9     result = []
10    a, b = 0, 1
11    while a < n:
12        result.append(a)
13        a, b = b, a+b
14    return result
15
16 def hello():
17     print('hello world')
18 # Módulos separados
19 import fib
20 import fib2
21 import hello

```

0

Ver anotações

Fonte: elaborada pela autora.

Falamos em módulo, mas em Python se ouve muito o termo *biblioteca*. O que será que eles têm em comum?

Na verdade, **um módulo pode ser uma biblioteca de códigos!** Observe a Figura 3.7, na qual temos o módulo *math*, que possui diversas funções matemáticas, e o módulo *os*, que possui funções de sistema operacional, como capturar o caminho (*getcwd*), listar um diretório (*listdir*), criar uma nova pasta (*mkdir*), dentre inúmeras outras. Esses módulos são bibliotecas de funções pertinentes a um determinado assunto (matemática e sistema operacional), as quais possibilitam a reutilização de código de uma forma elegante e eficiente.

Figura 3.7a | Módulo como biblioteca

```

1 from math import log # importando a função log do módulo math
2 log(100, 10) # retorna o resultado do log de 100 na base 10

```

Fonte: elaborada pela autora.

Figura 3.7b | Módulo como biblioteca

```
1 # Importando o módulo os (sistema operacional).
2 import os
3
4 #Retorna uma string representando o diretório de trabalho atual.
5 cwd = os.getcwd()
```

Fonte: elaborada pela autora.

0

Ver anotações

## COMO UTILIZAR UM MÓDULO

Para utilizar um módulo é preciso importá-lo para o arquivo. Essa importação pode ser feita de maneiras distintas:

1. `import moduloXXText`
- 1.2 `import moduloXX as apelido`
2. `from moduloXX import itemA, itemB`

Utilizando as duas primeiras formas de importação (1 e 1.2), **todas** as funcionalidades de um módulo são carregadas na memória. A diferença entre elas é que, na primeira, usamos o nome do módulo e, na segunda, atribuímos a este um apelido (*as = alias*). Na outra forma de importação (2), somente funcionalidades **específicas** de um módulo são carregadas na memória.

A forma de importação também determina a sintaxe para utilizar a funcionalidade. Observe os códigos a seguir.

In [1]:

```
import math

math.sqrt(25)
math.log2(1024)
math.cos(45)
```

Out[1]:

```
0.5253219888177297
```

In [2]:

```
import math as m

m.sqrt(25)
m.log2(1024)
m.cos(45)
```

0

Out[2]:

```
0.5253219888177297
```

In [3]:

```
from math import sqrt, log2, cos

sqrt(25)
log2(1024)
cos(45)
```

Out[3]:

```
0.5253219888177297
```

Ver anotações

Na entrada 1, usamos a importação que carrega todas as funções na memória.

Observe (linhas 3 a 5) que precisamos usar a seguinte sintaxe: `nomemodulo.nomeitem`

Na entrada 2, usamos a importação que carrega todas as funções na memória, mas, no caso, demos um apelido para o módulo. Veja (linhas 3 a 5) que, para usá-la, precisamos colocar o apelido do módulo: `apelido.nomeitem`

Na entrada 3, usamos a importação que carrega funções específicas na memória.

Veja (linhas 3 a 5) que, para usá-la, basta invocar a função.

#### BOA PRÁTICA

Todos os *import* devem ficar no começo do arquivo (<https://bit.ly/2XWFFjR>).

Ainda segundo a documentação do site, é uma boa prática declarar primeiro as bibliotecas-padrão (módulos *built-in*), depois as bibliotecas de

terceiros e, por fim, os módulos específicos criados para a aplicação. Cada bloco deve ser separado por uma linha em branco.

0

Ver anotações

## CLASSIFICAÇÃO DOS MÓDULOS (BIBLIOTECAS)

Podemos classificar os módulos (bibliotecas) em três categorias, cada uma das quais vamos estudar:

1. **Módulos built-in**: embutidos no interpretador.
2. **Módulos de terceiros**: criados por terceiros e disponibilizados via PyPI.
3. **Módulos próprios**: criados pelo desenvolvedor.

## MÓDULOS **BUILT-IN**

Ao instalar o interpretador Python, também é feita a instalação de uma biblioteca de módulos, que pode variar de um sistema operacional para outro.

“

Alguns módulos estão embutidos no interpretador; estes possibilitam acesso a operações que não são parte do núcleo da linguagem, mas estão no interpretador seja por eficiência ou para permitir o acesso a chamadas do sistema operacional.

— (PSF, 2020b, [s.p.])

Como estão embutidos no interpretador, esses módulos não precisam de nenhuma instalação adicional.

São vários os módulos *built-in* disponíveis. No endereço <https://bit.ly/2FgUxmJ> você encontra a lista com todos os recursos disponíveis. Vamos explorar alguns deles.

## MÓDULO **RANDOM**

*Random* é um módulo *built-in* usado para criar número aleatórios. Vamos explorar as funções:

- *random.randint(a, b)*: retorna um valor inteiro aleatório, de modo que esse número esteja entre a, b.

- *random.choice(seq)*: extrai um valor de forma aleatória de uma certa sequência.
- *random.sample(population, k)*: retorna uma lista com  $k$  elementos, extraídos da população.

In [4]:

```
import random

print(random.randint(0, 100))
print(random.choice([1, 10, -1, 100]))
print(random.sample(range(100000), k=12))
```

```
80
100
[18699, 46029, 49868, 59986, 14361, 27678, 69635, 39589, 74599, 6587, 61176,
14191]
```

Ver anotações

## MÓDULO *os*

*OS* é um módulo *built-in* usado para executar comandos no sistema operacional.

Vamos explorar as funções:

- *os.getcwd()*: retorna uma string com o caminho do diretório de trabalho.
- *os.listdir(path='.'*): retorna uma lista com todas as entradas de um diretório. Se não for especificado um caminho, então a busca é realizada em outro diretório de trabalho.
- *os.cpu\_count()*: retorna um inteiro com o número de CPUs do sistema.
- *os.getlogin()*: retorna o nome do usuário logado.
- *os.getenv(key)*: retorna uma string com o conteúdo de uma variável de ambiente especificada na key.
- *os.getpid()*: retorna o id do processo atual.

In [5]:

```
import os

os.getcwd()
os.listdir()
os.cpu_count()
os.getlogin()
os.getenv(key='path')
os.getpid()
```

0

Ver anotações

Out[5]:

```
6476
```

## MÓDULO **RE**

O módulo *re* (*regular expression*) fornece funções para busca de padrões em um texto. Uma expressão regular especifica um conjunto de strings que corresponde a ela. As funções neste módulo permitem verificar se uma determinada string corresponde a uma determinada expressão regular. Essa técnica de programação é utilizada em diversas linguagens de programação, pois a construção de *re* depende do conhecimento de padrões. Vamos explorar as funções:

- *re.search(pattern, string, flags=0)*: varre a string procurando o primeiro local onde o padrão de expressão regular produz uma correspondência e o retorna. Retorna *None* se nenhuma correspondência é achada.
- *re.match(pattern, string, flags=0)*: procura por um padrão no começo da string. Retorna *None* se a sequência não corresponder ao padrão.
- *re.split(pattern, string, maxsplit=0, flags=0)*: divide uma string pelas ocorrências do padrão.

Para entendermos o funcionamento da expressão regular, vamos considerar um cenário onde temos um nome de arquivo com a data: meuArquivo\_20-01-2020.py. Nosso objetivo é guardar a parte textual do nome em uma variável para a usarmos posteriormente. Vamos utilizar os três métodos para fazer essa separação. O *search()* faz a procura em toda string, o *match()* faz a procura somente no começo

(razão pela qual, portanto, também encontrará neste caso) e o *split()* faz a transformação em uma lista. Como queremos somente a parte textual, pegamos a posição 0 da lista.

In [6]:

```
import re

string = 'meuArquivo_20-01-2020.py'
padrao = "[a-zA-Z]*"

texto1 = re.search(padrao, string).group()
texto2 = re.match(padrao, string).group()
texto3 = re.split("_", string)[0]

print(texto1)
print(texto2)
print(texto3)
```

```
meuArquivo
meuArquivo
meuArquivo
```

0

Ver anotações

Na linha 4, da entrada 6, construímos uma expressão regular para buscar por sequências de letras maiúsculas e minúsculas [a-zA-Z], que pode variar de tamanho 0 até N (\*). Nas linhas 6 e 7 usamos esse padrão para fazer a procura na string. Ambas as funções conseguiram encontrar; e, então, usamos a função *group()* da *re* para capturar o resultado. Na linha 8, usamos o padrão "\_" como a marcação de onde cortar a string, o que resulta em uma lista com dois valores – como o texto é a primeira parte, capturamos essa posição com o [0].

0

Ver anotações

## ■ MÓDULO **DATETIME**

Trabalhar com datas é um desafio nas mais diversas linguagens de programação. Em Python há um módulo *built-in* capaz de lidar com datas e horas. O módulo *datetime* fornece classes para manipular datas e horas. Uma vez que esse módulo possui classes, então a sintaxe para acessar os métodos deve ser algo similar a: `modulo.classe.metodo()`. Dada a diversa quantidade de possibilidades de se trabalhar com esse módulo, vamos ver um pouco das classes *datetime* e *timedelta*.

In [7]:

```
import datetime as dt

# Operações com data e hora
hoje = dt.datetime.today()
ontem = hoje - dt.timedelta(days=1)
uma_semana_atras = hoje - dt.timedelta(weeks=1)

agora = dt.datetime.now()
duas_horas_atras = agora - dt.timedelta(hours=2)

# Formatação
hoje_formatado = dt.datetime.strftime(hoje, "%d-%m-%Y")
#https://docs.python.org/3/library/datetime.html#strftime-strptime-behavior
ontem_formatado = dt.datetime.strftime(ontem, "%d de %B de %Y")

# Conversão de string para data
data_string = '11/06/2019 15:30'
data_dt = dt.datetime.strptime(data_string, "%d/%m/%Y %H:%M")
```

0

Ver anotações

Na entrada 7, usamos algumas funcionalidades disponíveis no módulo *datetime*. Repare que fizemos a importação com a utilização do apelido de *dt*, prática essa que é comum para nomes grandes.

Linha 4: usamos o método *today()* da classe *datetime* para capturar a data e a hora do sistema.

Linha 5: usamos a classe *timedelta* para subtrair 1 dia de uma data específica.

Linha 6: usamos a classe *timedelta* para subtrair 1 semana de uma data específica.

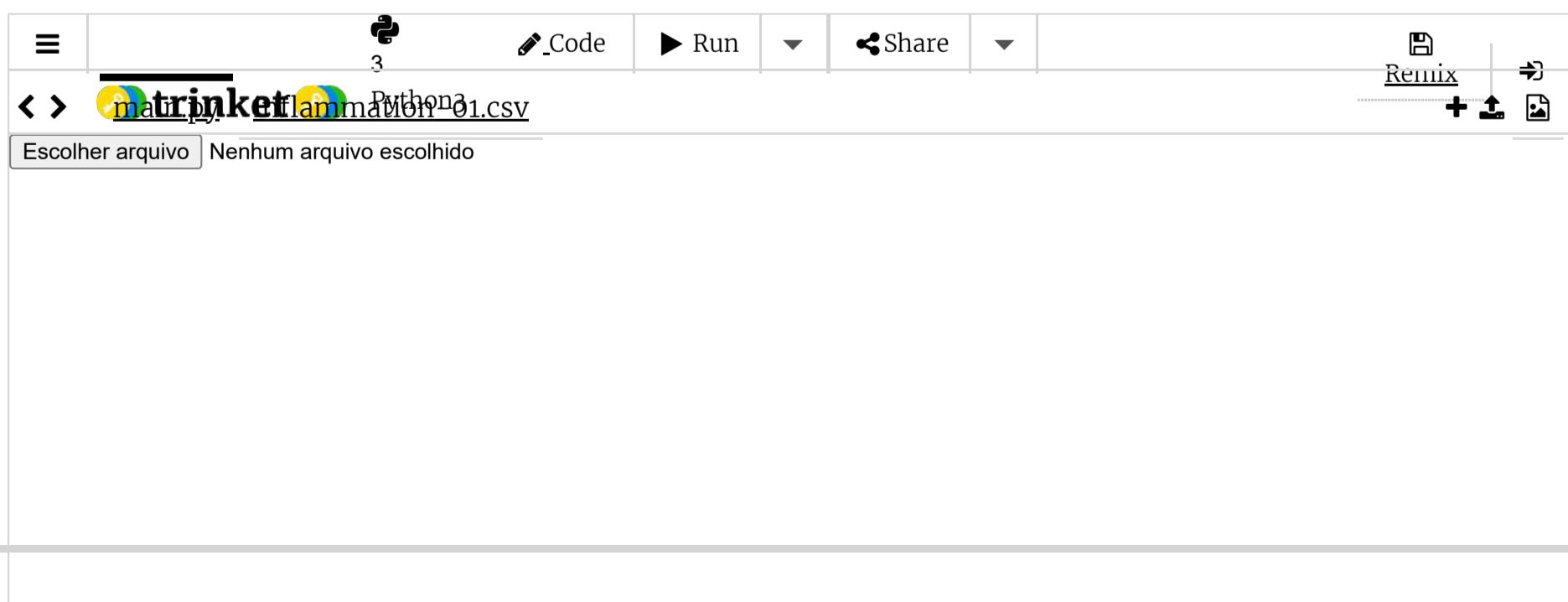
Linha 8: usamos o método *now()* da classe *datetime* para capturar a data e hora do sistema.

Linha 9: usamos a classe *timedelta* para subtrair 2 horas de uma data específica.

Linhas 12 e 13: usamos o método *strftime()* da classe *datetime* para formatar a aparência de uma data específica. [Acesse o endereço <https://bit.ly/2E33mzR> para verificar as possibilidades de formatação.]

Linha 17: usamos o método *strptime()* da classe *datetime*, para converter uma string em um objeto do tipo *datetime*. Essa transformação é interessante, pois habilita as operações que vimos.

Aproveite o emulador a seguir e teste os vários módulos que utilizamos até o momento. [Também é interessante acessar a documentação e explorar novas possibilidades: <https://bit.ly/3ajlgcJ>.]



0

Ver anotações

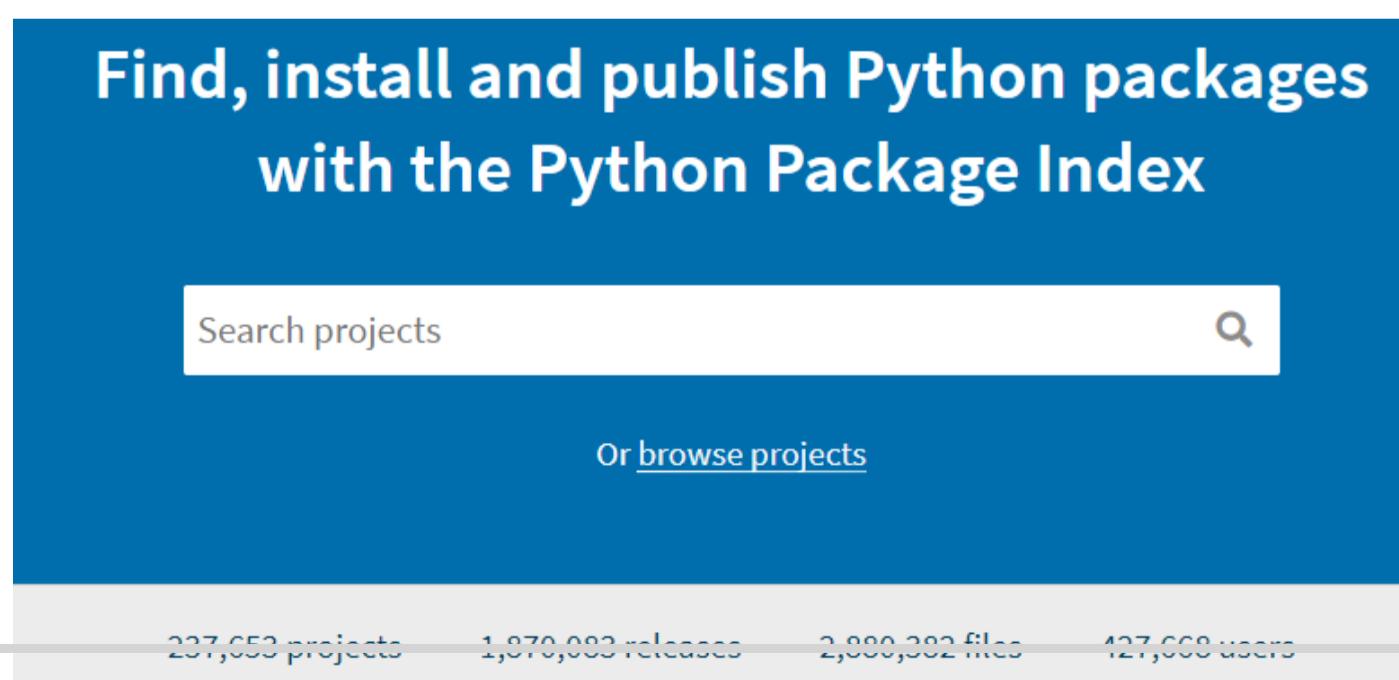
## MÓDULOS DE TERCEIROS

Na documentação oficial da linguagem Python (<https://www.python.org/>), você encontra, em um dos menus, a opção PyPI, que te levará para a página <https://pypi.org/>. PyPI é a abreviação para *Python Package Index*, que é um repositório para programas Python. Programadores autônomos e empresas podem, com isso, criar uma solução em Python e disponibilizar em forma de biblioteca no repositório PyPI, o que permite que todos usufruam e contribuam para o crescimento da linguagem. No próprio portal existe uma documentação explicando como distribuir sua solução com PyPI em <https://bit.ly/3iNxbnt>.

No momento em que este material está sendo produzido, o repositório PyPI conta com 237.653 projetos, conforme mostra a Figura 3.8. Estamos falando de quase 300 mil bibliotecas prontas para usar. São tantas opções, que, se estudássemos uma biblioteca por dia, demoraríamos cerca de  $(237653 // 365) 651$  anos para ver todas! Com tamanha diversidade, o caminho é, diante da necessidade de resolver um problema, buscar em fóruns e comunidades de programadores informações sobre bibliotecas que podem ser usadas para resolver o problema.

Para utilizar uma biblioteca do repositório PyPI, é preciso instalá-la. Para isso, abra um terminal no sistema operacional e digite: *pip install biblioteca* [biblioteca é o nome do pacote que deseja instalar. Por exemplo: *pip install numpy*.]

Figura 3.8 | Repositório PyPI



Fonte: Pypi. Disponível em: <https://pypi.org/>.

Como já deu para perceber, não é possível que consigamos estudar todas as bibliotecas. No entanto, vamos conhecer algumas delas. No dia a dia, existem bibliotecas que têm sido amplamente utilizadas, como as para tratamento e visualização de dados, para implementações de inteligência artificial (*deep learning* e *machine learning*), para tratamento de imagens, para conexão com banco de dados, dentre outras. Veja algumas a seguir:

Ver anotações

## 1. Bibliotecas para tratamento de imagens

- **Pillow:** esta biblioteca oferece amplo suporte aos formatos de arquivo, uma representação interna eficiente e recursos de processamento de imagem bastante poderosos.
- **OpenCV Python:** é uma biblioteca de código aberto licenciada por BSD que inclui várias centenas de algoritmos de visão computacional.
- **Luminoth:** é um *kit* de ferramentas de código aberto para visão computacional. Atualmente, atua com a detecção de objetos, mas a ideia é expandi-la.
- **Mahotas:** é uma biblioteca de algoritmos rápidos de visão computacional (todos implementados em C++ para ganhar velocidade) que opera com matrizes *NumPy*.

## 2. Bibliotecas para visualização de dados

- **Matplotlib:** é uma biblioteca abrangente para criar visualizações estáticas, animadas e interativas em Python.
- **Bokeh:** é uma biblioteca de visualização interativa para navegadores modernos. Oferece interatividade de alto desempenho em conjuntos de dados grandes ou de streaming.
- **Seaborn:** é uma biblioteca para criar gráficos estatísticos em Python.
- **Altair:** é uma biblioteca declarativa de visualização estatística para Python.

## 3. Bibliotecas para tratamento de dados

- **Pandas:** é um pacote Python que fornece estruturas de dados rápidas, flexíveis e expressivas, projetadas para facilitar o trabalho com dados estruturados (em forma de tabela).
- **NumPy:** além de seus óbvios usos científicos, a NumPy também pode ser usada como um eficiente recipiente multidimensional de dados genéricos
- **Pyspark:** Spark é um sistema de computação em cluster rápido e geral para Big Data.
- **Pingouin:** é um pacote estatístico Python baseado em Pandas.

o

Ver anotações

#### 4. Bibliotecas para tratamento de textos

- **Punctuation:** esta é uma biblioteca Python que removerá toda a pontuação em uma string.
- **NLTK:** o *Natural Language Toolkit* é um pacote Python para processamento de linguagem natural.
- **FlashText:** este módulo pode ser usado para substituir palavras-chave em frases ou extraí-las.
- **TextBlob:** é uma biblioteca Python para processamento de dados textuais.

#### 5. Internet, rede e cloud

- **Requests:** permite que você envie solicitações HTTP/1.1 com extrema facilidade. Não há necessidade de adicionar manualmente *queries* de consulta aos seus URLs ou de codificar os dados PUT e POST: basta usar o método JSON.
- **BeautifulSoup:** é uma biblioteca que facilita a captura de informações de páginas da web.
- **Paramiko:** é uma biblioteca para fazer conexões SSH2 (cliente ou servidor). A ênfase está no uso do SSH2 como uma alternativa ao SSL para fazer conexões seguras entre scripts Python.
- **s3fs:** é uma interface de arquivos Python para S3 (Amazon Simple Storage Service).

#### 6 Bibliotecas para acesso a bancos de dados

- **mysql-connector-python:** permite que programas em Python acessem bancos de dados MySQL.
- **cx-Oracle:** permite que programas em Python acessem bancos de dados Oracle.
- **psycopg2:** permite que programas em Python acessem bancos de dados PostgreSQL.
- **SQLAlchemy:** fornece um conjunto completo de padrões de persistência, projetados para acesso eficiente e de alto desempenho a diversos banco de dados, adaptado para uma linguagem de domínio simples e Python.

0

Ver anotações

## 7. Deep learning - Machine learning

- **Keras:** é uma biblioteca de rede neural profunda de código aberto.
- **TensorFlow:** é uma plataforma de código aberto de ponta a ponta para aprendizado de máquina, desenvolvida originalmente pela Google.
- **PyTorch:** é um pacote Python que fornece dois recursos de alto nível: i) computação de tensor (como NumPy) com forte aceleração de GPU; e ii) redes neurais profundas.
- **Scikit Learn:** módulo Python para aprendizado de máquina construído sobre o SciPy (SciPy é um software de código aberto para matemática, ciências e engenharia).

## 8. Biblioteca para jogos - PyGame

- **PyGame:** é uma biblioteca para a construção de aplicações gráficas e aplicação multimídia, utilizada para desenvolver jogos.

Além dessas categorias e bibliotecas citadas, como você já sabe, existem inúmeras outras. Você, como profissional desenvolvedor, deve buscá-las e estudar aquelas da área em que deseja atuar. A grande vantagem de usar bibliotecas é que elas encapsulam a complexidade de uma determinada tarefa, razão pela qual, com poucas linhas de códigos, conseguimos realizar tarefas complexas.

Aprenderemos a trabalhar com banco de dados em uma outra aula e teremos uma unidade inteira dedicada ao estudo da biblioteca Pandas. Para conhecermos um pouco do poder das bibliotecas em Python, vamos falar um pouco sobre o pacote *requests*.

0

Ver anotações

## BIBLIOTECA REQUESTS

A biblioteca **requests** habilita funcionalidades do protocolo HTTP, como o `get` e o `post`. Dentre seus métodos, o `get()` é o responsável por capturar informação da internet. A documentação sobre ela está disponível no endereço [https://requests.readthedocs.io/pt\\_BR/latest/](https://requests.readthedocs.io/pt_BR/latest/). Essa biblioteca foi construída com o intuito de substituir o módulo `urllib2`, que demanda muito trabalho para obter os resultados. O método `get()` permite que você informe a URL de que deseja obter informação. Sua sintaxe é: `requests.get('https://XXXXXXX')`. Para outros parâmetros dessa função, como autenticação, cabeçalhos, etc., consulte a documentação.

Observe o código a seguir. Na entrada 8, importamos a biblioteca *requests* e, na linha 3, usamos o método `get()` para capturar um conteúdo de uma API do github e guardar na variável *info*. Ao fazer uma requisição podemos olhar algumas informações da extração pela propriedade *headers*.

In [8]:

```
import requests

info = requests.get('https://api.github.com/events')
info.headers
```

Out[8]:

```
{'date': 'Thu, 04 Jun 2020 22:09:33 GMT', 'content-type': 'application/json; charset=utf-8', 'server': 'GitHub.com', 'status': '200 OK', 'cache-control': 'public, max-age=60, s-maxage=60', 'vary': 'Accept, Accept-Encoding, Accept, X-Requested-With, Accept-Encoding', 'etag': 'W/"078bb18598ef42449c62d7d39a8f303a"', 'last-modified': 'Thu, 04 Jun 2020 22:04:33 GMT', 'x-poll-interval': '60', 'x-github-media-type': 'github.v3; format=json', 'link': '<https://api.github.com/events?page=2>; rel="next", <https://api.github.com/events?page=10>; rel="last"', 'access-control-expose-headers': 'ETag, Link, Location, Retry-After, X-GitHub-OTP, X-RateLimit-Limit, X-RateLimit-Remaining, X-RateLimit-Reset, X-OAuth-Scopes, X-Accepted-OAuth-Scopes, X-Poll-Interval, X-GitHub-Media-Type, Deprecation, Sunset', 'access-control-allow-origin': '*', 'strict-transport-security': 'max-age=31536000; includeSubdomains; preload', 'x-frame-options': 'deny', 'x-content-type-options': 'nosniff', 'x-xss-protection': '1; mode=block', 'referrer-policy': 'origin-when-cross-origin, strict-origin-when-cross-origin', 'content-security-policy': "default-src 'none'", 'content-encoding': 'gzip', 'X-Ratelimit-Limit': '60', 'X-Ratelimit-Remaining': '58', 'X-Ratelimit-Reset': '1591310892', 'Accept-Ranges': 'bytes', 'Transfer-Encoding': 'chunked', 'X-GitHub-Request-Id': 'EAF6:7629:700E8:A32A6:5ED9711D'}
```

0

Ver anotações

In [9]:

```
print(info.headers['date']) # Data de extração  
print(info.headers['server']) # Servidor de origem  
print(info.headers['status']) # Status HTTP da extração, 200 é ok  
print(info.encoding) # Encoding do texto  
print(info.headers['last-modified']) # Data da última modificação da informação
```

```
Thu, 04 Jun 2020 22:09:33 GMT  
GitHub.com  
200 OK  
utf-8  
Thu, 04 Jun 2020 22:04:33 GMT
```

A propriedade *headers* retorna um dicionário de informações. Veja que, na entrada 9, extraímos algumas informações dessa propriedade. Na linha 1, acessamos a data de extração; na linha 2, o servidor que foi acessado; na linha 3, o status da extração; na linha 4, a decodificação texto; e na linha 5, a data da última modificação da informação no servidor. Veja que essas informações podem ser usadas em um relatório!

Para acessar o conteúdo que foi extraído, podemos usar a propriedade *text*, que converte todo o conteúdo para uma string, ou então o método *json()*, que faz conversão para uma lista de dicionários. Observe o código a seguir. Na entrada 10, temos o conteúdo como uma string e, na entrada 11, o conteúdo como uma lista de dicionários. Nesse caso, bastaria usar a linguagem Python para fazer os devidos tratamentos e extrair as informações!

Ver anotações

In [10]:

```
texto_str = info.text
print(type(texto_str))
texto_str[:100] # exibe somente os 100 primeiros caracteres
```

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

Out[10]:

```
'[{"id": "12535753096", "type": "PushEvent", "actor": {"id": 5858581, "login": "grahamegrieve", "display_login": "grahamegrieve"}]
```

In [11]:

```
texto_json = info.json()
print(type(texto_json))
texto_json[0]
```

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

Out[11]:

```
{'id': '12535753096',
 'type': 'PushEvent',
 'actor': {'id': 5858581,
            'login': 'grahamegrieve',
            'display_login': 'grahamegrieve',
            'gravatar_id': '',
            'url': 'https://api.github.com/users/grahamegrieve',
            'avatar_url': 'https://avatars.githubusercontent.com/u/5858581?'},
 'repo': {'id': 178767413,
           'name': 'HL7/fhir-ig-publisher',
           'url': 'https://api.github.com/repos/HL7/fhir-ig-publisher'},
 'payload': {'push_id': 5179281979,
             'size': 1,
             'distinct_size': 1,
             'ref': 'refs/heads/master',
             'head': 'c76e1dbce501f23988ad4d91df705942ca9b978f',
             'before': '3c6af8f114145351d82d36f506093a542470db0a',
             'commits': [{"sha": "c76e1dbce501f23988ad4d91df705942ca9b978f",
                          'author': {'email': 'grahameg@gmail.com', 'name': 'Grahame Grieve'},
                          'message': '* add -no-sushi command',
                          'distinct': True,
                          'url': 'https://api.github.com/repos/HL7/fhir-ig-publisher/commits/c76e1dbce501f23988ad4d91df705942ca9b978f'}]},
             'public': True,
             'created_at': '2020-06-04T22:04:33Z',
             'org': {'id': 21250901,
                     'login': 'HL7',
                     'gravatar_id': '',
                     'url': 'https://api.github.com/orgs/HL7',
                     'avatar_url': 'https://avatars.githubusercontent.com/u/21250901?'}}
```

Ver anotações

## EXEMPLIFICANDO

Vamos utilizar a biblioteca *requests* para extrair informações da Copa do Mundo de Futebol Feminino, que aconteceu no ano de 2019. As informações estão disponíveis no endereço <http://worldcup.sfg.io/matches>, no formato chave:valor. Após extrair as informações, vamos gerar um relatório que contém informações de cada jogo no seguinte formato: (dia/mes/ano) - time 1 x time 2 = gols time 1 a gols time 2. Então, vamos lá!

0

Ver anotações

In [12]:

```
# primeiro passo extrair as informações com o request utilizando o método json().  
  
import requests  
import datetime as dt  
  
jogos = requests.get('http://worldcup.sfg.io/matches').json()  
print(type(jogos))  
  
<class 'list'>
```

Na entrada 12, foi realizada a extração com o *requests* e já convertemos o conteúdo para *json()*. Observe que, como resultado, temos uma lista de dicionários. Na linha 7, estamos extraíndo as informações do primeiro dicionário da lista, ou seja, as informações do primeiro jogo. Nossa missão é criar uma lógica que extraia as informações de cada jogo, conforme solicitado, e gere um relatório. Então vamos a lógica para extrair.

In [13]:

```
# segundo passo: percorrer cada dicionário da lista (ou seja, cada jogo)
extraindo as informações

info_relatorio = []
file = open('relatorio_jogos.txt', "w") # cria um arquivo txt na pasta em
que está trabalhando.

for jogo in jogos:
    data = jogo['datetime'] # extrai a data
    data = dt.datetime.strptime(data, "%Y-%m-%dT%H:%M:%SZ") # converte de
string para data
    data = data.strftime("%d/%m/%Y") # formata

    nome_time1 = jogo['home_team_country']
    nome_time2 = jogo['away_team_country']

    gols_time1 = jogo['home_team']['goals']
    gols_time2 = jogo['away_team']['goals']

    linha = f"({data}) - {nome_time1} x {nome_time2} = {gols_time1} a
{gols_time2}"
    file.write(linha + '\n') # escreve a linha no arquivo txt
    info_relatorio.append(linha)

file.close() # é preciso fechar o arquivo
info_relatorio[:5]
```

Out[13]:

```
['(07/06/2019) - France x Korea Republic = 4 a 0',
 '(08/06/2019) - Germany x China PR = 1 a 0',
 '(08/06/2019) - Spain x South Africa = 3 a 1',
 '(08/06/2019) - Norway x Nigeria = 3 a 0',
 '(09/06/2019) - Brazil x Jamaica = 3 a 0']
```

0

Ver anotações

Na entrada 13, usamos uma estrutura de repetição para percorrer cada item do dicionário, extraíndo as informações. Chamamos a atenção para as linhas 13 e 14, nas quais usamos duas chaves. Isso foi feito porque, dentro do dicionário, existe outro dicionário. Veja: **'home\_team': {'country': 'France', 'code': 'FRA', 'goals': 4, 'penalties': 0}**. Dentro da chave *home\_team* existe um outro dicionário. Portanto, para acessar os gols, precisamos também acessar a chave interna *goals*, ficando então **jogo['home\_team']['goals']**.

Na entrada 13, usamos a função *built-in open()* para criar um arquivo chamado *relatorio\_jogos.txt*, no qual escreveremos informações – por isso o parâmetro "w". Na linha 18, escrevemos cada linha gerada no arquivo, concatenando com uma nova linha "\n" a cada informação gerada. Como passamos somente o nome do arquivo, ele será gerado na pasta onde estiver trabalhando.

0

Ver anotações

## MATPLOTLIB

Matplotlib é uma biblioteca com funcionalidades para criar gráficos, cuja documentação está disponível no endereço <https://matplotlib.org/>. É composta por uma série de exemplos. Vamos utilizar a interface Pyplot para criar um gráfico simples baseado nas informações que salvamos sobre os jogos da Copa do Mundo de Futebol Feminino de 2019.

Fizemos a extração e criamos um relatório, salvando-o como *relatorio\_jogos.txt*. Agora vamos ler os dados que foram persistidos no arquivo, extrair somente as datas no formato *dd/mm* e contabilizar quantos jogos aconteceram em cada data. Em seguida, vamos usar o Pyplot para construir esse gráfico de contagem.

In [14]:

```
# ler o arquivo salvo  
file = open('relatorio_jogos.txt', 'r')  
print('file = ', file, '\n')  
info_relatorio = file.readlines()  
file.close()  
  
print("linha 1 = ", info_relatorio[0])  
file = <_io.TextIOWrapper name='relatorio_jogos.txt' mode='r'  
encoding='cp1252'>
```

```
linha 1 = (07/06/2019) - France x Korea Republic = 4 a 0
```

0

Ver anotações

Para ler um arquivo, usamos a função *built-in open()*, passando como parâmetros o nome do arquivo e a opção 'r', que significa que queremos abrir o arquivo em modo leitura (r = read). A função *open()* retorna um objeto do tipo "*\_io.TextIOWrapper*", conforme podemos observar pelo print na linha 3. Para acessar o conteúdo do arquivo, precisamos usar a função *readlines()*, que cria uma lista, na qual cada elemento é uma linha do arquivo. Após a criação da lista, podemos fechar o arquivo. Na linha 7, imprimimos o primeiro item da lista criada, que corresponde à primeira linha do arquivo de que fizemos a leitura. Para cada linha, queremos somente a parte que corresponde ao dia e mês: dd/mm, razão pela qual vamos criar uma nova lista que contém somente essas datas. Observe o código a seguir.

In [15]:

```
# Extrair somente a parte 'dd/mm' da linha  
datas = [linha[1:6] for linha in info_relatorio]  
print(sorted(datas))
```

```
[ '02/07', '03/07', '06/07', '07/06', '07/07', '08/06', '08/06', '08/06',
'09/06', '09/06', '09/06', '10/06', '10/06', '11/06', '11/06', '11/06', '12/06',
'12/06', '12/06', '13/06', '13/06', '14/06', '14/06', '14/06', '15/06', '15/06',
'16/06', '16/06', '17/06', '17/06', '17/06', '17/06', '18/06', '18/06', '19/06',
'19/06', '20/06', '20/06', '20/06', '20/06', '22/06', '22/06', '23/06', '23/06',
'24/06', '24/06', '25/06', '25/06', '27/06', '28/06', '29/06', '29/06' ]
```

[Ver anotações](#)

Agora temos uma lista com todas as datas dos jogos e precisamos contar quantas vezes cada uma aparece. Assim vamos ter a quantidade de jogos por dia. Para fazer esse trabalho, vamos utilizar a operação `count()` disponível para os objetos do tipo sequência: `sequencia.count(valor)`, que retorna quantas vezes o valor aparece na sequência. Então, se fizermos `datas.count('08/06')`, temos que obter o valor 3. Uma vez que precisamos fazer isso para todas as datas, vamos, então, usar uma list comprehension para fazer essa iteração. Para cada data, vamos gerar uma tupla com dois valores: (data, count). Em nossa lista final, queremos ter uma linha para cada data, exemplo: ('08/06', 3). Então, para remover as duplicações, vamos utilizar o construtor `set()`. Observe o código a seguir:

In [16]:

```
datas_count = [(data, datas.count(data)) for data in set(datas)]
print(datas_count)
```

```
[('17/06', 4), ('22/06', 2), ('24/06', 2), ('08/06', 3), ('07/06', 1), ('18/06',
2), ('03/07', 1), ('07/07', 1), ('27/06', 1), ('28/06', 1), ('25/06', 2),
('09/06', 3), ('20/06', 4), ('13/06', 2), ('12/06', 3), ('19/06', 2), ('11/06',
3), ('14/06', 3), ('16/06', 2), ('10/06', 2), ('29/06', 2), ('02/07', 1),
('23/06', 2), ('15/06', 2), ('06/07', 1)]
```

Com o passo anterior, temos uma lista de tuplas com a data e quantidade de jogos! Por uma questão de conveniência, vamos transformar essa lista em um dicionário usando o construtor `dict()`. Veja a seguir.

In [17]:

```
datas_count = dict(datas_count)
print(datas_count)
```

```
{'17/06': 4, '22/06': 2, '24/06': 2, '08/06': 3, '07/06': 1, '18/06': 2,
'03/07': 1, '07/07': 1, '27/06': 1, '28/06': 1, '25/06': 2, '09/06': 3, '20/06
4, '13/06': 2, '12/06': 3, '19/06': 2, '11/06': 3, '14/06': 3, '16/06': 2,
'10/06': 2, '29/06': 2, '02/07': 1, '23/06': 2, '15/06': 2, '06/07': 1}
```

0

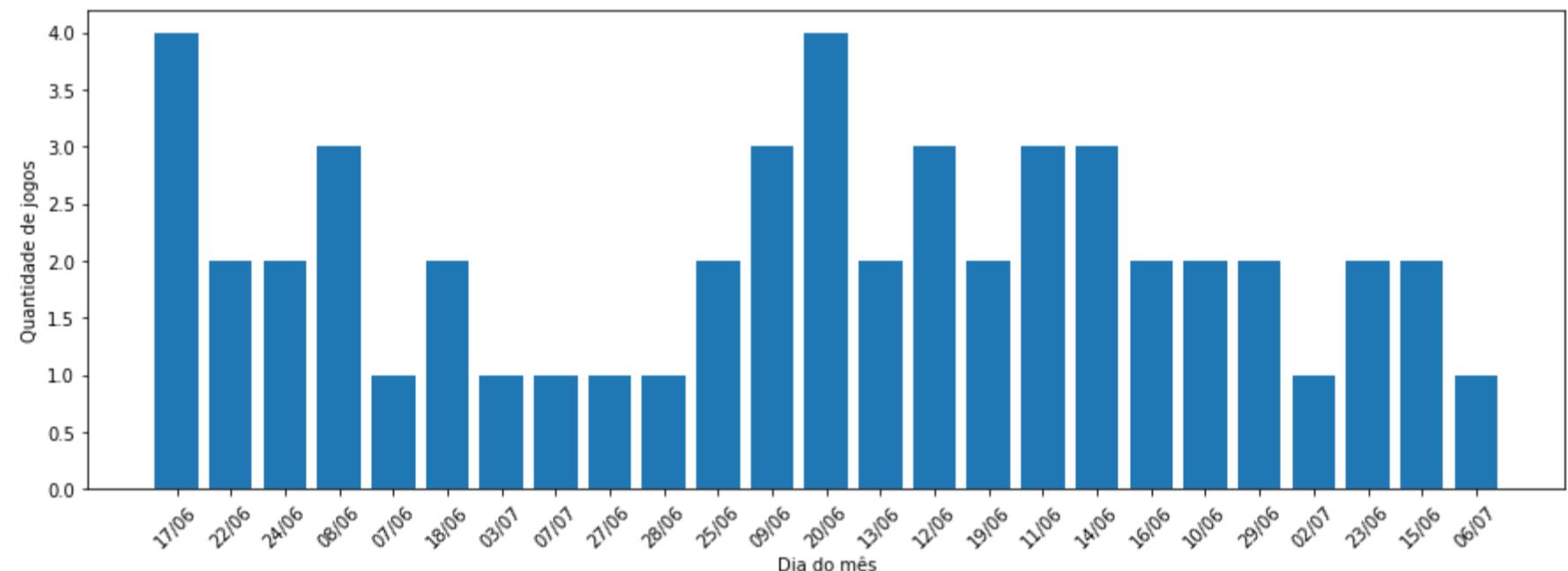
Ver anotações

Essa transformação da lista para dicionário nos permite extrair as chaves (que são as datas) e os valores (que são as quantidades). Esses dois itens serão usados nos eixos *x* e *y* do gráfico.

Agora que já preparamos os dados, vamos utilizar a interface Pyplot da biblioteca *matplotlib* para criar nosso gráfico. Para que possamos ver um gráfico dentro de um notebook, temos que habilitar a opção `%matplotlib inline` e importar a biblioteca. Fazemos isso nas linhas 1 e 2 da entrada 18. Nas linhas 4 e 5, usamos as informações do nosso dicionário para definir os dados que serão usados nos eixos *x* e *y*. Na linha 7, configuramos o tamanho do nosso gráfico. Nas linhas 8 e 9 definimos os rótulos dos eixos; na linha 10, configuramos uma rotação para as datas que vão aparecer no eixo *x*; a linha 12 é onde de fato criamos o gráfico, escolhendo a opção de barras (bar) e passando as informações a serem plotadas. Na linha 14, usamos a função *show* para exibir nosso gráfico.

In [18]:

```
%matplotlib inline  
import matplotlib.pyplot as plt  
  
eixo_x = datas_count.keys()  
eixo_y = datas_count.values()  
  
plt.figure(figsize=(15, 5))  
plt.xlabel('Dia do mês')  
plt.ylabel('Quantidade de jogos')  
plt.xticks(rotation=45)  
  
plt.bar(eixo_x, eixo_y)  
  
plt.show()
```



A quantidade de funcionalidades embutidas em uma biblioteca traz um universo de possibilidades ao desenvolvedor. Consulte sempre a documentação e os fóruns para acompanhar as novidades.

## MÓDULOS PRÓPRIOS

Os códigos podem ser organizados em diversos arquivos com extensão .py, ou seja, em módulos. Cada módulo pode importar outros módulos, tanto os pertencentes ao mesmo projeto, como os *built-in* ou de terceiros. A Figura 3.9, ilustra a modularidade em Python. Criamos um módulo (arquivo Python) chamado

0

Ver anotações

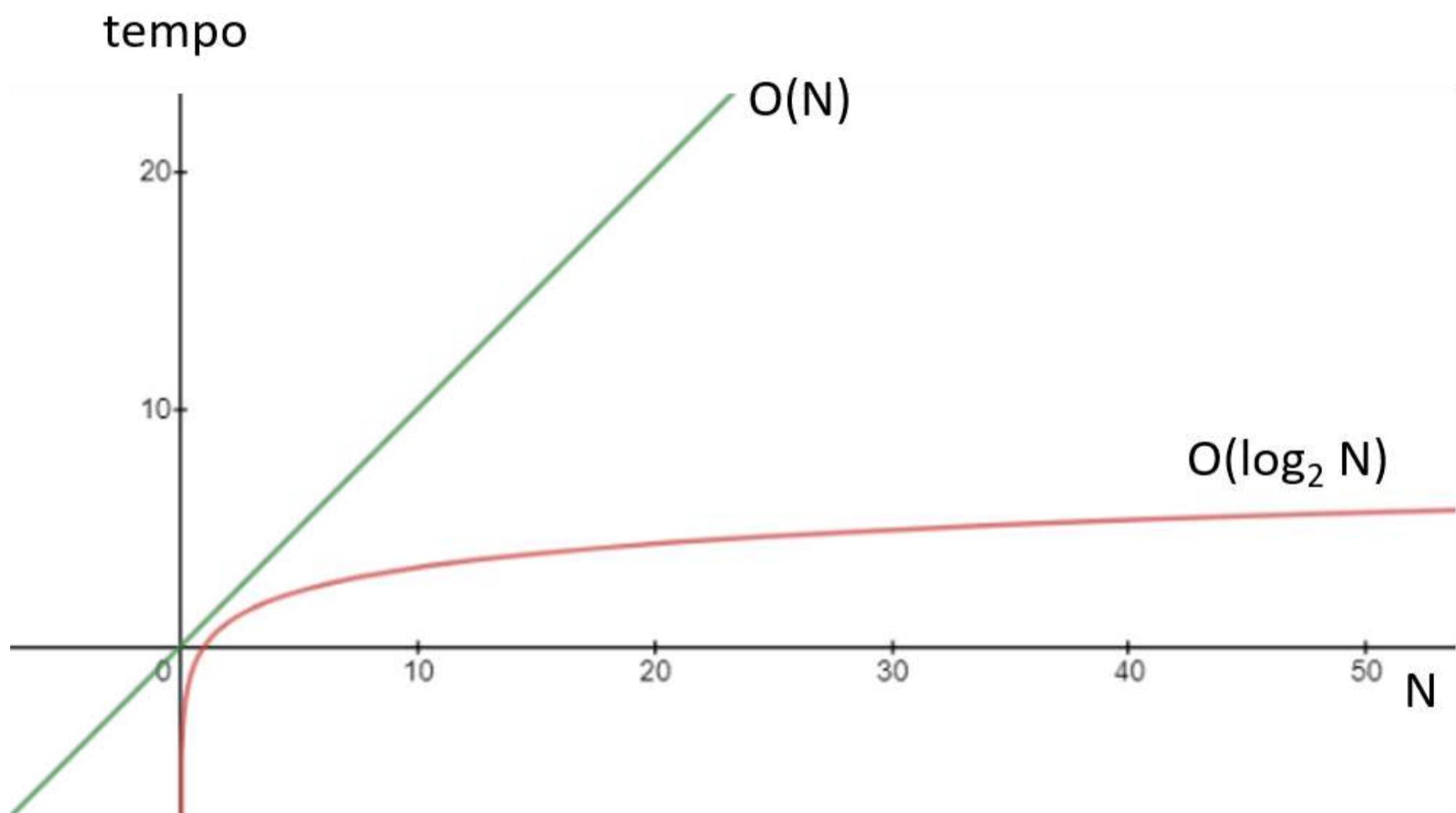
0

[Ver anotações](#)

*utils.py*. Esse módulo possui uma função que cria uma conexão *ssh* com um determinado servidor. Podemos entender um cliente *SSH* como um túnel de comunicação. Veja que no módulo precisamos usar a biblioteca *paramiko* para construir essa conexão. A função *create\_ssh\_client* retorna um *client*, ou seja, a conexão em si. Em um outro módulo, chamado principal, importamos a função do módulo *utils*. É dentro do módulo principal que vamos utilizar a funcionalidade da conexão para copiar um arquivo que está em um servidor para outro local. **É importante ressaltar que, da forma pela qual fizemos a importação, ambos os arquivos *.py* precisam estar no mesmo nível de pasta.**

Se precisarmos usar o módulo *utils* em vários projetos, é interessante transformá-lo em uma biblioteca e disponibilizá-la via PyPI.

Figura 3.9 | Módulo em Python



Fonte: elaborada pela autora.

Para finalizar, vamos aprender como transformar módulos em scripts que podem ser chamados via linha de comando. Ao criar um arquivo *.py*, pode-se utilizar o terminal do sistema operacional para executá-lo; por exemplo: `python`

`meu_programa.py`. Para que esse programa de fato execute, ele precisa ter a variável `_name_` com valor `"__main__"`, conforme mostra a Figura 3.10. Veja que dentro do `_name_ == "__main__"`, a função `main()` é invocada, fazendo com que o script execute.

0

Figura 3.10 | Módulos como scripts



Fonte: elaborada pela autora.

Ver anotações

## REFERÊNCIAS E LINKS ÚTEIS

API CNAE - Cadastro Nacional de Atividades Econômicas. **API e documentação**.

Versão: 2.0.0. CNAE, 2017. Disponível em: <https://bit.ly/3amaPGE>. Acesso em: 30 jul. 2020.

LJUBOMIR, P. **Introdução à computação usando Python**: um foco no desenvolvimento de aplicações. Rio de Janeiro: LTC, 2016.

PSF - Python Software Foundation. **Modules**. 2020b. Disponível em: <https://bit.ly/30SmzNC>. Acesso em: 04 jun. 2020.

PyPI. Python Package Index. Página inicial. 2020. Disponível em: <https://pypi.org/>. Acesso em: 04 jun. 2020.

RAMALHO, L. **Fluent Python**. Gravenstein: O'Reilly Media, 2014.

# FOCO NO MERCADO DE TRABALHO

## BIBLIOTECAS E MÓDULOS EM PYTHON

Vanessa Cadan Scheffer

0

Ver anotações

### BIBLIOTECA *REQUESTS*

A biblioteca requests habilita funcionalidades do protocolo HTTP, como o *get* e o *post*. Dentre seus métodos, o *get()* é o responsável por capturar informação da internet.



Fonte: Shutterstock.

### Deseja ouvir este material?

Áudio disponível no material digital.

### DESAFIO

No Brasil, existe um órgão responsável por gerar as estatísticas da atividade econômica no país. Para tal tarefa, as atividades são classificadas em grupos; por exemplo, as atividades do grupo 262 referem-se à fabricação de equipamentos de

informática e periféricos. "A CNAE, Classificação Nacional de Atividades Econômicas, é a classificação oficialmente adotada pelo Sistema Estatístico Nacional na produção de estatísticas por tipo de atividade econômica, e pela Administração Pública, na identificação da atividade econômica em cadastros e registros de pessoa jurídica." (API CNAE, 2017, [s.p.])

0

Ver anotações

Como desenvolvedor em uma empresa de consultoria de software, você foi alocado em um projeto com base no qual o cliente deseja automatizar a extração dos dados do CNAE e gerar um relatório. Os dados estão disponíveis no endereço <https://servicodados.ibge.gov.br/api/v2/cnae/classes>. Você deve extraí-los e gerar as seguintes informações:

- Quantas atividades distintas estão registradas?
- Quantos grupos de atividades existem?
- Quantas atividades estão cadastradas em cada grupo?
- Qual grupo ou quais grupos possuem o maior número de atividades vinculadas?

## RESOLUÇÃO

Para automatizar o processo de extração dos dados do CNAE e gerar o relatório, vamos ter de usar bibliotecas. Para fazer a extração dos dados do CNAE, podemos usar a biblioteca *requests*. Para responder às perguntas, vamos precisar manipular listas e dicionários. Então vamos começar pela extração.

In [19]:

```
import requests

dados =
    requests.get('https://servicodados.ibge.gov.br/api/v2/cnae/classes').json() #
resulta em uma lista de diconários
dados[0] # exibindo o primeiro registro de dados (primeiro dicionário da lista)
```

Out[19]:

```
{'id': '01113',
'descricao': 'CULTIVO DE CEREAIS',
'grupo': {'id': '011',
'descricao': 'PRODUÇÃO DE LAVOURAS TEMPORÁRIAS',
'divisao': {'id': '01',
'descricao': 'AGRICULTURA, PECUÁRIA E SERVIÇOS RELACIONADOS',
'secao': {'id': 'A',
'descricao': 'AGRICULTURA, PECUÁRIA, PRODUÇÃO FLORESTAL, PESCA E
AQÜICULTURA'}}},
'observacoes': ['Esta classe compreende - o cultivo de alpiste, arroz, aveia,
centeio, cevada, milho, milheto, painço, sorgo, trigo, trigo preto, triticale e
outros cereais não especificados anteriormente',
'Esta classe compreende ainda - o beneficiamento de cereais em estabelecimento
agrícola, quando atividade complementar ao cultivo\r\n- a produção de sementes
de cereais, quando atividade complementar ao cultivo',
'Esta classe NÃO compreende - a produção de sementes certificadas dos cereais
desta classe, inclusive modificadas geneticamente (01.41-5)\r\n- os serviços de
preparação de terreno, cultivo e colheita realizados sob contrato (01.61-0)\r\n-
o beneficiamento de cereais em estabelecimento agrícola realizado sob contrato
(01.63-6)\r\n- o processamento ou beneficiamento de cereais em estabelecimento
não-agrícola (grupo 10.4) e (grupo 10.6)\r\n- a produção de biocombustível
(19.32-2)']}
```

Ver anotações

Agora que temos os dados guardados em uma lista de dicionários, podemos usar a função *built-in len()* para saber quantos elementos essa lista tem. Esse resultado será a quantidade de dicionários que representa a quantidade distintas de

atividades.

In [20]:

```
# Quantidade distintas de atividades, basta saber o tamanho da lista.  
  
qtde_atividades_distintas = len(dados)
```

0

Ver anotações

Para saber quantos grupos de atividades existem e já começar a preparar os dados para os próximos passos, vamos criar uma lista que percorre cada registro e extrai a informação do grupo. Dado um registro, essa informação está na chave interna 'descricao' da chave externa 'grupo'. Logo, para acessar, temos que usar a sintaxe: *dicionario['chave\_externa']['chave\_interna']*. Na entrada 21, criamos uma lista vazia na linha 3 e, dentro da estrutura de repetição, vamos extraíndo a informação e guardando-a na lista.

In [21]:

```
# Criar uma lista dos grupos de atividades, extraíndo a descrição de cada registro  
  
grupos = []  
for registro in dados:  
    grupos.append(registro['grupo']['descricao'])  
  
grupos[:10]
```

Out[21]:

```
[ 'PRODUÇÃO DE LAVOURAS TEMPORÁRIAS' ,  
  'HORTICULTURA E FLORICULTURA' ,  
  'HORTICULTURA E FLORICULTURA' ,  
  'EXTRAÇÃO DE MINERAIS METÁLICOS NÃO-FERROSOS' ]
```

0

Ver anotações

Agora que temos uma lista com todos os grupos, para saber quantos grupos distintos existem, basta eliminar as duplicações e fazer a contagem. Na entrada 22, usamos o construtor `set()` para criar um conjunto de dados, sem repetições e sem alterar a lista com todos, uma vez que ainda vamos utilizá-la. O resultado do `set()` fazemos a contagem com a função `len()`, obtendo, então, a quantidade de grupos distintos.

In [22]:

```
# A partir da lista, podemos extrair a quantidade de grupos de atividades  
  
qtde_grupos_distintos = len(set(grupos)) # o construtor set cria uma estrutura  
de dados removendo as duplicações.
```

Agora vamos contar quantas atividades estão cadastradas em cada grupo. O código na entrada 23 faz esse trabalho. Usamos uma list comprehension para criar uma lista de tuplas. Cada tupla vai conter o grupo e a contagem de quantas vezes esse grupo aparece na lista de grupos: `(grupo, grupos.count(grupo))`. Isso será feito para cada grupo distinto: `for grupo in set(grupos)`.

In [23]:

```
# Resultado é uma lista de tuplas. Cria uma nova lista com o grupo e a
quantidade de atividades pertencentes a ele

grupos_count = [(grupo, grupos.count(grupo)) for grupo in set(grupos)]
grupos_count[:5]
```

Out[23]:

```
[('TECELAGEM, EXCETO MALHA', 3),
 ('COMÉRCIO ATACADISTA DE PRODUTOS DE CONSUMO NÃO-ALIMENTAR', 8),
 ('ATIVIDADES DE ORGANIZAÇÕES ASSOCIATIVAS PATRONAIS, EMPRESARIAIS E
PROFISSIONAIS',
 2),
 ('SEGURIDADE SOCIAL OBRIGATÓRIA', 1),
 ('FABRICAÇÃO DE ELETRODOMÉSTICOS', 2)]
```

0

Ver anotações

Para sabermos qual grupo ou quais grupos possuem o maior número de atividades vinculadas, vamos transformar a lista de tuplas em um dicionário.

In [24]:

```
# Por conveniência, transformamos a lista em um dicionário

grupos_count = dict(grupos_count)
```

Agora podemos criar uma nova lista que contém todos os grupos que possuem a contagem com o mesmo valor da quantidade máxima que encontramos. Ao usar dicionário, conseguimos acessar a chave e o valor, o que facilita o trabalho.

In [25]:

```
# A partir do dicionário vamos descobrir qual (ou quais) grupos possuem mais  
atividades
```

```
valor_maximo = max(grupos_count.values())  
grupos_mais_atividades = [chave for (chave, valor) in grupos_count.items() if  
valor == valor_maximo]  
print(len(grupos_mais_atividades))  
grupos_mais_atividades
```

```
1
```

0

Ver anotações

Out[25]:

```
['REPRESENTANTES COMERCIAIS E AGENTES DO COMÉRCIO, EXCETO DE VEÍCULOS  
AUTOMOTORES E MOTOCICLETAS']
```

Para formalizar a entrega do componente de extração, que tal criar uma classe e um método com todo o código? Assim, quando for preciso extrair, basta instanciar a classe e invocar o método.

In [26]:

```
import requests

from datetime import datetime

class ETL:

    def __init__(self):
        self.url = None

    def extract_cnae_data(self, url):
        self.url = url
        data_extracao = datetime.today().strftime("%Y/%m/%d - %H:%M:%S")
        # Faz extração
        dados = requests.get(self.url).json()

        # Extrai os grupos dos registros
        grupos = []
        for registro in dados:
            grupos.append(registro['grupo']['descricao'])

        # Cria uma lista de tuplas (grupo, quantidade_atividades)
        grupos_count = [(grupo, grupos.count(grupo)) for grupo in set(grupos)]
        grupos_count = dict(grupos_count) # transforma a lista em dicionário

        valor_maximo = max(grupos_count.values()) # Captura o valor máximo de
atividades
        # Gera uma lista com os grupos que possuem a quantidade máxima de
atividades
        grupos_mais_atividades = [chave for (chave, valor) in
grupos_count.items() if valor == valor_maximo]

        # informações
        qtde_atividades_distintas = len(dados)
        qtde_grupos_distintos = len(set(grupos))
```

0

Ver anotações

```
print(f"Dados extraídos em: {data_extracao}")  
print(f"Quantidade de atividades distintas =  
{qtde_atividades_distintas}")  
print(f"Quantidade de grupos distintos = {qtde_grupos_distintos}")  
print(f"Grupos com o maior número de atividades =  
{grupos_mais_atividades}, atividades = {valor_maximo}")  
  
return None
```

0

Ver anotações

In [27]:

```
# Usando a classe ETL  
  
ETL().extract_cnae_data('https://servicodados.ibge.gov.br/api/v2/cnae/classes')
```

```
Dados extraídos em: 2020/06/04 - 19:09:39  
Quantidade de atividades distintas = 673  
Quantidade de grupos distintos = 285  
Grupos com o maior número de atividades = ['REPRESENTANTES COMERCIAIS E AGENTES  
DO COMÉRCIO, EXCETO DE VEÍCULOS AUTOMOTORES E MOTOCICLETAS'], atividades = 9
```

## DESAFIO DA INTERNET

Ganhar habilidade em programação exige estudo e treino (muito treino). Acesse a biblioteca virtual no endereço <http://biblioteca-virtual.com/> e busque pelo livro a seguir referenciado. Na página 198, no Capítulo 6 da referida obra, você encontrará o problema prático 6.9 (*Implemente a função adivinhe()*). Que tal tentar resolver esse desafio?

Ver anotações

LJUBOMIR, P. **Introdução à computação usando Python**: um foco no desenvolvimento de aplicações. Rio de Janeiro: LTC, 2016.

NÃO PODE FALTAR

## APLICAÇÃO DE BANCO DE DADOS COM PYTHON

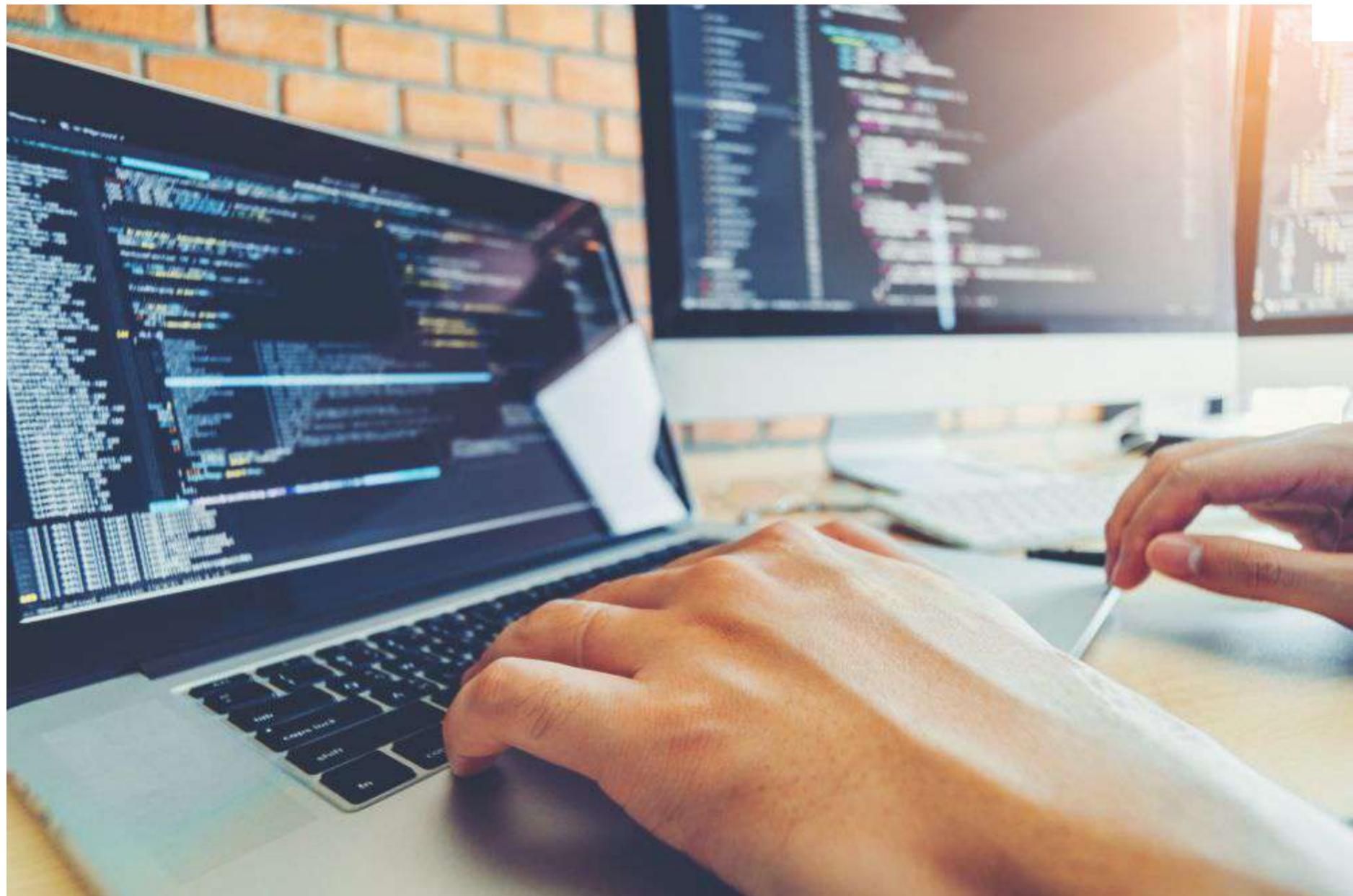
Vanessa Cadan Scheffer

0

Ver anotações

### LINGUAGEM DE CONSULTA ESTRUTURADA - SQL

O SQL é a linguagem que permite aos usuários se comunicarem com banco de dados relacionais.



Fonte: Shutterstock.

### Deseja ouvir este material?

Áudio disponível no material digital.

### INTRODUÇÃO A BANCO DE DADOS

Grande parte dos softwares que são desenvolvidos (se não todos) acessa algum tipo de mecanismo para armazenar dados. Podem ser dados da aplicação, como

~~cadastros de clientes, ou então dados sobre a execução da solução, os famosos~~

logs. Esses dados podem ser armazenados em arquivos, cenário no qual se destacam os arquivos delimitados, com extensão CSV (*comma separated values*), os arquivos JSON (*JavaScript Object Notation*) e os arquivos XML (*Extensible Markup Language*). Outra opção para persistir os dados é utilizar um sistema de banco de dados.

o

Ver anotações

Segundo Date (2003), um sistema de banco de dados é basicamente apenas um sistema computadorizado de persistência de registros. O próprio banco de dados pode ser considerado como um repositório para uma coleção de dados computadorizados. Os sistemas de banco de dados podem ser divididos em duas categorias: banco de dados relacional e banco de dados NoSQL.

A teoria base dos bancos de dados relacional existe desde a década de 1970 (MACHADO, 2014). Nessa abordagem, os dados são persistidos em uma estrutura bidimensional, chamada de relação (que é uma tabela), que está baseada na teoria dos conjuntos pertencentes à matemática. Cada unidade de dados é conhecida como coluna, ao passo que cada unidade do grupo é conhecida como linha, tupla ou registro.

Com o surgimento de grandes aplicações web e de outras tecnologias como o IoT, o volume de dados que trafegam na rede e são processados aumentou consideravelmente, o que pode ser um desafio em termos de performance para os bancos relacionais. Além do volume, o formato dos dados também é desafiador para a persistência nessa tecnologia, uma vez que eles são persistidos em linhas e colunas.

Como armazenar fotos, vídeos, e outros formatos? Para suprir essa nova demanda, pesquisadores se dedicaram a buscar soluções para o manuseio de dados em grande escala e em outros formatos não estruturados, obtendo, como um dos resultados, o banco de dados não relacional, que geralmente é referenciado como NoSQL (VAISH, 2013). Existe uma discussão sobre o significado de NoSQL: em algumas literaturas, ele é tratado como *not only SQL* (não somente SQL), o que remete à possibilidade de existir também o SQL nessa tecnologia. Mas, segundo

Vaish (2013), originalmente, NoSQL era a combinação das palavras *No* e *SQL*, que literalmente dizia respeito a não usar a linguagem SQL (*structured query language* - linguagem de consulta estruturada) para acessar e manipular dados em sistemas gerenciadores de banco de dados. Qualquer que seja a origem do termo, hoje o NoSQL é usado para abordar a classe de bancos de dados que não seguem os princípios do sistema de gerenciamento de banco de dados relacional (RDBMS) e são projetados especificamente para lidar com a velocidade e a escala de aplicações como Google, Facebook, Yahoo, Twitter, dentre outros.

Ver anotações

## | LINGUAGEM DE CONSULTA ESTRUTURADA - SQL

Para se comunicar com um banco de dados relacional, existe uma linguagem específica conhecida como SQL, que, tal como dito, significa *structured query language* ou, traduzindo, *linguagem de consulta estruturada*. Em outras palavras, SQL é a linguagem que permite aos usuários se comunicarem com banco de dados relacionais (ROCKOFF, 2016). Em 1986, o American National Standards Institute (ANSI) publicou seu primeiro conjunto de padrões em relação à linguagem SQL e, desde então, passou por várias revisões. Algumas empresas de software para banco de dados, como Oracle e Microsoft, adaptaram a linguagem SQL adicionando inúmeras extensões e modificações à linguagem padrão. Mas, embora cada fornecedor tenha sua própria interpretação exclusiva do SQL, ainda existe uma linguagem base que é padrão a todos fornecedores.

As instruções da linguagem SQL são divididas em três grupos: DDL, DML, DCL (ROCKOFF, 2016), descritos a seguir.

- **DDL** é um acrônimo para *Data Definition Language* (linguagem de definição de dados). Fazem parte deste grupo as instruções destinadas a criar, deletar e modificar banco de dados e tabelas. Neste módulo vão aparecer comandos como CREATE, o ALTER e o DROP.

- **DML** é um acrônimo para *Data Manipulation Language* (linguagem de manipulação de dados). Fazem parte deste grupo as instruções destinadas a recuperar, atualizar, adicionar ou excluir dados em um banco de dados. Neste módulo vão aparecer comandos como INSERT, UPDATE e DELETE.
- **DCL** é um acrônimo para *Data Control Language* (linguagem de controle de dados). Fazem parte deste grupo as instruções destinadas a manter a segurança adequada para o banco de dados. Neste módulo vão aparecer comandos como GRANT e REVOKE.

**DICA**

Como sugestão de literatura para noções de SQL, uma opção é o *Guia mangá de bancos de dados* (TAKAHASHI; AZUMA, 2009).

## BANCO DE DADOS RELACIONAL

### CONEXÃO COM BANCO DE DADOS RELACIONAL

Ao criar uma aplicação em uma linguagem de programação que precisa acessar um sistema gerenciador de banco de dados relacional (RDBMS), uma vez que são processos distintos, é preciso criar uma conexão entre eles. Após estabelecida a conexão, é possível (de alguma forma) enviar comandos SQL para efetuar as ações no banco (RAMAKRISHNAN; GEHRKE, 2003). Para fazer a conexão e permitir que uma linguagem de programação se comunique com um banco de dados com a utilização da linguagem SQL, podemos usar as tecnologias ODBC (Open Database Connectivity) e JDBC (Java Database Connectivity).

**“**

Ambos, ODBC e JDBC, expõem os recursos de banco de dados de uma forma padronizada ao programador de aplicativo através de uma interface de programação de aplicativo (API — application programming interface)

— RAMAKRISHNAN; GEHRKE, 2003, p. 162.

A grande vantagem de utilizar as tecnologias ODBC ou JDBC está no fato de que uma aplicação pode acessar diferentes RDBMS sem precisar recompilar o código. Essa transparência é possível porque a comunicação direta com o RDBMS é feita por um driver. Um driver é um software específico responsável por traduzir as chamadas ODBC e JDBC para a linguagem do RDBMS.

O JDBC é uma API padrão em Java, inicialmente desenvolvida pela Sun Microsystems (MENON, 2005). Em outras palavras, JDBC é um conjunto de classes desenvolvidas em Java que abstraem a conexão com um RDBMS. Cada fornecedor de RDBMS, como Oracle e Microsoft, constrói e distribui, gratuitamente, um driver JDBC. Diferentes RDBMS necessitam de diferentes drivers para comunicação; por exemplo, em uma aplicação que se conecta a três RDBMS distintos, serão necessários três drivers distintos. ODBC também é uma API padronizada para conexão com os diversos RDBMS (MICROSOFT, 2017). As funções na API ODBC são implementadas por desenvolvedores de drivers específicos do RDBMS e, para utilizá-las, você deve configurar uma entrada nas propriedades do sistema.

o

Ver anotações

## CONEXÃO DE BANCO DADOS SQL EM PYTHON

Nesta aula, vamos explorar como utilizar um banco de dados relacional em Python. Agora que já sabemos que para acessar esse tipo de tecnologia precisamos de um mecanismo de conexão (ODBC ou JDBC) e uma linguagem para nos comunicarmos com ele (SQL), vamos ver como atuar em Python.

Para se comunicar com um RDBMS em Python, podemos utilizar bibliotecas já disponíveis, com uso das quais, por meio do driver de um determinado fornecedor, será possível fazer a conexão e a execução de comandos SQL no banco. Por exemplo, para se conectar com um banco de dados Oracle, podemos usar a biblioteca cx-Oracle, ou, para se conectar a um PostgreSQL, temos como opção o psycopg2. Visando à padronização entre todos os módulos de conexão com um RDBMS e o envio de comandos, o **PEP 249 (Python Database API Specification)**

v2.0) elenca um conjunto de regras que os fornecedores devem seguir na construção de módulos para acesso a banco de dados. Por exemplo, a documentação diz que todos módulos devem implementar o método `connect(parameters...)` para se conectar a um banco. Veja que, dessa forma, caso seja necessário alterar o banco de dados, somente os parâmetros mudam, não o código.

o

Ver anotações

Agora que já temos essa visão geral, vamos explorar a implementação em Python usando um mecanismo de banco de dados SQL chamado SQLite.

## BANCO DE DADOS SQLITE

“

"O SQLite é uma biblioteca em linguagem C, que implementa um mecanismo de banco de dados SQL pequeno, rápido, independente, de alta confiabilidade e completo"

— SQLITE, 2020, [s.p.], tradução nossa.

Essa tecnologia pode ser embutida em telefones celulares e computadores e vem incluída em inúmeros outros aplicativos que as pessoas usam todos os dias. Ao passo que a maioria dos bancos de dados SQL usa um servidor para rodar e gerenciar, o SQLite não possui um processo de servidor separado. O SQLite lê e grava diretamente em arquivos de disco, ou seja, um banco de dados SQL completo com várias tabelas, índices, triggers e visualizações está contido em um único arquivo de disco.

O interpretador Python possui o módulo *built-in sqlite3*, que permite utilizar o mecanismo de banco de dados SQLite.

“

O módulo `sqlite3` foi escrito por Gerhard Häring e fornece uma interface SQL compatível com a especificação DB-API 2.0 descrita pelo PEP 249.

— PSF, 2020d, [s.p.], tradução nossa.

Para permear nosso estudo, vamos criar um banco de dados chamado *aulaDB*, no qual vamos criar a tabela fornecedor, conforme Quadro 3.1:

Quadro 3.1 | Tabela fornecedor

Campo	Tipo	Obrigatório
id_fornecedor	inteiro	sim
nome_fornecedor	texto	sim
cnpj	texto	sim
cidade	texto	não
estado	texto	sim
cep	texto	sim
data_cadastro	data	sim

0

Ver anotações

Fonte: elaborado pelo autora.

## | CRIANDO UM BANCO DE DADOS

O primeiro passo é importar o módulo *sqlite3*. Como o módulo está baseado na especificação DB-API 2.0 descrita pelo PEP 249, ele utiliza o método *connect()* para se conectar a um banco de dados. Em razão da natureza do SQLite (ser um arquivo no disco rígido), ao nos conectarmos a um banco, o arquivo é imediatamente criado na pasta do projeto (se estiver usando o projeto Anaconda, o arquivo é criado na mesma pasta em que está o Jupyter Notebook). Se desejar criar o arquivo em outra pasta, basta especificar o caminho juntamente com o nome, por exemplo: C:/Users/Documents/meu\_projeto/meus\_bancos/bancoDB.db. Observe do código a seguir.

In [1]:

```
import sqlite3

conn = sqlite3.connect('aulaDB.db')
print(type(conn))
<class 'sqlite3.Connection'>
```

0

Ver anotações

Ao executar o código da entrada 1, o arquivo é criado, e a variável "conn" agora é um objeto da classe *Connection* pertencente ao módulo *sqlite3*.

## CRIANDO UMA TABELA

Agora que temos uma conexão com um banco de dados, vamos utilizar uma instrução DDL da linguagem SQL para criar a tabela fornecedor. O comando SQL que cria a tabela fornecedor está no código a seguir e foi guardado em uma variável chamada *ddl\_create*.

**Observação: se tentar criar uma tabela que já existe, um erro é retornado.**

Caso execute todas as células novamente, certifique-se de apagar a tabela no banco, para evitar o erro.

In [2]:

```
ddl_create = """
CREATE TABLE fornecedor (
    id_fornecedor INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
    nome_fornecedor TEXT NOT NULL,
    cnpj VARCHAR(18) NOT NULL,
    cidade TEXT,
    estado VARCHAR(2) NOT NULL,
    cep VARCHAR(9) NOT NULL,
    dataCadastro DATE NOT NULL
);"""
"""
```

O comando DDL implementado na entrada 2 faz parte do conjunto de instruções SQL, razão pela qual deve seguir a sintaxe que essa linguagem determina.

**SAIBA MAIS**

Caso queira saber mais sobre o comando CREATE, recomendamos a leitura das páginas 49 a 52 da seguinte obra, disponível na biblioteca virtual.  
RAMAKRISHNAN, R.; GEHRKE, J. **Sistemas de gerenciamento de banco de dados**. 3. ed. Porto Alegre : AMGH, 2003.

0

Ver anotações

Podemos notar o padrão na instrução, que começa com o comando CREATE TABLE, seguido do nome da tabela a ser criada e, entre parênteses, o nome do campo, o tipo e a especificação de quando não são aceitos valores nulos. O primeiro campo possui uma instrução adicional, que é o autoincremento, ou seja, para cada novo registro inserido, o valor desse campo aumentará um.

Já temos a conexão e a DDL. Agora basta utilizar um mecanismo para que esse comando seja executado no banco. Esse mecanismo, segundo o PEP 249, deve estar implementado em um método chamado `execute()` de um objeto **cursor**. Os cursores desempenham o papel de pontes entre os conjuntos fornecidos como respostas das consultas e as linguagens de programação que não suportam conjuntos (RAMAKRISHNAN; GEHRKE, 2003). Portanto, sempre que precisarmos executar um comando SQL no banco usando a linguagem Python, usaremos um cursor para construir essa ponte. Observe o código a seguir.

In [3]:

```
cursor = conn.cursor()
cursor.execute(ddl_create)
print(type(cursor))

print("Tabela criada!")
print("Descrição do cursor: ", cursor.description)
print("Linhas afetadas: ", cursor.rowcount)
cursor.close()
conn.close()

<class 'sqlite3.Cursor'>
Tabela criada!
Descrição do cursor: None
Linhas afetadas: -1
```

0

Ver anotações

Na linha 1 da entrada 3, a partir da conexão, criamos um objeto cursor. Na linha 2, invocamos o método `execute()` desse objeto para, enfim, criar a tabela pelo comando armazenado na variável `ddl_create`. Como o cursor é uma classe, ele possui métodos e atributos. Nas linhas 6 e 7 estamos acessando os atributos `description` e `rowcount`. O primeiro diz respeito a informações sobre a execução; e o segundo a quantas linhas foram afetadas. No módulo `sqlite3`, o atributo `description` fornece os nomes das colunas da última consulta. Como se trata de uma instrução DDL, a `description` retornou `None` e a quantidade de linhas afetadas foi `-1`. Todo cursor e toda conexão, após executarem suas tarefas, devem ser fechados pelo método `close()`.

Segundo o PEP 249 (2020), todos os módulos devem implementar 7 campos para o resultado do atributo `description`: `name`, `type_code`, `display_size`, `internal_size`, `precision`, `scale` e `null_ok` (<https://www.python.org/dev/peps/pep-0249/#description>).

Além de criar uma tabela, também podemos excluí-la. A sintaxe para apagar uma tabela (e todos seus dados) é "DROP TABLE `table_name`".

## CRUD - CREATE, READ, UPDATE, DELETE

CRUD é um acrônimo para as quatro operações de DML que podemos fazer em uma tabela no banco de dados. Podemos inserir informações (create), ler (read), atualizar (update) e apagar (delete). Os passos necessários para efetuar uma das operações do CRUD são sempre os mesmos: (i) estabelecer a conexão com um banco; (ii) criar um cursor e executar o comando; (iii) gravar a operação; (iv) fecha o cursor e a conexão.

0

Ver anotações

### CREATE

Vamos começar inserindo registros na tabela fornecedor. Observe o código a seguir.

In [4]:

```
# Só é preciso importar a biblioteca uma vez. Importamos novamente para manter
todo o código em uma única célula
import sqlite3

conn = sqlite3.connect('aulaDB.db')
cursor = conn.cursor()

cursor.execute("""
INSERT INTO fornecedor (nome_fornecedor, cnpj, cidade, estado, cep,
dataCadastro)
VALUES ('Empresa A', '11.111.111/1111-11', 'São Paulo', 'SP', '11111-111',
'2020-01-01')
""")

cursor.execute("""
INSERT INTO fornecedor (nome_fornecedor, cnpj, cidade, estado, cep,
dataCadastro)
VALUES ('Empresa B', '22.222.222/2222-22', 'Rio de Janeiro', 'RJ', '22222-222',
'2020-01-01')
""")

cursor.execute("""
INSERT INTO fornecedor (nome_fornecedor, cnpj, cidade, estado, cep,
dataCadastro)
VALUES ('Empresa C', '33.333.333/3333-33', 'Curitiba', 'PR', '33333-333', '2020-
01-01')
""")

conn.commit()

print("Dados inseridos!")
print("Descrição do cursor: ", cursor.description)
print("Linhas afetadas: ", cursor.rowcount)
```

0

Ver anotações

```
cursor.close()
```

```
conn.close()
```

Dados inseridos!

Descrição do cursor: None

Linhas afetadas: 1

0

Ver anotações

Na entrada 4, fizemos a conexão e criamos um cursor (linhas 4 e 5). Através do cursor, inserimos 3 registros na tabela fornecedor. A sintaxe para a inserção exige que se passe os campos a serem inseridos e os valores. Veja que não passamos o campo *id\_fornecedor*, pois este foi criado como autoincremento. Após a execução das três inserções, na linha 22, usamos o método `commit()` para gravar as alterações na tabela. Veja que a quantidade de linhas afetadas foi 1, pois mostra o resultado da última execução do cursor, que foi a inserção de 1 registro.

Uma maneira mais prática de inserir vários registros é passar uma lista de tuplas, na qual cada uma destas contém os dados a serem inseridos em uma linha. Nesse caso, teremos que usar o método `executemany()` do cursor. Observe o código a seguir.

In [5]:

```
# Só é preciso importar a biblioteca uma vez. Importamos novamente para manter
todo o código em uma única célula
import sqlite3

0

conn = sqlite3.connect('aulaDB.db')
cursor = conn.cursor()

dados = [
    ('Empresa D', '44.444.444/4444-44', 'São Paulo', 'SP', '44444-444', '2020-
01-01'),
    ('Empresa E', '55.555.555/5555-55', 'São Paulo', 'SP', '55555-555', '2020-
01-01'),
    ('Empresa F', '66.666.666/6666-66', 'São Paulo', 'SP', '66666-666', '2020-
01-01')
]

cursor.executemany("""
INSERT INTO fornecedor (nome_fornecedor, cnpj, cidade, estado, cep,
dataCadastro)
VALUES (?, ?, ?, ?, ?, ?)""", dados)

conn.commit()

print("Dados inseridos!")
print("Descrição do cursor: ", cursor.description)
print("Linhas afetadas: ", cursor.rowcount)
cursor.close()
conn.close()
```

Dados inseridos!

Descrição do cursor: None

Linhas afetadas: 3

0

Ver anotações

Na entrada 5, criamos uma lista de tuplas chamada *dados*. Na linha 13 invocamos o método *executemany()* para inserir a lista. Veja que agora os valores foram substituídos por interrogações, e, além da instrução SQL, o método exige a passagem dos dados. Veja que agora foram afetadas 3 linhas no banco, pois esse foi o resultado do método do cursor.

Ver anotações

## READ

Agora que temos dados na tabela fornecedor, podemos avançar para a segunda operação, que é recuperar os dados. Também precisamos estabelecer uma conexão e criar um objeto cursor para executar a instrução de seleção. Ao executar a seleção, podemos usar o método *fetchall()* para capturar todas as linhas, através de uma lista de tuplas. Observe o código a seguir.

In [6]:

```
# Só é preciso importar a biblioteca uma vez. Importamos novamente para manter
# todo o código em uma única célula
import sqlite3

conn = sqlite3.connect('aulaDB.db')
cursor = conn.cursor()

cursor.execute("SELECT * FROM fornecedor")
resultado = cursor.fetchall()

print("Descrição do cursor: ", cursor.description)
print("Linhas afetadas: ", cursor.rowcount)

resultado[:2]
```

Descrição do cursor: (('id\_fornecedor', None, None, None, None, None, None), ('nome\_fornecedor', None, None, None, None, None, None), ('cnpj', None, None, None, None, None, None), ('cidade', None, None, None, None, None, None), ('estado', None, None, None, None, None, None), ('cep', None, None, None, None, None, None), ('data\_cadastro', None, None, None, None, None, None))

Linhas afetadas: -1

Out[6]:

```
[ (1,
    'Empresa A',
    '11.111.111/1111-11',
    'São Paulo',
    'SP',
    '11111-111',
    '2020-01-01'),
  (2,
    'Empresa B',
    '22.222.222/2222-22',
    'Rio de Janeiro',
    'RJ',
    '22222-222',
    '2020-01-01)]
```

Ver anotações

In [7]:

```
for linha in resultado:
    print(linha)
```

```
(1, 'Empresa A', '11.111.111/1111-11', 'São Paulo', 'SP', '11111-111', '2020-01-01')
(2, 'Empresa B', '22.222.222/2222-22', 'Rio de Janeiro', 'RJ', '22222-222', '2020-01-01')
(3, 'Empresa C', '33.333.333/3333-33', 'Curitiba', 'PR', '33333-333', '2020-01-01')
(4, 'Empresa D', '44.444.444/4444-44', 'São Paulo', 'SP', '44444-444', '2020-01-01')
(5, 'Empresa E', '55.555.555/5555-55', 'São Paulo', 'SP', '55555-555', '2020-01-01')
(6, 'Empresa F', '66.666.666/6666-66', 'São Paulo', 'SP', '66666-666', '2020-01-01')
```

0

Ver anotações

Na entrada 6, usamos a instrução SQL "select \* from fornecedor" para selecionar todos (\*) os dados da tabela fornecedor. O comando é executado pelo cursor e, através do método *fetchall()*, guardamos o resultado na variável "resultado". O resultado do método é uma lista de tuplas. Para ficar claro, na linha 13 imprimimos uma fatia da lista. Outro detalhe interessante é o resultado do atributo *description*, que retornou tuplas, informando o nome da coluna afetada. Os outros 6 campos da tupla retornaram *None* graças à implementação do módulo *sqlite3* (PSF, 2020d).

Na entrada 7, usamos uma estrutura de decisão para iterar na lista e imprimir cada valor. Veja que cada linha é uma tupla com as informações que inserimos.

A linguagem SQL é muito poderosa para manipulação de dados. Podemos selecionar somente os registros que satisfaçam uma determinada condição usando a cláusula "where", que funciona como uma estrutura condicional. Observe o código a seguir, no qual selecionamos somente o registro cujo id\_fornecedor é igual a 5.

In [8]:

```
cursor.execute("SELECT * FROM fornecedor WHERE id_fornecedor = 5")  
resultado = cursor.fetchall()  
print(resultado)  
  
cursor.close()  
conn.close()  
[(5, 'Empresa E', '55.555.555/5555-55', 'São Paulo', 'SP', '55555-555', '2020-01-01')]
```

0

Ver anotações

## ■ UPDATE

Ao inserir um registro no banco, pode ser necessário alterar o valor de uma coluna, o que pode ser feito por meio da instrução SQL UPDATE. Observe o código a seguir.

In [9]:

```
# Só é preciso importar a biblioteca uma vez. Importamos novamente para manter  
todo o código em uma única célula  
  
import sqlite3  
  
conn = sqlite3.connect('aulaDB.db')  
cursor = conn.cursor()  
  
cursor.execute("UPDATE fornecedor SET cidade = 'Campinas' WHERE id_fornecedor =  
5")  
conn.commit()  
  
cursor.execute("SELECT * FROM fornecedor")  
for linha in cursor.fetchall():  
    print(linha)  
  
cursor.close()  
conn.close()
```

```
(1, 'Empresa A', '11.111.111/1111-11', 'São Paulo', 'SP', '11111-111', '2020-01-01')
(2, 'Empresa B', '22.222.222/2222-22', 'Rio de Janeiro', 'RJ', '22222-222', '2020-01-01')
(3, 'Empresa C', '33.333.333/3333-33', 'Curitiba', 'PR', '33333-333', '2020-01-01')
(4, 'Empresa D', '44.444.444/4444-44', 'São Paulo', 'SP', '44444-444', '2020-01-01')
(5, 'Empresa E', '55.555.555/5555-55', 'Campinas', 'SP', '55555-555', '2020-01-01')
(6, 'Empresa F', '66.666.666/6666-66', 'São Paulo', 'SP', '66666-666', '2020-01-01')
```

0

Ver anotações

Na entrada 9, alteramos o campo *cidade* do registro com id\_fornecedor 5. No comando *update* é necessário usar a cláusula *where* para identificar o registro a ser alterado, caso não use, todos são alterados. Como estamos fazendo uma alteração no banco, precisamos gravar, razão pela qual usamos o *commit()* na linha 8. Para checar a atualização fizemos uma leitura mostrando todos os registros.

## | DELETE

Ao inserir um registro no banco, pode ser necessário removê-lo no futuro, o que pode ser feito por meio da instrução SQL DELETE. Observe o código a seguir.

In [10]:

```
# Só é preciso importar a biblioteca uma vez. Importamos novamente para manter
todo o código em uma única célula
import sqlite3

conn = sqlite3.connect('aulaDB.db')
cursor = conn.cursor()

cursor.execute("DELETE FROM fornecedor WHERE id_fornecedor = 2")
conn.commit()

cursor.execute("SELECT * FROM fornecedor")
for linha in cursor.fetchall():
    print(linha)

cursor.close()
conn.close()

(1, 'Empresa A', '11.111.111/1111-11', 'São Paulo', 'SP', '11111-111', '2020-01-
01')
(3, 'Empresa C', '33.333.333/3333-33', 'Curitiba', 'PR', '33333-333', '2020-01-
01')
(4, 'Empresa D', '44.444.444/4444-44', 'São Paulo', 'SP', '44444-444', '2020-01-
01')
(5, 'Empresa E', '55.555.555/5555-55', 'Campinas', 'SP', '55555-555', '2020-01-
01')
(6, 'Empresa F', '66.666.666/6666-66', 'São Paulo', 'SP', '66666-666', '2020-01-
01')
```

0

Ver anotações

Na entrada 10, apagamos o registro com `id_fornecedor` 2. No comando `delete`, é necessário usar a cláusula `where` para identificar o registro apagado. Como estamos fazendo uma alteração no banco, precisamos gravar, razão pela qual usamos o `commit()` na linha 8. Para checar a atualização, fizemos uma leitura que mostra todos os registros.

Com a operação *delete*, concluímos nosso CRUD em um banco de dados SQLite usando a linguagem Python. O mais interessante e importante é que todas as etapas e todos os comandos que usamos podem ser aplicados em qualquer banco de dados relacional, uma vez que os módulos devem seguir as mesmas regras.

o

Ver anotações

## INFORMAÇÕES DO BANCO DE DADOS E DAS TABELAS

Além das operações de CRUD, é importante sabermos extrair informações estruturais do banco de dados e das tabelas. Por exemplo, considerado um banco de dados, quais tabelas existem ali? Quais são os campos de uma tabela? Qual é a estrutura da tabela, ou seja, qual DDL foi usada para gerá-la? Os comandos necessários para extraí-las podem mudar entre os bancos, mas vamos ver como extraí-las do SQLite. No código a seguir (entrada 11), temos uma instrução SQL capaz de retornar as tabelas no banco SQLite (linha 8) e outra capaz de extraí-las das DDLs usadas para gerar as tabelas (linha 15).

In [11]:

```
# Só é preciso importar a biblioteca uma vez. Importamos novamente para manter
todo o código em uma única célula
import sqlite3

conn = sqlite3.connect('aulaDB.db')
cursor = conn.cursor()

# Lista as tabelas do banco de dados
cursor.execute("""SELECT name FROM sqlite_master WHERE type='table' ORDER BY
name""")
print('Tabelas:')
for tabela in cursor.fetchall():
    print(tabela)

# Captura a DDL usada para criar a tabela
tabela = 'fornecedor'
cursor.execute(f"""SELECT sql FROM sqlite_master WHERE type='table' AND
name='{tabela}'""")
print(f'\nDDL da tabela {tabela}:')
for schema in cursor.fetchall():
    print("%s" % (schema))

cursor.close()
conn.close()
```

0

Ver anotações

## Tabelas:

```
('fornecedor',)  
('sqlite sequence',)
```

DDL da tabela fornecedor:

```
CREATE TABLE fornecedor (
    id_fornecedor INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
    nome_fornecedor TEXT NOT NULL,
    cnpj VARCHAR(18) NOT NULL,
    cidade TEXT,
    estado VARCHAR(2) NOT NULL,
    cep VARCHAR(9) NOT NULL,
    data_cadastro DATE NOT NULL
)
```

8

[Ver anotações](#)

Que tal usar o ambiente a seguir para testar a conexão e as operações com o banco SQLite?! Execute o código, altere e faça novos testes!



## REFERÊNCIAS E LINKS ÚTEIS

BANIN, S. L. **Python 3 - conceitos e aplicações**: uma abordagem didática. São Paulo: Érica, 2018.

MACHADO, F. N. R. **Projeto e implementação de banco de dados**. 3. ed. São Paulo: Érica, 2014.

MENON, R. M. Introduction to JDBC. In: MENON, R. M. **Expert Oracle JDBC Programming**. New York: Apress, 2005. p. 79-113.

MICROSOFT. **O que é o ODBC?** 2017. Disponível em: <https://bit.ly/2XRHtuw>. Acesso em: 31 jul. 2020.

PEP 249 - Python database API specification v.2.0. **Python**, 2020. Disponível em: <https://bit.ly/3izcprq>. Acesso em: 31 jul. 2020.

PSF - Python Software Fundation. **Python Module Index**. 2020c. Disponível em: <https://bit.ly/3fTMkkY>. Acesso em: 04 jun. 2020

PSF - Python Software Fundation. **sqlite3**. 2020d. Disponível em: <https://bit.ly/3iE8ARY>. Acesso em: 4 jun. 2020.

RAMAKRISHNAN, R.; GEHRKE, J. **Sistemas de gerenciamento de banco de dados**. 3. ed. Porto Alegre: AMGH, 2003.

ROCKOFF, L. **The Language of SQL**. 2. ed. [S.l.]: Pearson Education, 2016.

SQLITE. **What Is SQLite?** 2020. Disponível em: <https://www.sqlite.org/index.html>. Acesso em: 12 jun. 2020.

TAKAHASHI, M.; AZUMA, S. **Guia mangá de bancos de dados**. São Paulo: Novatec, 2009.

VAISH, G. **Getting Started with NoSQL**. Birmingham: Packt Publishing, 2013.

## FOCO NO MERCADO DE TRABALHO

# APLICAÇÃO DE BANCO DE DADOS COM PYTHON

Vanessa Cadan Scheffer

0

Ver anotações

## SOLUÇÃO CRUD

CRUD é um acrônimo para as quatro operações de DML que podemos fazer em uma tabela no banco de dados, como inserir informações (CREATE), ler (READ), atualizar (UPDATE) e apagar (DELETE).



Fonte: Shutterstock.

## Deseja ouvir este material?

Áudio disponível no material digital.

## DESAFIO

Utilizar bancos de dados para persisti-los faz parte da realidade de toda empresa que tenha algum sistema computadorizado, mesmo o destinado somente para cadastro de cliente. Na edição do dia 6 de março de 2017, a revista *The Economist* publicou um artigo cujo título afirma "The world's most valuable resource is no longer oil, but data" (O recurso mais valioso do mundo não é mais petróleo, mas dados) (<https://econ.st/3goh4Mj>). Esse texto trouxe os holofotes para o que muitas empresas já sabiam: uma vez que os dados são preciosos, precisamos armazená-los e tratá-los para extrair informações.

Hoje é comum encontrar nas empresas equipes totalmente focadas em soluções que envolvem dados. Parte do time é responsável por construir componentes (softwares ou microsserviços) voltados às necessidades de armazenamento e recuperação de dados. Esses componentes precisam ser parametrizáveis, o que significa que o componente deve ser capaz de fazer a mesma tarefa para diferentes tecnologias. Para os casos de recuperação de dados em banco de dados relacional, construir um componente parametrizável significa recuperar dados de diferentes RDBMS com o mesmo código mas com diferentes parâmetros.

Como desenvolvedor em uma empresa de consultoria de software, você foi alocado para construir uma solução parametrizável capaz de criar banco de dados na tecnologia SQLite, criar e apagar tabelas, o nome do banco, das tabelas e a DDL necessária para criar uma tabela (tudo deve ser parametrizável). Você também deve construir uma solução capaz de fazer um CRUD em um banco SQLite – veja que o componente deve funcionar para qualquer nome de banco e de tabela, ou seja, parâmetros. As regras que lhe foram passadas são as seguintes:

- Para criar uma nova base de dados, é necessário informar o nome.
- Para criar uma nova tabela, é necessário informar o nome do banco que receberá a tabela e a DDL de criação.
- Para excluir uma tabela, é necessário informar o nome do banco e da tabela.

Ver anotações

- Para inserir um novo registro em uma tabela, é preciso informar: o nome do banco e da tabela e um dicionário contendo a chave e o valor a ser inserido. Por exemplo, {'nome': 'João', 'idade': 30}.
- Para recuperar, os dados é preciso informar o nome do banco e da tabela.
- Para atualizar um novo registro em uma tabela é preciso informar: o nome do banco e da tabela e dois dicionários, um contendo a chave e o valor a ser atualizado e outro contendo a chave e o valor da condição do registro que será atualizado.
- Para excluir um registro em uma tabela, é preciso informar: o nome do banco e da tabela e um dicionário contendo a chave e o valor da condição para localizar o registro a ser inserido. Por exemplo, {'id\_cliente': 10}.

Então, mãos à obra!

## RESOLUÇÃO

Chegou o momento de implementar a versão inicial da solução parametrizável que realiza funções tanto de DDL quanto de DML em um banco de dados SQLite. Certamente, existem muitas possibilidades de implementação. Vejamos uma proposta inicial de solução.

Uma possibilidade é construir duas classes, uma com os métodos para DDL e outra para o CRUD (DML). Vamos, então, começar pela classe capaz de criar um banco, criar e apagar uma tabela. Veja a seguir a classe *DDLSQLite*. Criamos um método privado (lembre-se de que, em Python, trata-se somente uma convenção de nomenclatura) que retorna uma instância de conexão com um banco parametrizável. Também criamos três métodos públicos, um para criar um banco de dados, outro para criar uma tabela e o último para apagar uma tabela. Todos os métodos são parametrizáveis, conforme foi solicitado. Chamo a atenção para o

método *criar\_banco\_de\_dados()*, que cria o banco fazendo uma conexão e já fechando-a (lembre-se de que existem diferentes formas de fazer uma determinada implementação).

0

In [12]:

Ver anotações

```
import sqlite3

class DDLSQLite:

    def _conectar(self, nome_banco):
        nome_banco += '.db'
        conn = sqlite3.connect(nome_banco)
        return conn

    def criar_banco_de_dados(self, nome_banco):
        nome_banco += '.db'
        sqlite3.connect(nome_banco).close()
        print(f'O banco de dados {nome_banco} foi criado com sucesso!')
        return None

    def criar_tabela(self, nome_banco, ddl_create):
        conn = self._conectar(nome_banco)
        cursor = conn.cursor()
        cursor.execute(ddl_create)
        cursor.close()
        conn.close()
        print(f'Tabela criada com sucesso!')
        return None

    def apagar_tabela(self, nome_banco, tabela):
        conn = self._conectar(nome_banco)
        cursor = conn.cursor()
        cursor.execute(f"DROP TABLE {tabela}")
        cursor.close()
        conn.close()
        print(f'A tabela {tabela} foi excluída com sucesso!')
        return None
```

0

Ver anotações

O próximo passo é construir a classe que faz o CRUD. Observe a proposta da classe *CrudSQLite*. Como todas as operações precisam receber como parâmetro o nome do banco de dados, colocamos o parâmetro no método construtor, ou seja, para instanciar essa classe, deve ser informado o nome da banco e, a partir do objeto, basta chamar os métodos e passar os demais parâmetros. O nome do banco será usado pelo método privado *\_conectar()*, que retorna uma instância da conexão – esse método será chamado internamente pelos demais métodos. O método *inserir\_registro()*, extrai como uma tupla as colunas (linha 13) e os valores (linha 14) a serem usados para construir o comando de inserção. O método *ler\_registros()* seleciona todos os dados de uma tabela e os retorna, lembrando que o retorno será uma lista de tuplas. O método *atualizar\_registro()*, precisa tanto dos dados a serem atualizados (dicionário) quanto dos dados que serão usados para construir a condição. **Nessa primeira versão do componente, a atualização só pode ser feita desde que o valor da condição seja inteiro.** O método *apagar\_registro()* também só pode ser executado desde que o valor da condição seja inteiro.

Ver anotações

In [13]:

```
import sqlite3

class CrudSQLite:

    def __init__(self, nome_banco):
        self.nome_banco = nome_banco + '.db'

    def _conectar(self):
        conn = sqlite3.connect(self.nome_banco)
        return conn

    def inserir_registro(self, tabela, registro):
        colunas = tuple(registro.keys())
        valores = tuple(registro.values())

        conn = self._conectar()
        cursor = conn.cursor()
        query = f"""INSERT INTO {tabela} {colunas} VALUES {valores}"""
        cursor.execute(query)
        conn.commit()
        cursor.close()
        conn.close()
        print("Dados inseridos com sucesso!")
        return None

    def ler_registros(self, tabela):
        conn = self._conectar()
        cursor = conn.cursor()
        query = f"""SELECT * FROM {tabela}"""
        cursor.execute(query)
        resultado = cursor.fetchall()
        cursor.close()
        conn.close()
        return resultado
```

0

Ver anotações

```
def atualizar_registro(self, tabela, dado, condicao):
    campo_alterar = list(dado.keys())[0]
    valor_alterar = dado.get(campo_alterar)
    campo_condicao = list(condicao.keys())[0]
    valor_condicao = condicao.get(campo_condicao)

    conn = self._conectar()
    cursor = conn.cursor()
    query = f"""UPDATE {tabela} SET {campo_alterar} = '{valor_alterar}' WHERE {campo_condicao} = {valor_condicao}"""
    cursor.execute(query)
    conn.commit()
    cursor.close()
    conn.close()
    print("Dado atualizado com sucesso!")
    return None

def apagar_registro(self, tabela, condicao):
    campo_condicao = list(condicao.keys())[0]
    valor_condicao = condicao.get(campo_condicao)
    conn = self._conectar()
    cursor = conn.cursor()
    query = f"""DELETE FROM {tabela} WHERE {campo_condicao} = {valor_condicao}"""
    cursor.execute(query)
    conn.commit()
    cursor.close()
    conn.close()
    print("Dado excluído com sucesso!")
    return None
```

Antes de entregar uma solução, é preciso testar muitas vezes. Portanto, vamos aos testes. Primeiro vamos testar a classe DDLSQLite. Como podemos observar no resultado, tudo ocorreu como esperado: tanto o método para criar o banco quanto

Ver anotações

a tabela funcionaram corretamente.

In [14]:

```
# instancia um objeto
objeto_ddl = DDLSQLite()

# Cria um banco de dados
objeto_ddl.criar_banco_de_dados('desafio')

# Cria uma tabela chamada cliente
ddl_create = """
CREATE TABLE cliente (
    id_cliente INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
    nome_cliente TEXT NOT NULL,
    cpf VARCHAR(14) NOT NULL,
    email TEXT NOT NULL,
    telefone VARCHAR(15),
    cidade TEXT,
    estado VARCHAR(2) NOT NULL,
    data_cadastro DATE NOT NULL
);
"""

objeto_ddl.criar_tabela(nome_banco='desafio', ddl_create=ddl_create)

# Caso precise excluir a tabela, o comando a seguir deverá ser usado
# objeto_ddl.apagar_tabela(nome_banco='desafio', tabela='cliente')
```

O banco de dados desafio.db foi criado com sucesso!

Tabela criada com sucesso!

Vamos fazer os testes do CRUD em duas etapas. Na primeira, vamos inserir registros e consultarmos para checar se a inserção deu certo.

0

Ver anotações

In [15]:

```
objeto_dml = CrudSQLite(nome_banco='desafio')

# Inserir registros

dados = [
    {
        'nome_cliente': 'João',
        'cpf': '111.111.111-11',
        'email': 'joao@servidor',
        'cidade': 'São Paulo',
        'estado': 'SP',
        'data_cadastro': '2020-01-01'
    },
    {
        'nome_cliente': 'Maria',
        'cpf': '222.222.222-22',
        'email': 'maria@servidor',
        'cidade': 'São Paulo',
        'estado': 'SP',
        'data_cadastro': '2020-01-01'
    },
]
# Para cada dicionário na lista de dados, invoca o método de inserção
for valor in dados:
    objeto_dml.inserir_registro(tabela='cliente', registro=valor)

# Carrega dados salvos
dados_carregados = objeto_dml.ler_registros(tabela='cliente')
for dado in dados_carregados:
    print(dado)
```

0

Ver anotações

Dados inseridos com sucesso!

Dados inseridos com sucesso!

(1, 'João', '111.111.111-11', 'joao@servidor', None, 'São Paulo', 'SP', '2020-01-01')

(2, 'Maria', '222.222.222-22', 'maria@servidor', None, 'São Paulo', 'SP', '2020-01-01')

Agora vamos atualizar um registro e excluir outro.

In [16]:

```
# Atualiza registro
dado_atualizar = {'telefone': '(11)1.1111-1111'}
condicao = {'id_cliente': 1}

objeto_dml.atualizar_registro(tabela='cliente', dado=dado_atualizar,
condicao=condicao)

dados_carregados = objeto_dml.ler_registros(tabela='cliente')
for dado in dados_carregados:
    print(dado)

# Apaga registro
condicao = {'id_cliente': 1}

objeto_dml.apagar_registro(tabela='cliente', condicao=condicao)
```

Dado atualizado com sucesso!

(1, 'João', '111.111.111-11', 'joao@servidor', '(11)1.1111-1111', 'São Paulo', 'SP', '2020-01-01')

(2, 'Maria', '222.222.222-22', 'maria@servidor', None, 'São Paulo', 'SP', '2020-01-01')

Dado excluído com sucesso!

0

Ver anotações

Felizmente nosso componente funcionou conforme esperado. Gostaríamos de encerrar o desafio lembrando que são muitas as formas de implementação. Tratando-se de classes que se comunicam com banco de dados, o padrão de projeto singleton é o mais adequado para determinados casos, uma vez que permite instanciar somente uma conexão com o banco.

0

Ver anotações

## DESAFIO DA INTERNET

Ganhar habilidade em programação exige estudo e treino (muito treino). Acesse a biblioteca virtual no endereço: (<http://biblioteca-virtual.com/>), e busque pelo livro a seguir referenciado. Nas páginas 202 e 203 da obra, você encontra o exemplo 8.5, que faz a atualização de dados a partir de um arquivo no disco. Que tal implementar essa solução?

NÃO PODE FALTAR

# INTRODUÇÃO A BIBLIOTECA PANDAS

Vanessa Cadan Scheffer

0

Ver anotações

## ESTRUTURA DE DADOS

Pandas é um pacote Python que fornece estruturas de dados projetadas para facilitar o trabalho com dados estruturados (tabelas) e de séries temporais.



Fonte: Shutterstock.

## Deseja ouvir este material?

Áudio disponível no material digital.

## INTRODUÇÃO A BIBLIOTECA PANDAS

Dentre as diversas bibliotecas disponíveis no repositório PyPI, pandas é um pacote Python que fornece estruturas de dados projetadas para facilitar o trabalho com dados estruturados (tabelas) e de séries temporais

([https://pandas.pydata.org/docs/getting\\_started/overview.html](https://pandas.pydata.org/docs/getting_started/overview.html)). Esse pacote começou a ser desenvolvido em 2008, tornando-se uma solução open source no final de 2009. Desde 2015, o projeto pandas é patrocinado pela NumFOCUS (<https://pandas.pydata.org/about/>). Segundo sua documentação oficial, pandas pretende ser o alicerce de alto nível fundamental para uma análise prática, a dos dados do mundo real em Python. Além disso, ele tem o objetivo mais amplo de se tornar a ferramenta de análise/manipulação de dados de código aberto, mais poderosa e flexível disponível em qualquer linguagem de programação.

Para utilizar a biblioteca pandas é preciso fazer a instalação, como mostra a Figura 4.1: `pip install pandas` (caso esteja usando conda, verificar documentação). No momento em que esse material está sendo produzido, a biblioteca encontra-se na versão 1.0.4 e teve sua última atualização disponibilizada no dia 29 de maio de 2020 (Figura 4.1).

Ver anotações

Figura 4.1 - Instalação do pandas

 A figura ilustra a página [pypi.org](https://pypi.org/project/pandas/) com o botão para a instalação do pandas.

Fonte: Pypi. Disponível em: <https://pypi.org/project/pandas/>.

Como uma ferramenta de alto nível, pandas possui duas estruturas de dados que são as principais para a análise/manipulação de dados: a Series e o DataFrame. Uma **Series** é um como um vetor de dados (unidimensional), capaz de armazenar diferentes tipos de dados. Um **DataFrame** é conjunto de Series, ou como a documentação apresenta, um contêiner para Series. Ambas estruturas, possuem como grande característica, a indexação das linhas, ou seja, cada linha possui um rótulo (nome) que o identifica, o qual pode ser uma string, uma inteiro, um decimal ou uma data. A Figura 4.2 ilustra uma Series (A) e um DataFrame (B). Veja que uma Series possui somente "uma coluna" de informação e seus rótulos (índices). Um DataFrame pode ter uma ou mais colunas e além dos índices, também há um

rótulo de identificação com o nome da coluna. Podemos comparar um DataFrame como uma planilha eletrônico, como o Excel (da Microsoft) ou o Calc (do Open Office).

0

Ver anotações

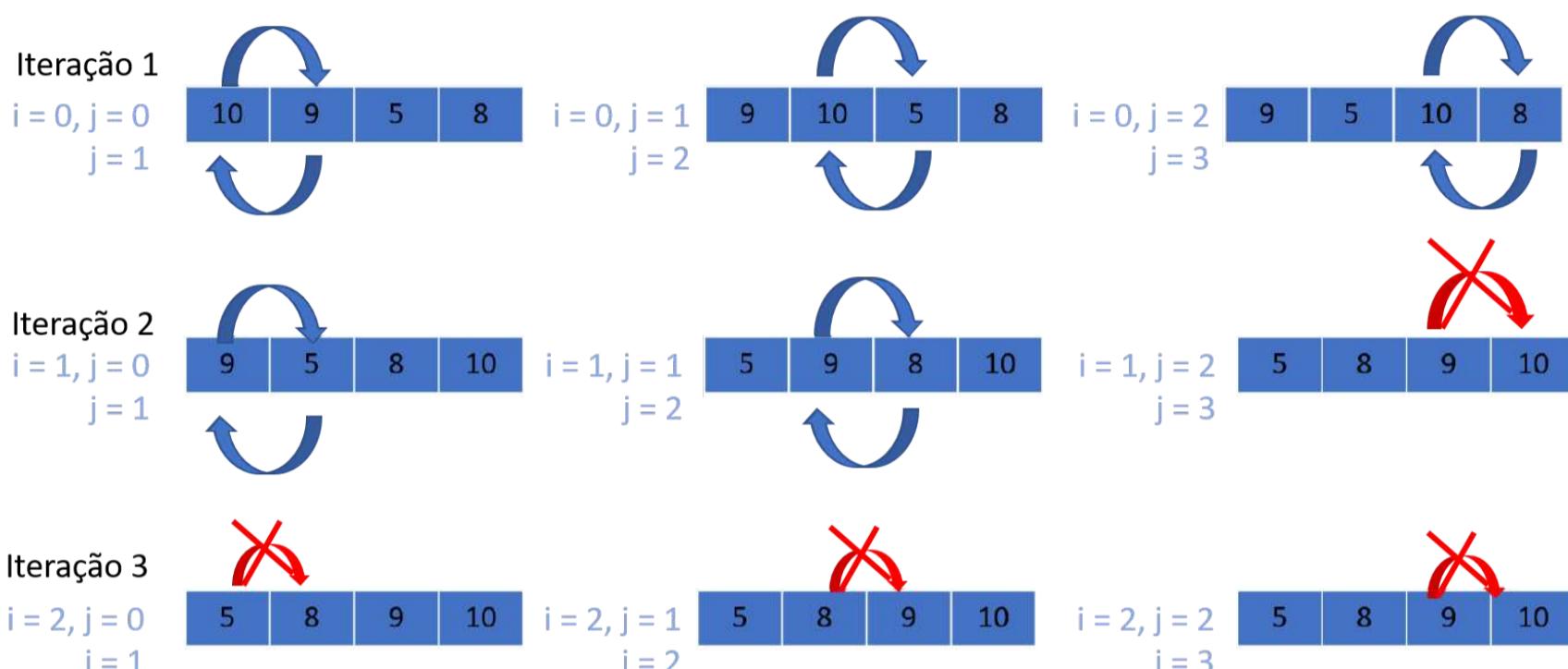
Figura 4.2 - Series (A) e DataFrame (B)

	Lista original			
Valor	10	9	5	8
Índice	0	1	2	3

i é o controle da repetição de todo o processo.

j é o controle interno, usado para a troca das posições.

Condição de parada: (não precisamos verificar a última posição i, j = n - 1



**Obs: Sempre volta no começo da iteração interna (j = 0)**

Fonte: elaborada pela autora.

Para finalizar nossa introdução ao paradigma OO, vamos falar do polimorfismo. Agora que já temos essa visão inicial, vamos começar a colocar a mão na massa! Começaremos aprendendo a criar Series e DataFrames, a partir de estruturas de dados em Python como listas e dicionários. Na próxima seção vamos aprender a trabalhar com fontes externas de dados. Após criarmos as estruturas do pandas, vamos ver como como extrair informações estatísticas básicas, bem como extrair informações gerais da estrutura e selecionar colunas específicas.

#### DICA

Na Internet você pode encontrar diversas "cheat sheet" (folha de dicas) sobre a biblioteca pandas. Recomendamos a cheat sheet oficial da biblioteca, disponível no endereço

[https://pandas.pydata.org/Pandas\\_Cheat\\_Sheet.pdf](https://pandas.pydata.org/Pandas_Cheat_Sheet.pdf)

Vamos importar a biblioteca antes de começar nossa primeira linha de código. Por convenção, a biblioteca é importada com o apelido (as) pd. Logo, para utilizar as funcionalidades, vamos utilizar a sintaxe pd.funcionalidade.

o

Ver anotações

In [1]:

```
import pandas as pd
```

## SERIES

Para construir um objeto do tipo Series, precisamos utilizar o método Series() do pacote pandas. O método possui o seguinte construtor: pandas.Series(data=None, index=None, dtype=None, name=None, copy=False, fastpath=False). Veja que todos os parâmetros possuem valores padrões (default) o que permite instanciar um objeto de diferentes formas. Para entender cada parâmetro, a melhor fonte de informações é a documentação oficial: <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.html>.

Dentre todos os parâmetros esperados, somente um é obrigatório para se criar uma Series com dados (se for uma Series sem dados, nenhum parâmetro é obrigatório), o parâmetro data=xxxx. Esse parâmetro pode receber, um simples valor (inteiro, string, float), uma lista de valores, ou um dicionário, vejamos os exemplos.

In [2]:

```
pd.Series(data=5) # Cria uma Series com o valor a
```

Out[2]:

```
0    5  
dtype: int64
```

In [3]:

```
lista_nomes = 'Howard Ian Peter Jonah Kellie'.split()
```

```
pd.Series(lista_nomes) # Cria uma Series com uma lista de nomes
```

Out[3]:

```
0    Howard  
1      Ian  
2    Peter  
3    Jonah  
4   Kellie  
dtype: object
```

In [4]:

```
dados = {  
    'nome1': 'Howard',  
    'nome2': 'Ian',  
    'nome3': 'Peter',  
    'nome4': 'Jonah',  
    'nome5': 'Kellie',  
}
```

```
pd.Series(dados) # Cria uma Series com um dicionário
```

Out[4]:

0

Ver anotações

```
nome1    Howard
nome2      Ian
nome3    Peter
nome4    Jonah
nome5   Kellie
dtype: object
```

0

Ver anotações

Na entrada 2, criamos uma Series com um único valor, veja que aparece 0 como índice e 5 como valor. Quando não deixamos explícito os rótulos (índices) que queremos usar é construído um range de 0 até N-1, onde N é a quantidade de valores. Outro detalhe interessante é o dtype (data type), que foi identificado como int64, nesse caso, já que temos somente um valor inteiro no objeto.

Na entrada 3, criamos uma Series a partir de uma lista de nomes, veja que agora os índices variam de 0 até 4 e o dtype é "object". Esse tipo de dado é usado para representar texto ou valores numéricos e não numéricos combinados.

Na entrada 4, criamos uma Series a partir de um dicionário, a grande diferença desse tipo de dado na construção é que a chave do dicionário é usada como índice.

Outra forma de construir a Series é passando os dados e os rótulos que desejamos usar. Veja na entrada 5 essa construção, na qual utilizamos uma lista de supostos cpfs para rotular os valores da Series.

In [5]:

```
cpfs = '111.111.111-11 222.222.222-22 333.333.333-33 444.444.444-44 555.555.555-
55'.split()

pd.Series(lista_nomes, index=cpfs)
```

Out[5]:

```
111.111.111-11      Howard  
222.222.222-22      Ian  
333.333.333-33      Peter  
444.444.444-44      Jonah  
555.555.555-55      Kellie  
dtype: object
```

0  
Ver anotações

Rotular as Series (e como veremos os DataFrames), é interessante para facilitar a localização e manipulação dos dados. Por exemplo, se quiséssemos saber o nome da pessoa com cpf 111.111.111-11, poderíamos localizar facilmente essa informação com o atributo **loc**, usando a seguinte sintaxe:

`series_dados.loc[rotulo]`, onde rótulo é índice a ser localizado. Veja o código a seguir na entrada 6, criamos uma Series com a lista de nomes e guardados dentro uma variável chamada `series_dados`. Na linha 3, com o atributo `loc`, localizamos a informação com índice '111.111.111-11'. Veremos mais sobre essa questão de filtrar informações, ao longo das aulas.

In [6]:

```
series_dados = pd.Series(lista_nomes, index=cpfs)  
  
series_dados.loc['111.111.111-11']
```

Out[6]:

```
'Howard'
```

## EXTRAINDO INFORMAÇÕES DE UMA SERIES

Já sabemos que estruturas de dados são utilizadas para armazenar dados e que, diferentes estruturas possuem diferentes atributos e métodos. Com as estruturas de dados do pandas não é diferente, tais objetos possuem atributos e métodos específicos, vamos conhecer alguns. Na entrada 7, criamos uma série contando números e um valor nulo (`None`). As informações extraídas das linhas 3 a 7, são mais com relação a "forma" dos dados. portanto poderiam ser usadas

independente do tipo de dado armazenado na Series, inclusive em um cenário de dados com diferentes tipos. Já as informações das linhas 9 a 15, como se tratam de funções matemáticas e estatísticas, podem fazer mais sentido quando utilizadas para tipos numéricos. Verifique no comentário a frente de cada comando, o que ele faz. Vale a pena ressaltar a diferença entre o atributo shape e o método count. O primeiro verifica quantas linhas a Series possui (quantos índices), já o segundo, conta quantos dados não nulos existem.

In [7]:

o

Ver anotações

```
series_dados = pd.Series([10.2, -1, None, 15, 23.4])

print('Quantidade de linhas = ', series_dados.shape) # Retorna uma tupla com o
número de linhas

print('Tipo de dados', series_dados.dtypes) # Retorna o tipo de dados, se for
misto será object

print('Os valores são únicos?', series_dados.is_unique) # Verifica se os valores
são únicos (sem duplicações)

print('Existem valores nulos?', series_dados.hasnans) # Verifica se existem
valores nulos

print('Quantos valores existem?', series_dados.count()) # Conta quantas valores
existem (exclui os nulos)

print('Qual o menor valor?', series_dados.min()) # Extrai o menor valor da
Series (nesse caso os dados precisam ser do mesmo tipo)

print('Qual o maior valor?', series_dados.max()) # Extrai o valor máximo, com a
mesma condição do mínimo

print('Qual a média aritmética?', series_dados.mean()) # Extrai a média
aritmética de uma Series numérica

print('Qual o desvio padrão?', series_dados.std()) # Extrai o desvio padrão de
uma Series numérica

print('Qual a mediana?', series_dados.median()) # Extrai a mediana de uma Series
numérica

print('\nResumo:\n', series_dados.describe()) # Exibe um resumo sobre os dados
na Series
```

0

Ver anotações

```
Quantidade de linhas = (5,)  
Tipo de dados float64  
Os valores são únicos? True  
Existem valores nulos? True  
Quantos valores existem? 4  
Qual o menor valor? -1.0  
Qual o maior valor? 23.4  
Qual a média aritmética? 11.899999999999999  
Qual o desvio padrão? 10.184301645179211  
Qual a mediana? 12.6
```

Resumo:

```
count      4.000000  
mean      11.900000  
std       10.184302  
min      -1.000000  
25%      7.400000  
50%      12.600000  
75%      17.100000  
max      23.400000  
dtype: float64
```

0

Ver anotações

## DATAFRAME

Para construir um objeto do tipo DataFrame, precisamos utilizar o método `DataFrame()` do pacote pandas. O método possui o seguinte construtor: `pandas.DataFrame(data=None, index=None, columns=None, dtype=None, copy=False)`. Veja que todos os parâmetros possuem valores padrões (default) o que permite instanciar um objeto de diferentes formas. Para entender cada parâmetro, a melhor fonte de informações é a documentação oficial:

<https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html>.

Dentre todos os parâmetros esperados, somente um é obrigatório para se criar um DataFrame com dados, o parâmetro `data=xxxx`. Esse parâmetro pode receber, um objeto iterável, como uma lista, tupla, um dicionário ou um DataFrame, vejamos os exemplos.

0

Ver anotações

## CONSTRUTOR DATAFRAME COM LISTA

Na entrada 8, criamos 4 listas, com mesmo tamanho (5 valores) que vamos usar como fonte de dados para criar os primeiros DataFrames. Na entrada 9, estamos invocando o método `DataFrame` e passando como parâmetro a lista de nomes e um nome (rótulo) para a coluna. Veja o resultado, temos os dados na coluna e os índices, que como não especificamos é atribuído o range de 0 a N-1. Na entrada 10, criamos o mesmo DataFrame, mas agora passando a lista de cpfs como índice. Na entrada 11, usamos a função `zip()` para criar tuplas, cada uma composta por um valor de cada lista, e a transformamos em uma lista de tuplas. Fizemos essa construção para criar um DataFrame, no qual cada lista passe a ser uma coluna, conforme pode ser observado no resultado.

In [2]:

```
lista_nomes = 'Howard Ian Peter Jonah Kellie'.split()
lista_cpfs = '111.111.111-11 222.222.222-22 333.333.333-33 444.444.444-44
555.555.555-55'.split()
lista_emails = 'risus.varius@dictumPhasellusin.ca Nunc@vulputate.ca
fames.ac.turpis@cursusa.org non@felisullamcorper.org
eget.dictum.placerat@necluctus.co.uk'.split()
lista_idades = [32, 22, 25, 29, 38]
```

In [9]:

```
pd.DataFrame(lista_nomes, columns=['nome'])
```

Out[9]:

	nome
0	Howard
1	Ian
2	Peter
3	Jonah
4	Kellie

0

Ver anotações

In [10]:

```
pd.DataFrame(lista_nomes, columns=[ 'nome' ], index=lista_cpfs)
```

Out[10]:

	nome
111.111.111-11	Howard
222.222.222-22	Ian
333.333.333-33	Peter
444.444.444-44	Jonah
555.555.555-55	Kellie

In [7]:

```
dados = list(zip(lista_nomes, lista_cpfs, lista_idades, lista_emails)) # cria
uma lista de tuplas

pd.DataFrame(dados, columns=[ 'nome', 'cpfs', 'idade', 'email'])
```

Out[7]:

	nome	cpfs	idade	email
0	Howard	111.111.111-11	32	risus.varius@dictumPhasellusin.ca
1	Ian	222.222.222-22	22	Nunc@vulputate.ca
2	Peter	333.333.333-33	25	fames.ac.turpis@cursusa.org
3	Jonah	444.444.444-44	29	non@felisullamcorper.org
4	Kellie	555.555.555-55	38	eget.dictum.placerat@necluctus.co.uk

## CONSTRUTOR DATAFRAME COM DICIONÁRIO

DataFrames também podem ser construídos a partir de estruturas de dados do tipo dicionário. Cada chave será uma coluna e pode ter atribuída uma lista de valores. **Obs: cada chave deve estar associada a uma lista de mesmo tamanho.**

Na entrada 12, criamos nosso dicionário de dados, veja que cada chave possui uma lista de mesmo tamanho e criamos nosso DataFrame, passando o dicionário como fonte de dados. Dessa forma o construtor já consegue identificar o nome das colunas.

Ver anotações

In [6]:

```
dados = {  
    'nomes': 'Howard Ian Peter Jonah Kellie'.split(),  
    'cpfs' : '111.111.111-11 222.222.222-22 333.333.333-33 444.444.444-44  
555.555.555-55'.split(),  
    'emails' : 'risus.varius@dictumPhasellusin.ca Nunc@vulputate.ca  
fames.ac.turpis@cursusa.org non@felisullamcorper.org  
eget.dictum.placerat@necluctus.co.uk'.split(),  
    'idades' : [32, 22, 25, 29, 38]  
}  
  
pd.DataFrame(dados)
```

Out[6]:

	<b>nomes</b>	<b>cpfs</b>	<b>emails</b>	<b>idades</b>
<b>0</b>	Howard	111.111.111-11	risus.varius@dictumPhasellusin.ca	32
<b>1</b>	Ian	222.222.222-22	Nunc@vulputate.ca	22
<b>2</b>	Peter	333.333.333-33	fames.ac.turpis@cursusa.org	25
<b>3</b>	Jonah	444.444.444-44	non@felisullamcorper.org	29
<b>4</b>	Kellie	555.555.555-55	eget.dictum.placerat@necluctus.co.uk	38

## EXTRAINDO INFORMAÇÕES DE UM DATAFRAME

Como já mencionamos, cada objeto possui seus próprios atributos e métodos, logo, embora Series e DataFrame tenham recursos em comum, eles também possuem suas particularidades. No DataFrame temos o método info() que mostra quantas linhas e colunas existem. Também exibe o tipo de cada coluna e quanto valores não nulos existem ali. Esse método também retorna uma informação sobre a quantidade de memória RAM essa estrutura está ocupando. Faça a leitura dos comentários e veja o que cada atributo e método retorna.

Ver anotações

In [13]:

```
df_dados = pd.DataFrame(dados)

print('\nInformações do DataFrame:\n')
print(df_dados.info()) # Apresenta informações sobre a estrutura do DF

print('\nQuantidade de linhas e colunas = ', df_dados.shape) # Retorna uma tupla com o número de linhas e colunas
print('\nTipo de dados:', df_dados.dtypes) # Retorna o tipo de dados, para cada coluna, se for misto será object

print('\nQual o menor valor de cada coluna?', df_dados.min()) # Extrai o menor de cada coluna
print('\nQual o maior valor?', df_dados.max()) # Extrai o valor máximo de cada coluna
print('\nQual a média aritmética?', df_dados.mean()) # Extrai a média aritmética de cada coluna numérica
print('\nQual o desvio padrão?', df_dados.std()) # Extrai o desvio padrão de cada coluna numérica
print('\nQual a mediana?', df_dados.median()) # Extrai a mediana de cada coluna numérica

print('\nResumo:', df_dados.describe()) # Exibe um resumo

df_dados.head() # Exibe os 5 primeiros registros do DataFrame
```

## Informações do DataFrame:

```
<class 'pandas.core.frame.DataFrame'>  
RangeIndex: 5 entries, 0 to 4  
Data columns (total 4 columns):  
nomes      5 non-null object  
cpfs       5 non-null object  
emails     5 non-null object  
idades     5 non-null int64  
dtypes: int64(1), object(3)  
memory usage: 240.0+ bytes  
None
```

Quantidade de linhas e colunas = (5, 4)

## Tipo de dados:

```
nomes      object  
cpfs       object  
emails    object  
idades    int64  
dtype: object
```

## Qual o menor valor de cada coluna?

```
nomes          Howard  
cpfs        111.111.111-11  
emails    Nunc@vulputate.ca  
idades         22  
dtype: object
```

## Qual o maior valor?

```
nomes          Peter  
cpfs        555.555.555-55  
emails  risus.varius@dictumPhasellusin.ca  
idades         38  
dtype: object
```

Ver anotações

Qual a média aritmética?

```
idades    29.2  
dtype: float64
```

Qual o desvio padrão?

```
idades    6.220932  
dtype: float64
```

Qual a mediana?

```
idades    29.0  
dtype: float64
```

Resumo:

```
          idades  
count    5.000000  
mean    29.200000  
std     6.220932  
min     22.000000  
25%    25.000000  
50%    29.000000  
75%    32.000000  
max    38.000000
```

Out[13]:

	<b>nomes</b>	<b>cpfs</b>	<b>emails</b>	<b>idades</b>
<b>0</b>	Howard	111.111.111-11	risus.varius@dictumPhasellusin.ca	32
<b>1</b>	Ian	222.222.222-22	Nunc@vulputate.ca	22
<b>2</b>	Peter	333.333.333-33	fames.ac.turpis@cursusa.org	25
<b>3</b>	Jonah	444.444.444-44	non@felisullamcorper.org	29
<b>4</b>	Kellie	555.555.555-55	eget.dictum.placerat@necluctus.co.uk	38

Agora que você aprendeu como criar dataframes e extrair informações. Utilize o emulador a seguir para testar o exemplo apresentado e crie o seu próprio DataFrame e extraia informações.

## SELEÇÃO DE COLUNAS EM UM DATAFRAME

Podemos realizar operações em colunas específicas de um DataFrame ou ainda criar um novo objeto contendo somente as colunas que serão usadas em uma determinada análise. Para selecionar uma coluna, as duas possíveis sintaxes são:

1. nome\_df.nome\_coluna

2. nome\_df[nome\_coluna]

A primeira forma é familiar aos desenvolvedores que utilizar a linguagem SQL, porém ela não aceita colunas com espaços entre as palavras. Já a segunda aceita. Se precisarmos selecionar mais do que uma coluna, então precisamos passar uma lista, da seguinte forma: nome\_df[['col1', 'col2', 'col3']], se preferir a lista pode ser criada fora da seção e passada como parâmetro.

Ao selecionar uma coluna, obtemos uma Series, consequentemente, podemos aplicar os atributos e métodos que aprendemos, por exemplo, para obter a média aritmética de uma determinada coluna. Observe os códigos a seguir. Na entrada

14, fizemos a seleção de uma única coluna "idades", veja na impressão que o tipo do objeto agora é uma Series. Na linha 4, a partir desse novo objeto, imprimimos a média de idade. Já na entrada 15, criamos uma lista com as colunas que queremos selecionar e na linha 2, passamos essa lista para seleção (df\_dados[colunas]), consequentemente, obtivemos um novo DataFrame, mas agora com duas coluna

Através da seleção de certas colunas podemos extrair informações específicas e até compará-las com outras colunas ou com outros dados. Esse recurso é muito utilizado por quem trabalha na área de dados.

In [14]:

```
df_uma_coluna = df_dados['idades']
print(type(df_uma_coluna))

print('Média de idades = ', df_uma_coluna.mean())
```

```
df_uma_coluna
```

```
<class 'pandas.core.series.Series'>
Média de idades =  29.2
```

Out[14]:

```
0    32
1    22
2    25
3    29
4    38
Name: idades, dtype: int64
```

In [15]:

```
colunas = ['nomes', 'cpfs']
df_duas_colunas = df_dados[colunas]
print(type(df_duas_colunas))
df_duas_colunas
```

```
<class 'pandas.core.frame.DataFrame'>
```

o

Ver anotações

Out[15]:

	nomes	cpfs
0	Howard	111.111.111-11
1	Ian	222.222.222-22
2	Peter	333.333.333-33
3	Jonah	444.444.444-44
4	Kellie	555.555.555-55

0

Ver anotações

### EXEMPLIFICANDO

Vamos utilizar tudo que já aprendemos e fazer uma atividade de raspagem web (web scraping). Vamos acessar a seguinte página de notícias do jornal New York Times: <https://nyti.ms/3aHRu2D>. A partir dessa fonte de informações vamos trabalhar para criar um DataFrame contendo o dia da notícia, o comentário que foi feito, a explicação que foi dada e o link da notícia.

Vamos começar nossa raspagem utilizando um recurso que já nos é familiar, a biblioteca requests! Fazer a extração da notícia com o `request.get()` convertendo tudo para uma única string, por isso vamos usar a propriedade `text`. Na linha 4, da entrada 16 imprimimos os 100 primeiros caracteres do texto que capturamos. Veja que foi lido o conteúdo HTML da página.

In [16]:

```
import requests

texto_string =
requests.get('https://www.nytimes.com/interactive/2017/06/23/opinion/trump
lies.html').text
print(texto_string[:100])
```

```
<!DOCTYPE html>
<!--[if (gt IE 9)|(IE)]> <!--><html lang="en" class="no-js page-
interactive section
```

0

Ver anotações

Como temos um conteúdo em HTML é conveniente utilizar a biblioteca BeautifulSoup, para converter a string em uma estrutura HTML e então filtrar determinadas tags. Veja na entrada 17, estamos importando a biblioteca e através da classe BeautifulSoup, instanciamos um objeto passando o texto, em string, e o parâmetro '**html.parser**'. Agora, com o objeto do tipo BeautifulSoup, podemos usar o método **find\_all()** para buscar todas as ocorrências de uma determinada tag, no nosso caso estamos buscando pelas tags span, que contenham um atributo 'class': 'short-desc'. O resultado dessa busca é uma conjunto iterável (class 'bs4.element.ResultSet'), como se fosse uma lista, então na linha 8, estamos exibindo a notícia no índice 5 desse iterável e na linha 10, estamos exibindo o "conteúdo" desse mesmo elemento, veja que contents, retorna uma lista do conteúdo. Obs: para saber qual tag buscar, antes é preciso examinar o código fonte da página que se deseja "raspar".

In [17]:

```
from bs4 import BeautifulSoup

bsp_texto = BeautifulSoup(texto_string, 'html.parser')
lista_noticias = bsp_texto.find_all('span', attrs={'class': 'short-desc'})

print(type(bsp_texto))
print(type(lista_noticias))
print(lista_noticias[5])

lista_noticias[5].contents
```

```
<class 'bs4.BeautifulSoup'>
<class 'bs4.element.ResultSet'>
<span class="short-desc"><strong>Jan. 25 </strong>“You had millions of
people that now aren't insured anymore.” <span class="short-truth"><a
href="https://www.nytimes.com/2017/03/13/us/politics/fact-check-trump-
obamacare-health-care.html" target="_blank">(The real number is less than
1 million, according to the Urban Institute.)</a></span></span>
```

0

Ver anotações

Out[17]:

```
[<strong>Jan. 25 </strong>,
““You had millions of people that now aren't insured anymore.” ”,
<span class="short-truth"><a
href="https://www.nytimes.com/2017/03/13/us/politics/fact-check-trump-
obamacare-health-care.html" target="_blank">(The real number is less than
1 million, according to the Urban Institute.)</a></span>]
```

Na entrada 18, criamos uma estrutura de repetição que vai percorrer cada notícia do objeto iterável do tipo `bs4.element.ResultSet`, extraindo as informações que queremos. Para explicar cada linha de comando, vamos considerar a saída dos dados obtidas anteriormente, ou seja, a notícia na posição 5.

- **Linha 1:** Criamos uma lista vazia.
- **Linha 4:** O código `noticia.contents[0]` retorna: `<strong>Jan. 25 </strong>`, ao acessar a propriedade `text`, eliminamos as tags, então temos `Jan. 25`. Usamos a função `strip()` para eliminar espaço em branco na string e concatenamos com o ano.
- **Linha 5:** O código `contents[1]` retorna: `““You had millions of people that
now aren't insured anymore.” ”` usamos o `strip()` para eliminar espaços em branco e a função `replace` para substituir os caracteres especiais por nada.
- **Linha 6:** O código `noticia.contents[2]` retorna: `<a
href="https://www.nytimes.com/2017/03/13/us/politics/fact-check-`

*trump-obamacare-health-care.html" target="\_blank"*  
*>(The real number is less than 1 million, according to the Urban*  
*Institute.)</a></span>, ao acessar a propriedade text, eliminamos as*  
*tags então temos (The real number is less than 1 million, according to*  
*the Urban Institute.), o qual ajustamos para eliminar espaços e os*  
*parênteses.*

- Linha 7: o código `noticia.find('a')['href']` retorna:  
*<https://www.nytimes.com/2017/03/13/us/politics/fact-check-trump-obamacare-health-care.html>*
- Aprendemos a nossa lista de dados uma tupla com as quatro informações que extraímos.

In [18]:

```
dados = []  
  
for noticia in lista_noticias:  
    data = noticia.contents[0].text.strip() + ', 2017' # Dessa informação  
<strong>Jan. 25 </strong> vai extrair Jan. 25, 2017  
    comentario = noticia.contents[1].strip().replace("“",  
        '').replace("”", '')  
    explicacao = noticia.contents[2].text.strip().replace("( ",  
        ').replace(" )", '')  
    url = noticia.find('a')['href']  
    dados.append((data, comentario, explicacao, url))  
  
dados[1]
```

Out[18]:

o

Ver anotações

```
('Jan. 21, 2017',  
 'A reporter for Time magazine – and I have been on their cover 14 or 15  
 times. I think we have the all-time record in the history of Time  
 magazine.',  
 'Trump was on the cover 11 times and Nixon appeared 55 times.',  
 'http://nation.time.com/2013/11/06/10-things-you-didnt-know-about-  
 time/')
```

0

Ver anotações

Agora que temos nossa lista de tuplas com os dados, podemos criar o DataFrame e disponibilizar para um cientista de dados fazer a análise de sentimentos. Veja que na entrada 19, usamos o construtor DataFrame passando os dados e o nome das colunas. Pelo atributo shape conseguimos de saber que foram extraídas 180 notícias e, que cada coluna possui o tipo object (que já era esperado por ser texto).

In [19]:

```
df_noticias = pd.DataFrame(dados, columns=['data', 'comentário',  
 'explicação', 'url'])  
  
print(df_noticias.shape)  
print(df_noticias.dtypes)  
df_noticias.head()
```

```
(180, 4)  
data          object  
comentário    object  
explicação   object  
url           object  
dtype: object
```

Out[19]:

data	comentário	explicação	url
Jan. 021, 2017	I wasn't a fan of Iraq. I didn't want to go in...	He was for an invasion before he was against it.	<a href="https://www.buzzfeed.com/andrewkaczynski/trump-wasnt-a-fan-of-iraq">https://www.buzzfeed.com/andrewkaczynski/trump-wasnt-a-fan-of-iraq</a> <span style="float: right;">0 Ver anotações</span>
Jan. 121, 2017	A reporter for Time magazine — and I have been...	Trump was on the cover 11 times and Nixon appeared...	<a href="http://nation.time.com/2013/11/06/10-things-you-didnt-know-about-trump/">http://nation.time.com/2013/11/06/10-things-you-didnt-know-about-trump/</a>
Jan. 223, 2017	Between 3 million and 5 million illegal votes ...	There's no evidence of illegal voting.	<a href="https://www.nytimes.com/2017/01/23/us/politics/electoral-vote-fraud.html">https://www.nytimes.com/2017/01/23/us/politics/electoral-vote-fraud.html</a>
Jan. 325, 2017	Now, the audience was the biggest ever. But th...	Official aerial photos show Obama's 2009 inauguration.	<a href="https://www.nytimes.com/2017/01/21/us/politics/trumps-inauguration.html">https://www.nytimes.com/2017/01/21/us/politics/trumps-inauguration.html</a>
Jan. 425, 2017	Take a look at the Pew reports 2017 (which show vot...	The report never mentioned voter fraud.	<a href="https://www.nytimes.com/2017/01/24/us/politics/trumps-voter-fraud.html">https://www.nytimes.com/2017/01/24/us/politics/trumps-voter-fraud.html</a>

## LEITURA DE DADOS ESTRUTURADOS COM A BIBLIOTECA PANDAS

Um dos grandes recursos da biblioteca pandas é sua capacidade de fazer leitura de dados estruturados, através de seus métodos, guardando em um DataFrame. A biblioteca possui uma série de métodos "read", cuja sintaxe é:

pandas.read\_xxxxx() onde a sequência de X representa as diversas opções disponíveis. Para finalizar nossa aula, vamos ver como fazer a leitura de uma tabela em uma página web, utilizando o método pandas.read\_html(). A documentação desse método está disponível em [https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read\\_html.html](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_html.html) e seu construtor possui os seguintes parâmetros: pandas.read\_html(io, match='.+', flavor=None, header=None, index\_col=None, skiprows=None, attrs=None, parse\_dates=False, thousands=',', encoding=None, decimal='.', converters=None, na\_values=None, keep\_default\_na=True, displayed\_only=True). Dentre todos, somente o "io" é o que recebe a URL a ser usada. Esse método procura por tags <table> na estrutura do código HTML e devolve uma lista de DataFrames contendo as tabelas que localizou.

Na URL <https://www.fdic.gov/bank/individual/failed/banklist.html>, encontra-se uma tabela com bancos norte americanos que faliram desde 1º de outubro de 2000, cada linha representa um banco. Vamos utilizar o método read\_html() para capturar os dados e carregar em um DataFrame. Observe o código na entrada 20, o método read\_html capturou todas as tabelas no endereço passado como parâmetro, sendo que cada tabela é armazenada em um DataFrame e o método retorna uma lista com todos eles. Veja na linha 4, que ao imprimirmos o tipo do resultado guardado na variável dfs, obtemos uma lista e ao verificarmos quantos DataFrames foram criados (len(dfs)), somente uma tabela foi encontrada, pois o tamanho da lista é 1.

In [20]:

Ver anotações

```
url = 'https://www.fdic.gov/bank/individual/failed/banklist.html'  
dfs = pd.read_html(url)  
  
print(type(dfs))  
print(len(dfs))  
<class 'list'>  
1
```

0

Ver anotações

Sabendo que o tamanho da lista resultado do método é 1, então para obter a tabela que queremos, basta acessar a posição 0 da lista. Observe na entrada 21, guardamos o único DataFrame da lista em uma nova variável, verificamos quantas linhas existem e quais os tipos de cada coluna, com exceção da coluna CERT, todas as demais são texto. Usamos o método head para ver os cinco primeiros registros do DataFrame.

In [21]:

```
df_bancos = dfs[0]  
  
print(df_bancos.shape)  
print(df_bancos.dtypes)  
  
df_bancos.head()
```

```
(561, 6)  
Bank Name          object  
City              object  
ST                object  
CERT             int64  
Acquiring Institution    object  
Closing Date       object  
dtype: object
```

Out[21]:

	<b>Bank Name</b>	<b>City</b>	<b>ST</b>	<b>CERT</b>	<b>Acquiring Institution</b>	<b>Closing Date</b>
<b>0</b>	The First State Bank	Barboursville	WV	14361	MVB Bank, Inc.	April 3, 2020
<b>1</b>	Ericson State Bank	Ericson	NE	18265	Farmers and Merchants Bank	February 14, 2020
<b>2</b>	City National Bank of New Jersey	Newark	NJ	21111	Industrial Bank	November 1 2019
<b>3</b>	Resolute Bank	Maumee	OH	58317	Buckeye State Bank	October 25, 2019
<b>4</b>	Louisa Community Bank	Louisa	KY	58112	Kentucky Farmers Bank Corporation	October 25, 2019

Ver anotações

## REFERÊNCIAS E LINKS ÚTEIS

PyPI. Python Package Index. Disponível em: <https://pypi.org/>. Acesso em: 17 jun. 2020.

Leonard Richardson. Beautiful Soup Documentation. Disponível em: <https://www.crummy.com/software/BeautifulSoup/bs4/doc/>. Acesso em: 17 jun. 2020.

Pandas Team. About pandas. Disponível em: <https://pandas.pydata.org/about/>. Acesso em: 17 jun. 2020.

Pandas Team. DataFrame. Disponível em: <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html>. Acesso em: 17 jun. 2020.

Pandas Team. pandas documentation. Disponível em: <https://pandas.pydata.org/pandas-docs/stable/index.html>. Acesso em: 17 jun. 2020.

Pandas Team. Package overview. Disponível em: [https://pandas.pydata.org/docs/getting\\_started/overview.html](https://pandas.pydata.org/docs/getting_started/overview.html). Acesso em: 17 jun. 2020.

Pandas Team. Series. Disponível em: <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.html>. Acesso em: 17 jun. 2020.

0

Ver anotações

# FOCO NO MERCADO DE TRABALHO

## INTRODUÇÃO A BIBLIOTECA PANDAS

Vanessa Cadan Scheffer

0  
Ver anotações

### WEB SCRAPING

Técnica de extração de dados utilizada para coletar dados de sites através de tags HTML e atributos CSS.



Fonte: Shutterstock.

### Deseja ouvir este material?

Áudio disponível no material digital.

### DESAFIO

Em um artigo publicado no dia 06 de março de 2019, no portal Computer World, o autor fala sobre o profissional que deseja seguir a carreira de analista de dados, o qual deve ter habilidades em: filtrar dados, construir APIs, web scraping e ter conhecimento nas linguagens Git, MySQL e Python. (MOTIM, Raphael Bueno da. Carreira de analista de dados oferece salários de até R\$ 12,5 mil. 2019. Disponível em: <https://computerworld.com.br/2019/03/06/carreira-de-analista-de-dados-oferece-salarios-de-ate-r-125-mil/>. Acesso em: 17 jun. 2020).

Como desenvolvedor em uma empresa de consultoria de software, você foi alocado em uma equipe de marketing analítico em uma marca esportiva, que necessita fazer a coleta das principais notícias do mundo de esporte em um determinado portal. O cliente pediu para que o portal <https://globoesporte.globo.com/>. O cliente deseja um componente capaz de fazer a extração dos dados em forma tabular, com os seguintes campos: manchete,

descrição, link, seção, hora da extração, tempo decorrido da publicação até a hora da extração. O Quadro 4.1 apresenta, visualmente, como os dados devem ser organizados e exibidos.

0  
Ver anotações

Quadro 4.1 - Resultado esperado

<b>manchete</b>	<b>Descrição</b>	<b>link</b>	<b>seção</b>	<b>hora_extração</b>	<b>time_delta</b>
Título da manchete (quando houver)	Descrição da manchete para a notícia	link para a notícia	Seção que a notícia foi marcada	Data e hora da extração	Quanto tempo se passou da hora da publicação até a extração

Fonte: elaborada pela autora.

O grande desafio no trabalho de web scraping é descobrir qual o padrão nas tags HTML e atributos CSS usados. Pois somente através deles é que conseguiremos alcançar a informação solicitada. Como o cliente já trabalha com o portal de notícias, foram lhe passadas as seguintes informações técnicas que o ajudarão a fazer a extração.

Para extração de todas as informações localize todas as div com atributo 'class':'feed-post-body'. De cada item localizado extraia:

- A manchete que ocupa a primeira posição do conteúdo.
- O link que pode ser localizado pela tag "a" e pelo atributo "href".
- A descrição pode estar na terceira posição conteúdo ou dentro de uma div com atributo 'class':'bstn-related'
- A seção está dentro de uma div com atributo 'class':'feed-post-metadata'.
  - Localize o span com atributo 'class': 'feed-post-metadata-section'.
- O tempo decorrido está uma div com atributo 'class':'feed-post-metadata'.
  - Localize o span com atributo 'class': 'feed-post-datetime'.

Caso tente acessar o texto de uma tag não localizada, um erro é dado, para evitar esses casos, os campos descrição, seção e time\_delta devem ser tratados para esses casos, retornando None (nulo). Agora é com você, faça a implementação e gere um DataFrame com as informações solicitadas.

## RESOLUÇÃO

Para fazer o web scraping solicitado, vamos utilizar as bibliotecas requests, BeautifulSoup, pandas e datetime. As duas primeiras serão usadas para fazer a captura do conteúdo da página, pandas para entregar os resultados em forma estruturada e datetime para marcar o dia e hora da extração.

In [22]:

```
from datetime import datetime  
  
import requests  
  
from bs4 import BeautifulSoup  
  
import pandas as pd
```

Com as bibliotecas importadas, vamos acessar o portal e utilizar a propriedade text da biblioteca requests para capturar em formato de string. Em seguida, vamos transformar essa string em formato html, para que possamos localizar as tags de nosso interesse. Na linha 2, registramos o horário da extração. Na linha 5, procuramos todas as tags div com o atributo que nos foi indicado. Essa linha retornará uma lista com cada notícia. Veja que na linha 6 imprimimos quantas notícias foram encontradas e na linha 7 imprimimos o conteúdo da primeira notícia. Lembre-se que contents transforma cada início e final da div em um elemento da lista.

In [23]:

```
texto_string = requests.get('https://globoesporte.globo.com/').text  
hora_extracao = datetime.now().strftime("%d-%m-%Y %H:%M:%S")  
  
bsp_texto = BeautifulSoup(texto_string, 'html.parser')  
lista_noticias = bsp_texto.find_all('div', attrs={'class':'feed-post-body'})  
print("Quantidade de manchetes = ", len(lista_noticias))  
lista_noticias[0].contents
```

```
Quantidade de manchetes = 10
```

Out[23]:

```
[<div class="feed-post-header"></div>,<div class="_label_event"><div class="feed-post-body-title gui-color-primary gui-color-hover"><div class="_ee"><a class="feed-post-link gui-color-primary gui-color-hover" href="https://globoesporte.globo.com/futebol/futebol-internacional/futebol-italiano/jogo/17-06-2020/napoli-juventusita.ghtml">VICE SENHORA</a></div></div></div>,<div class="_label_event"><div class="feed-post-body-resumo">Napoli vence Juventus nos pênaltis e leva Copa da Itália</div></div>,<div class="feed-media-wrapper"><div class="_label_event"><a class="feed-post-figure-link gui-image-hover" href="https://globoesporte.globo.com/futebol/futebol-internacional/futebol-italiano/jogo/17-06-2020/napoli-juventusita.ghtml"><div class="bstn-fd-item-cover"><picture class="bstn-fd-cover-picture"></picture></div></a></div></div>,<div class="feed-post-metadata"><span class="feed-post-datetime">Há 3 horas</span><span class="feed-post-metadata-section"> futebol italiano </span></div>]
```

Dentro dessa estrutura, procurando pelas tags corretas, vamos encontrar todas as informações que foram solicitadas. Pela saída anterior podemos ver que a manchete ocupa a posição 2 da lista de conteúdos, logo para guardar a manchete devemos fazer:

In [24]:

```
lista_noticias[0].contents[1].text.replace(' ', '')
```

Out[24]:

```
'VICE SENHORA'
```

Para extração do link para notícia, como ele se encontra também na posição 1 da lista, vamos utilizar o método `find('a')` para localizá-lo e extrair da seguinte forma:

In [25]:

```
lista_noticias[0].find('a').get('href')
```

Out[25]:

```
'https://globoesporte.globo.com/futebol/futebol-internacional/futebol-italiano/jogo/17-06-2020/napoli-juventusita.ghtml'
```

Para a descrição, como ela pode estar na terceira posição ou em outra tag, vamos ter que testar em ambas e caso não esteja, então retornar None (nulo). Veja a seguir.

In [26]:

```
descricao = lista_noticias[0].contents[2].text
if not descricao:
    descricao = noticia.find('div', attrs={'class': 'bstn-related'})
    descricao = descricao.text if descricao else None # Somente acessará a propriedade text caso tenha encontrado ("find")
descricao
```

Out[26]:

```
'Napoli vence Juventus nos pênaltis e leva Copa da Itália'
```

Para extração da seção e do tempo decorrido, vamos acessar primeiro o atributo 'feed-post-metadata' e guardar em uma variável, para em seguida, dentro desse novo subconjunto, localizar os atributos 'feed-post-datetime' e 'feed-post-metadata-section'. Como existe a possibilidade dessa informação não existir, precisamos garantir que somente acessaremos a propriedade text (linhas 6 e 7) caso tenha encontrando ("find"). Veja a seguir

In [27]:

```
metadados = lista_noticias[0].find('div', attrs={'class': 'feed-post-metadata'})

time_delta = metadados.find('span', attrs={'class': 'feed-post-datetime'})
secao = metadados.find('span', attrs={'class': 'feed-post-metadata-section'})

time_delta = time_delta.text if time_delta else None
secao = secao.text if secao else None

print('time_delta = ', time_delta)
print('seção = ', secao)

time_delta = Há 3 horas
seção = futebol italiano
```

Veja que para a notícia 0 extraímos todas as informações solicitadas, mas precisamos extrair de todas, portanto cada extração deve ser feita dentro de uma estrutura de repetição. Para criar um DataFrame com os dados, vamos criar uma lista vazia e a cada iteração apendar uma tupla com as informações extraídas. Com essa lista, podemos criar nosso DataFrame, passando os dados e os nomes das colunas. Veja a seguir:

In [28]:

```

dados = []

for noticia in lista_noticias:
    manchete = noticia.contents[1].text.replace(' ', '')
    link = noticia.find('a').get('href')

    descricao = noticia.contents[2].text
    if not descricao:
        descricao = noticia.find('div', attrs={'class': 'bstn-related'})
        descricao = descricao.text if descricao else None

    metadados = noticia.find('div', attrs={'class': 'feed-post-metadata'})
    time_delta = metadados.find('span', attrs={'class': 'feed-post-datetime'})
    secao = metadados.find('span', attrs={'class': 'feed-post-metadata-section'})

    time_delta = time_delta.text if time_delta else None
    secao = secao.text if secao else None

    dados.append((manchete, descricao, link, secao, hora_extracao, time_delta))

df = pd.DataFrame(dados, columns=['manchete', 'descrição', 'link', 'seção',
                                   'hora_extração', 'time_delta'])
df.head()

```

Out[28]:

	<b>manchete</b>	<b>descrição</b>		<b>link</b>	<b>seção</b>	<b>hora_ext</b>
0	VICE SENHORA	Napoli vence Juventus nos pênaltis e leva Copa...	https://globoesporte.globo.com/futebol/futebol...		futebol italiano	17-06-2021 18:58:17
1	ESPERA AÍ, LIVERPOOL	Em noite trágica de David Luiz, Manchester Cit...	https://globoesporte.globo.com/futebol/futebol...		futebol inglês	17-06-2021 18:58:17
2	BASTIDORES CONTURBADOS	João Doria só libera volta aos treinos a parti...	https://globoesporte.globo.com/sp/futebol/camp...		campeonato paulista	17-06-2021 18:58:17

0

Ver anotações

	<b>manchete</b>	<b>descrição</b>		<b>link</b>	<b>seção</b>	<b>hora_ext</b>	
3	Na véspera de Flamengo e Bangu, Ferj lança nov...	Maracanã passa por processos de higienização e...	https://globoesporte.globo.com/futebol/noticia...	futebol	17-06-2022	18:52	Ver anotações
4	Doutor na terra do Padim Ciço: as memórias do ...	Partida em junho de 1984 marcou a despedida de...	https://globoesporte.globo.com/ce/futebol/noti...	futebol	17-06-2022	18:58:17	

Vamos tornar nossa entrega mais profissional e transformar a solução em uma classe, assim toda vez que for preciso fazer a extração, basta instanciar um objeto e executar o método de extração.

In [29]:

```
from datetime import datetime

import requests
from bs4 import BeautifulSoup
import pandas as pd

class ExtracaoPortal:
    def __init__(self):
        self.portal = None

    def extrair(self, portal):
        self.portal = portal
        texto_string = requests.get('https://globoesporte.globo.com/').text
        hora_extracao = datetime.now().strftime("%d-%m-%Y %H:%M:%S")

        bsp_texto = BeautifulSoup(texto_string, 'html.parser')
        lista_noticias = bsp_texto.find_all('div', attrs={'class': 'feed-post-body'})
        dados = []

        for noticia in lista_noticias:
            manchete = noticia.contents[1].text.replace('\'', '')
            link = noticia.find('a').get('href')

            descricao = noticia.contents[2].text
            if not descricao:
                descricao = noticia.find('div', attrs={'class': 'bbtn-related'})
                descricao = descricao.text if descricao else None

            metadados = noticia.find('div', attrs={'class': 'feed-post-metadata'})
            time_delta = metadados.find('span', attrs={'class': 'feed-post-datetime'})
            secao = metadados.find('span', attrs={'class': 'feed-post-metadata-section'})

            time_delta = time_delta.text if time_delta else None
            secao = secao.text if secao else None

            dados.append((manchete, descricao, link, secao, hora_extracao,
time_delta))

        df = pd.DataFrame(dados, columns=['manchete', 'descrição', 'link',
'seção', 'hora_extração', 'time_delta'])
        return df
```

In [30]:

```
df = ExtracaoPortal().extrair("https://globoesporte.globo.com/")
df.head()
```

Out[30]:

	<b>manchete</b>	<b>descrição</b>	<b>link</b>	<b>seção</b>	<b>hor</b>	<b>tr</b>
0	VICE SENHORA	Napoli vence Juventus nos pênaltis e leva Copa...	https://globoesporte.globo.com/futebol/futebol...	futebol italiano	17-06-202 18:55	Ver anotações
1	ESPERA AÍ, LIVERPOOL	Em noite trágica de David Luiz, Manchester Cit...	https://globoesporte.globo.com/futebol/futebol...	futebol inglês	17-06-202 18:58:18	
2	BASTIDORES CONTURBADOS	João Doria só libera volta aos treinos a partir...	https://globoesporte.globo.com/sp/futebol/camp...	campeonato paulista	17-06-202 18:58:18	
3	Na véspera de Flamengo e Bangu, Ferj lança nov...	Maracanã passa por processos de higienização e...	https://globoesporte.globo.com/futebol/noticia...	futebol	17-06-202 18:58:18	
4	Doutor na terra do Padim Ciço: as memórias do ...	Partida em junho de 1984 marcou a despedida de...	https://globoesporte.globo.com/ce/futebol/noti...	futebol	17-06-202 18:58:18	

## DESAFIO DA INTERNET

Ganhar habilidade em programação exige estudo e treino (muito treino). Acesse o endereço <https://medium.com/data-hackers/como-fazer-web-scraping-em-python-23c9d465a37f> e pratique um pouco mais essa habilidade!

NÃO PODE FALTAR

# INTRODUÇÃO A MANIPULAÇÃO DE DADOS EM PANDAS

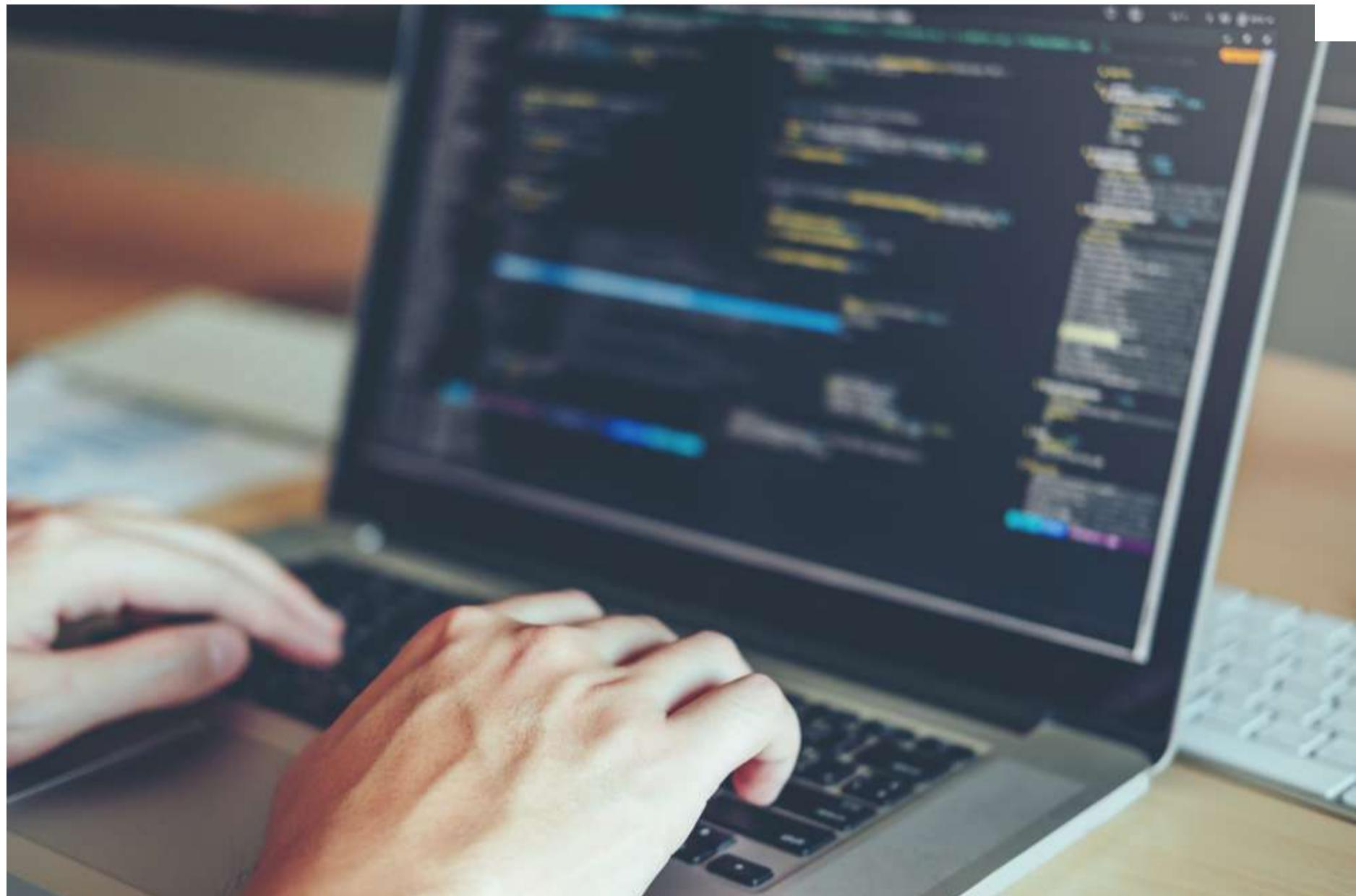
Vanessa Cadan Scheffer

0

Ver anotações

## MÉTODOS PARA MANIPULAÇÃO DE DADOS

Além dos métodos para carregar e salvar dados, a biblioteca pandas possui também métodos para a transformação dos dados e a extração de informação.



Fonte: Shutterstock.

### **Deseja ouvir este material?**

Áudio disponível no material digital.

## MÉTODOS PARA LEITURA E ESCRITA DA BIBLIOTECA PANDAS

A biblioteca pandas foi desenvolvida para trabalhar com dados estruturados, ou seja, dados dispostos em linhas e colunas. Os dados podem estar gravados em arquivos, em páginas web, em APIs, em outros softwares, em object stores

(sistemas de armazenamento em cloud) ou em bancos de dados. Para todas essas origens (e até mais), a biblioteca possui métodos capazes de fazer a leitura dos dados e carregar em um DataFrame.

Todos os métodos capazes de fazer a leitura dos dados estruturados possuem prefixo `pd.read_xxx`, onde `pd` é o apelido dado no momento da importação da biblioteca e `xxx` é o restante da sintaxe do método. Além de fazer a leitura a biblioteca possui diversos métodos capazes de escrever o DataFrame em um arquivo, em um banco ou ainda simplesmente copiar para a área de transferência do sistema operacional. O Quadro 4.2, apresenta todos os métodos de leitura e escrita. Veja que são suportados tanto a leitura de arquivos de texto, como binários e de bancos.

0  
Ver anotações

Quadro 4.2 - Métodos para leitura e escrita de dados estruturados

<b>Tipo de dado</b>	<b>Descrição do dado</b>	<b>Método para leitura</b>	<b>Método para escrita</b>
texto	<a href="#">CSV</a>	<code>read_csv</code>	<code>to_csv</code>
texto	Fixed-Width texto  File	<code>read_fwf</code>	
texto	<a href="#">JSON</a>	<code>read_json</code>	<code>to_json</code>
texto	<a href="#">HTML</a>	<code>read_html</code>	<code>to_html</code>
texto	Local clipboard	<code>read_clipboard</code>	<code>to_clipboard</code>
	<a href="#">MS Excel</a>	<code>read_excel</code>	<code>to_excel</code>
binário	<a href="#">OpenDocument</a>	<code>read_excel</code>	
binário	<a href="#">HDF5 Format</a>	<code>read_hdf</code>	<code>to_hdf</code>

<b>Tipo de dado</b>	<b>Descrição do dado</b>	<b>Método para leitura</b>	<b>Método para escrita</b>
binário	<a href="#">Feather Format</a>	read_feather	to_feather
binário	<a href="#">Parquet Format</a>	read_parquet	to_parquet
binário	<a href="#">ORC Format</a>	read_orc	
binário	<a href="#">Msgpack</a>	read_msgpack	to_msgpack
binário	<a href="#">Stata</a>	read_stata	to_stata
binário	<a href="#">SAS</a>	read_sas	
binário	<a href="#">SPSS</a>	read_spss	
binário	<a href="#">Python Pickle Format</a>	read_pickle	to_pickle
SQL	<a href="#">SQL</a>	read_sql	to_sql
SQL	<a href="#">Google BigQuery</a>	read_gbq	to_gbq

0

Ver anotações

Fonte: adaptado de [https://pandas.pydata.org/pandas-docs/stable/user\\_guide/io.html](https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html)

Dentre todos os possíveis métodos para leitura, nessa aula vamos estudar o `read_json`, o `read_csv` e a função `read_sql`, que contempla a função `read_sql_query`.

**JSON** (JavaScript Object Notation - Notação de Objetos JavaScript) é uma formatação leve de troca de dados e independente de linguagem de programação. Os dados nesse formato podem ser encontrados como uma coleção de pares chave/valor ou uma lista ordenada de valores (<https://www.json.org/json-pt.html>). Uma chave pode conter uma outra estrutura interna, criando um arquivo com multiníveis. Nessa aula vamos estudar somente a leitura de arquivos com um único nível, ou seja, para uma chave há somente um valor.

**CSV** (comma-separated values - valores separados por vírgula) é um formato de arquivo, nos quais os dados são separados por um delimitador. Originalmente, esse delimitador é uma vírgula (por isso o nome), mas na prática um arquivo CSV pode ser criado com qualquer delimitador, por exemplo, por ponto e vírgula (;), por pipe (|), dentre outros. Por ser um arquivo de texto, é fácil de ser lido em qualquer sistema, por isso se tornou tão democrático.

Ver anotações

## | LEITURA DE JSON E CSV COM PANDAS

Agora que vamos começar a ver as implementações, vamos fazer a importação da biblioteca, como já sabemos, só precisamos importar uma única vez no notebook ou no script .py.

In [1]:

```
import pandas as pd
```

A leitura de um arquivo JSON deve ser feita com o método:

```
pandas.read_json(path_or_buf=None, orient=None, typ='frame', dtype=None,  
convert_axes=None, convert_dates=True, keep_default_dates=True, numpy=False,  
precise_float=False, date_unit=None, encoding=None, lines=False, chunksize=None,  
compression='infer'). Os detalhes de cada parâmetro podem ser consultados na  
documentação oficial: https://pandas.pydata.org/pandas-  
docs/stable/reference/api/pandas.read\_json.html. O único parâmetro que é  
obrigatório para se carregar os dados é o "path_or_buf", no qual deve ser passado  
um caminho para o arquivo ou um "arquivo como objeto" que é um arquivo lido  
com a função open(), por exemplo.
```

Na entrada 2, estamos usando o método `read_json` para carregar dados de uma API. Veja que estamos passando o caminho para o método. Nessa fonte de dados, são encontradas a taxa selic de cada dia.

In [2]:

```
pd.read_json("https://api.bcb.gov.br/dados/serie/bcdata.sgs.11/dados?  
formato=json").head()
```

Out[2]:

	<b>data</b>	<b>valor</b>
<b>0</b>	04/06/1986	0.065041
<b>1</b>	05/06/1986	0.067397
<b>2</b>	06/06/1986	0.066740
<b>3</b>	09/06/1986	0.068247
<b>4</b>	10/06/1986	0.067041

0

[Ver anotações](#)

A leitura de um arquivo CSV deve ser feita com o método:

```
pandas.read_csv(filepath_or_buffer: Union[str, pathlib.Path, IO[~ AnyStr]],  
sep=',', delimiter=None, header='infer', names=None, index_col=None,  
usecols=None, squeeze=False, prefix=None, mangle_dupe_cols=True, dtype=None,  
engine=None, converters=None, true_values=None, false_values=None,  
skipinitialspace=False, skiprows=None, skipfooter=0, nrows=None, na_values=None,  
keep_default_na=True, na_filter=True, verbose=False, skip_blank_lines=True,  
parse_dates=False, infer_datetime_format=False, keep_date_col=False,  
date_parser=None, dayfirst=False, cache_dates=True, iterator=False,  
chunksize=None, compression='infer', thousands=None, decimal: str = '.',  
lineterminator=None, quotechar='"', quoting=0, doublequote=True,  
escapechar=None, comment=None, encoding=None, dialect=None,  
error_bad_lines=True, warn_bad_lines=True, delim_whitespace=False,  
low_memory=True, memory_map=False, float_precision=None). São muitos parâmetros,  
o que proporciona uma versatilidade incrível para esse método. Os detalhes de  
cada parâmetro podem ser consultados na documentação oficial:  
https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read\_csv.html. Vamos comentar alguns.
```

Para realizar o carregamento os dados, é necessário incluir o caminho(diretório), portanto o parâmetro "filepath\_or\_buffer" é obrigatório. Outro parâmetro que é importante para a leitura desse arquivo é o sep ou delimiter (ambos fazem a mesma coisa), veja que sep, por padrão possui o valor ',', ou seja, caso não seja especificado nenhum valor, então o método fará a leitura dos dados considerando que estão separados por vírgula. O parâmetro header, tem como valor padrão 'infer', que significa que o método realiza a inferência para os nomes das colunas partir da primeira linha de dados do arquivo.

Na entrada 3, estamos fazendo a leitura de uma fonte CSV, cujos campos são separados por vírgula, logo não foi preciso especificar um delimitador diferente.

In [3]:

```
pd.read_csv("https://people.sc.fsu.edu/~jburkardt/data/csv/cities.csv").head()
```

Out[3]:

	<b>LatD</b>	<b>LatM</b>	<b>LatS</b>	<b>"NS"</b>	<b>LonD</b>	<b>LonM</b>	<b>LonS</b>	<b>"EW"</b>	<b>"City"</b>	<b>"State"</b>
<b>0</b>	41	5	59	"N"	80	39	0	"W"	"Youngstown"	OH
<b>1</b>	42	52	48	"N"	97	23	23	"W"	"Yankton"	SD
<b>2</b>	46	35	59	"N"	120	30	36	"W"	"Yakima"	WA
<b>3</b>	42	16	12	"N"	71	48	0	"W"	"Worcester"	MA
<b>4</b>	43	37	48	"N"	89	46	11	"W"	"Wisconsin Dells"	WI

## MANIPULAÇÃO DE DADOS COM PANDAS

Além de vários métodos para carregar e salvar os dados, a biblioteca pandas possui uma diversidade de métodos para a transformação dos dados e a extração de informação para áreas de negócio. Nessa seção vamos conhecer alguns deles.

O trabalho com dados: capturar os dados em suas origens, fazer transformações nos dados a fim de padronizá-los, aplicar técnicas estatísticas clássicas ou algoritmos de machine/deep learning feito por engenheiros e cientistas de dados.

Cada profissional atuando em uma parte específica, dependendo da organização da empresa. Em todo esse trabalho é comum fazer a divisão em duas etapas: (i) captura e transformação/padronização dos dados, (ii) extração de informações.

0

#### EXEMPLIFICANDO

Para entender como essas duas etapas acontecem no dia a dia das empresas, vamos utilizar os dados da taxa Selic. Selic, ou taxa Selic é a referência base de juros da economia brasileira (<https://blog.nubank.com.br/taxa-selic/>). O valor da Selic influencia todo o setor financeiro do Brasil, por exemplo, se você pretende fazer um financiamento ou um investimento, precisa olhar a taxa Selic, pois ela influenciará o valor a ser pago ou ganho. "Selic é a sigla para Sistema Especial de Liquidação e Custódia, um programa totalmente virtual em que os títulos do Tesouro Nacional são comprados e vendidos diariamente por instituições financeiras" (NUBANK, p. 1, 2020). Quem decide, de fato, o valor da Selic é o Copom (Comitê de Política Monetária do Banco Central), que a cada 45 dias se reúne para determinar se a taxa aumenta ou diminui. Na prática, se a taxa está alta, os financiados podem ficar mais caros e o contrário também, se a taxa está mais baixa, então os financiamentos ficam mais baratos. Resumindo, quando a taxa Selic aumenta a economia desacelera, quando ela abaixa a economia aquece, isso é preciso para controlar a inflação.

Ver anotações

Agora que já conhecemos que a taxa Selic influencia, nossos financiamentos, investimentos e até mesmo o que compramos no mercado, vamos extrair as informações disponibilizadas pelo governo e fazer algumas análises. Vamos começar pela etapa de extração e transformação dos dados.

## ETAPA DE CAPTURA E TRANSFORMAÇÃO/ PADRONIZAÇÃO DOS DADOS

o

Ver anotações

A extração dos dados pode ser realizada por meio do método `read_json()` e guardando em um DataFrame (DF) pandas. Ao carregar os dados em um DF, podemos visualizar quantas linhas e colunas, bem como, os tipos de dados em cada coluna, com o método `info()`. Observe a entrada 4 e a saída do código a seguir, o método retorna que o DF possui 8552 registros (entradas) e 2 colunas (A quantidade de linhas certamente será diferente quando executar esse código). O índices são numéricos e variam de 0 a 8551 (N-1). Outras duas informações relevantes que esse método retorna é sobre a quantidade de células sem valor (non-null) e o tipo dos dados nas colunas. Como podemos ver, ambas colunas possuem 8552 valores não nulos, como esse número é igual a quantidade de linhas, então não existem valores faltantes. Quanto ao tipo de dados, o "data" é um object, ou seja, são todos do tipo strings ou existe mistura de tipos; "valor" é um float.

In [4]:

```
df_selic = pd.read_json("https://api.bcb.gov.br/dados/serie/bcdata.sgs.11/dados?formato=json")
```

```
print(df_selic.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 8552 entries, 0 to 8551
Data columns (total 2 columns):
data      8552 non-null object
valor     8552 non-null float64
dtypes: float64(1), object(1)
memory usage: 133.7+ KB
None
```

## REMOVER LINHAS DUPLICADAS

Para o carregamento de uma base de dados, um dos primeiros tratamentos que devemos fazer é remover os dados duplicados. Certamente, qual registro remove depende da área de negócio e do problema a ser resolvido. Por exemplo, queremos manter o registro da compra atual, ou queremos manter a primeira compra. Um DataFrame da biblioteca pandas possui o método `meu_df.drop_duplicates()` que permite fazer essa remoção de dados duplicados.

Observe a entrada 5 da linha de código a seguir, usamos o método para remover as linhas duplicadas, pedindo para manter o último registro (`keep='last'`) e, através do parâmetro `inplace=True`, estamos fazendo com que a transformação seja salva do DataFrame, na prática estamos sobrescrevendo o objeto na memória. **Caso `inplace` não seja passado, a transformação é aplicada, mas não é salva, ou seja, o DF continua da mesma forma anterior a transformação.** Outro parâmetro interessante do método é o `subset`, que permite que a remoção de dados duplicado seja feita com base em uma ou mais colunas.

In [5]:

```
df_selic.drop_duplicates(keep='last', inplace=True)
```

## CRIAR NOVAS COLUNAS

A segunda transformação que veremos é como criar uma nova coluna. A sintaxe é similar a criar uma nova chave em um dicionário: `meu_df[ 'nova_coluna' ] = dado`. Vamos criar uma coluna que adiciona a data de extração das informações. Observe a seguir, do módulo datetime, estamos usando a classe date e o método today(). Na entrada 6, guardamos a data em uma nova coluna, veja que a biblioteca pandas "entende" que esse valor deve ser colocado em todas as linhas. Na linha 7, estamos criando uma nova coluna com o nome do responsável pela extração, veja que basta atribuir a string a uma nova coluna. Temos ainda um problema com o

Ver anotações

tipo de dados das datas, embora cada valor seja do tipo "date", veja que pelo info() ainda obtemos uma coluna object, para que de fato, a biblioteca interprete como um tipo data, vamos ter que utilizar o método da própria biblioteca para fazer a conversão.

0

Ver anotações

In [6]:

```
from datetime import date  
from datetime import datetime as dt  
  
data_extracao = date.today()  
  
df_selic['data_extracao'] = data_extracao  
df_selic['responsavel'] = "Autora"  
  
print(df_selic.info())  
df_selic.head()
```

```
<class 'pandas.core.frame.DataFrame'>  
Int64Index: 8552 entries, 0 to 8551  
Data columns (total 4 columns):  
 data            8552 non-null object  
 valor           8552 non-null float64  
 data_extracao   8552 non-null object  
 responsavel     8552 non-null object  
dtypes: float64(1), object(3)  
memory usage: 334.1+ KB  
None
```

Out[6]:

	<b>data</b>	<b>valor</b>	<b>data_extracao</b>	<b>responsavel</b>
<b>0</b>	04/06/1986	0.065041	2020-07-19	Autora
<b>1</b>	05/06/1986	0.067397	2020-07-19	Autora
<b>2</b>	06/06/1986	0.066740	2020-07-19	Autora

	<b>data</b>	<b>valor</b>	<b>data_extracao</b>	<b>responsavel</b>
3	09/06/1986	0.068247	2020-07-19	Autora
4	10/06/1986	0.067041	2020-07-19	Autora

0

Ver anotações

## | MÉTODO TO\_DATETIME() E ASTYPE()

Trabalhar com o tipo "data" pode trazer vantagens, como por exemplo, ordenar os dados de data mais recente para mais antiga, ou ainda verificar a variação da taxa selic em um determinado período. Vamos utilizar os métodos `pandas.to_datetime()` e `minha_series.astype()` para fazer a conversão e transformar as colunas `data` e `data_extracao`. Observe o código a seguir, onde realizamos a transformação e guardamos dentro da própria coluna, dessa forma os valores são sobreescritos. Na linha 1, usamos a notação `pd.to_datetime()`, porque é um método da biblioteca e não do DF.

Na entrada 7 (linha 1), o método recebe a coluna com os valores a serem alterados (`df_selic['data']`) e um segundo parâmetro, indicando que no formato atual (antes da conversão) o dia está primeiro (`dayfirst=True`). Em seguida, na linha 2, como a coluna "data\_extracao" foi criada com o método `today()`, o formato já está correto para a conversão. Nessa conversão usamos o método `astype`, que transforma os dados de uma coluna (que é uma Series) em um determinado tipo, nesse caso, o tipo `datetime` especificado. Com `astype()` podemos padronizar valores das colunas, por exemplo, transformando todos em `float`, ou `int`, ou `str`, ou outro tipo. Veja que agora, ao usar o método `info()`, temos que ambas colunas são do tipo `datetime` (`datetime` da biblioteca `pandas`). O formato resultante ano-mês-dia é um padrão do `datetime64[ns]`, que segue o padrão internacional, no qual o ano vem primeiro, seguido do mês e por último o dia. Poderíamos usar o `strftime()` para transformar o traço em barra (/), mas aí o resultado seriam strings e não datas.

In [7]:

```
df_selic['data'] = pd.to_datetime(df_selic['data'], dayfirst=True)
df_selic['data_extracao'] = df_selic['data_extracao'].astype('datetime64[ns]')

print(df_selic.info())
df_selic.head()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 8552 entries, 0 to 8551
Data columns (total 4 columns):
data            8552 non-null datetime64[ns]
valor           8552 non-null float64
data_extracao   8552 non-null datetime64[ns]
responsavel    8552 non-null object
dtypes: datetime64[ns](2), float64(1), object(1)
memory usage: 334.1+ KB
None
```

Ver anotações

Out[7]:

	<b>data</b>	<b>valor</b>	<b>data_extracao</b>	<b>responsavel</b>
<b>0</b>	1986-06-04	0.065041	2020-07-19	Autora
<b>1</b>	1986-06-05	0.067397	2020-07-19	Autora
<b>2</b>	1986-06-06	0.066740	2020-07-19	Autora
<b>3</b>	1986-06-09	0.068247	2020-07-19	Autora
<b>4</b>	1986-06-10	0.067041	2020-07-19	Autora

## | SERIES.STR

Muitas vezes precisamos padronizar os valores em colunas, por exemplo, queremos ter certeza que a coluna "responsável" possui todas as palavras padronizadas em letras maiúsculas. Quando selecionamos uma coluna no DF sabemos que o resultado é uma Series e esse objeto tem um recurso "str", que permite aplicar as funções de string para todos os valores da Series.

Observe o trecho de código seguir, selecionamos a coluna responsável, acessamos o recurso str e aplicamos o método upper(). Dessa forma, a biblioteca pandas "entende" que queremos converter todos os valores dessa coluna para letras maiúsculas. Como atribuímos o resultado na própria coluna, o valor antigo é substituído.

0

Ver anotações

In [8]:

```
df_selic['responsavel'] = df_selic['responsavel'].str.upper()  
  
df_selic.head()
```

Out[8]:

	<b>data</b>	<b>valor</b>	<b>data_extracao</b>	<b>responsavel</b>
<b>0</b>	1986-06-04	0.065041	2020-07-19	AUTORA
<b>1</b>	1986-06-05	0.067397	2020-07-19	AUTORA
<b>2</b>	1986-06-06	0.066740	2020-07-19	AUTORA
<b>3</b>	1986-06-09	0.068247	2020-07-19	AUTORA
<b>4</b>	1986-06-10	0.067041	2020-07-19	AUTORA

## ■ MÉTODO SORT\_VALUES()

No código a seguir, estamos usando o método `sort_values()` que permite ordenar DF, de acordo com os valores de uma coluna. Esse método é do DataFrame, por isso a notação `meu_df.metodo()`. Utilizamos três parâmetros do método `sort_values`: o primeiro informando qual coluna deve ser usada para ordenar, o segundo, para que seja feito em ordem decrescente (do maior para o menor) e o terceiro (`inplace=True`) significa que queremos modificar o próprio objeto, na prática estamos sobrescrevendo o DF.

0

Ver anotações

In [9]:

```
df_selic.sort_values(by='data', ascending=False, inplace=True)

df_selic.head()
```

Out[9]:

	<b>data</b>	<b>valor</b>	<b>data_extracao</b>	<b>responsavel</b>
<b>8551</b>	2020-07-17	0.008442	2020-07-19	AUTORA
<b>8550</b>	2020-07-16	0.008442	2020-07-19	AUTORA
<b>8549</b>	2020-07-15	0.008442	2020-07-19	AUTORA
<b>8548</b>	2020-07-14	0.008442	2020-07-19	AUTORA
<b>8547</b>	2020-07-13	0.008442	2020-07-19	AUTORA

## ■ MÉTODO RESET\_INDEX() E SET\_INDEX()

Ao fazermos a ordenação dos dados com o método `sort_values()`, veja que os índices dos cinco primeiros registros é 8551, 8550...85XX. Nenhuma transformação afeta o índice, lembra-se como não especificamos rótulos ele usa um intervalo numérico, mas esse intervalo é **diferente da posições de um vetor**, pois é um nome e vai acompanhar a linha independente da transformação.

As únicas formas de alterar o índice são com os métodos `reset_index()` e `set_index()`. O primeiro redefine o índice usando o padrão e o segundo utiliza define novos índices. Veja o código na entrada 10, estamos usando o método `reset_index()` para redefinir os índices padrão (sequência numérica). O primeiro parâmetro (`drop=True`), diz que não queremos usar o índice que será substituído em uma nova coluna e `inplace`, informa para gravar as alterações no próprio objeto. Veja na saída que agora os cinco primeiros registros, possuem índices de a 4.

o

Ver anotações

In [10]:

```
df_selic.reset_index(drop=True, inplace=True)

df_selic.head()
```

Out[10]:

	<b>data</b>	<b>valor</b>	<b>data_extracao</b>	<b>responsavel</b>
<b>0</b>	2020-07-17	0.008442	2020-07-19	AUTORA
<b>1</b>	2020-07-16	0.008442	2020-07-19	AUTORA
<b>2</b>	2020-07-15	0.008442	2020-07-19	AUTORA
<b>3</b>	2020-07-14	0.008442	2020-07-19	AUTORA
<b>4</b>	2020-07-13	0.008442	2020-07-19	AUTORA

Durante a transformação dos dados, pode ser necessário definir novos valores para os índices, ao invés de usar o range numérico. Essa transformação pode ser feita usando o método `meu_df.set_index()`. O método permite especificar os novos valores usando uma coluna já existente ou então passando uma lista, de tamanho igual a quantidade de linhas.

Observe os códigos nas entradas 11 e 12. Na entrada 11, estamos criando uma lista que usada os índices do DF, adicionando um prefixo para cada valor. Na linha 2 são impressos os cinco primeiros itens da nova lista. Na entrada 12, estamos

definindo o novo índice com base na lista criada. Veja que o parâmetros keys, recebe como parâmetro uma lista de lista e o segundo parâmetro especifica que a modificação deve ser salva no objeto. Na impressão das cinco primeiras linhas do DF, podemos ver o novo índice.

0

In [11]:

```
lista_novo_indice = [f'selic_{indice}' for indice in df_selic.index]
```

```
print(lista_novo_indice[:5])
```

```
['selic_0', 'selic_1', 'selic_2', 'selic_3', 'selic_4']
```

In [12]:

```
df_selic.set_index(keys=[lista_novo_indice], inplace=True)
```

```
df_selic.head()
```

Out[12]:

	<b>data</b>	<b>valor</b>	<b>data_extracao</b>	<b>responsavel</b>
<b>selic_0</b>	2020-07-17	0.008442	2020-07-19	AUTORA
<b>selic_1</b>	2020-07-16	0.008442	2020-07-19	AUTORA
<b>selic_2</b>	2020-07-15	0.008442	2020-07-19	AUTORA
<b>selic_3</b>	2020-07-14	0.008442	2020-07-19	AUTORA
<b>selic_4</b>	2020-07-13	0.008442	2020-07-19	AUTORA

Ao especificar um índice com valor conceitual, podemos usar as funções `idxmin()` e `idxmax()` para descobrir qual o índice do menor e do maior de uma Series. Observe os exemplos a seguir.

In [13]:

```
print(df_selic['valor'].idxmin())
print(df_selic['valor'].idxmax())
```

Ver anotações

```
selic_7606
```

```
selic_7623
```

## ETAPA DE EXTRAÇÃO DE INFORMAÇÕES

Agora que já fizemos as transformações que gostaríamos, podemos passar a fase de extração de informações.

### FILTROS COM LOC

Um dos recursos mais utilizados por equipes das áreas de dados é a aplicação de filtros. Imagine a seguinte situação, uma determinada pesquisa quer saber qual é a média de idade de todas as pessoas na sua sala de aula, bem como a média de idades somente das mulheres e somente dos homens. A distinção por gênero é um filtro! Esse filtro vai possibilitar comparar a idade de todos, com a idade de cada grupo e entender se as mulheres ou homens estão abaixo ou acima da média geral.

DataFrames da biblioteca pandas possuem uma propriedade chamada **loc** (<https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.loc.html>). Essa propriedade permite acessar um conjunto de linhas (filtrar linhas), por meio do índice ou por um vetor booleano (vetor de True ou False).

Vamos começar explorando o filtro pelos índices. Entrada 14 estamos localizando (loc), o registro que possui índice 'selic\_0', como resultado obtém-se uma Series. Na entrada 15, foram filtrados três registros, para isso foi necessário passar uma lista contendo os índices, como resultado obtivemos um novo DF. Na entrada 16, fizemos um fatiamento (slice), procurando um intervalo de índices.

In [14]:

```
df_selic.loc['selic_0']
```

Out[14]:

0

Ver anotações

```
data           2020-07-17 00:00:00
valor          0.008442
data_extracao  2020-07-19 00:00:00
responsavel    AUTORA
Name: selic_0, dtype: object
```

0

In [15]:

```
df_selic.loc[['selic_0', 'selic_4', 'selic_200']]
```

Ver anotações

Out[15]:

	<b>data</b>	<b>valor</b>	<b>data_extracao</b>	<b>responsavel</b>
<b>selic_0</b>	2020-07-17	0.008442	2020-07-19	AUTORA
<b>selic_4</b>	2020-07-13	0.008442	2020-07-19	AUTORA
<b>selic_200</b>	2019-09-30	0.020872	2020-07-19	AUTORA

In [16]:

```
df_selic.loc[:'selic_5']
```

Out[16]:

	<b>data</b>	<b>valor</b>	<b>data_extracao</b>	<b>responsavel</b>
<b>selic_0</b>	2020-07-17	0.008442	2020-07-19	AUTORA
<b>selic_1</b>	2020-07-16	0.008442	2020-07-19	AUTORA
<b>selic_2</b>	2020-07-15	0.008442	2020-07-19	AUTORA
<b>selic_3</b>	2020-07-14	0.008442	2020-07-19	AUTORA
<b>selic_4</b>	2020-07-13	0.008442	2020-07-19	AUTORA
<b>selic_5</b>	2020-07-10	0.008442	2020-07-19	AUTORA

A propriedade loc, aceita um segundo argumento, que é a coluna (ou colunas) que serão exibidas para os índices escolhidos. Na entrada 17 selecionamos uma única coluna e na entrada 18 uma lista de colunas.

O mesmo resultado poderia ser alcançado passando a coluna, ou a lista de colunas conforme as sintaxes:

- df\_selic.loc[['selic\_0', 'selic\_4', 'selic\_200']]['valor']

- df\_selic.loc[['selic\_0', 'selic\_4', 'selic\_200']][['valor', 'data\_extracao']]

In [17]:

```
df_selic.loc[['selic_0', 'selic_4', 'selic_200'], 'valor']
```

0

Out[17]:

selic_0	0.008442
selic_4	0.008442
selic_200	0.020872
Name:	valor, dtype: float64

Ver anotações

In [18]:

```
df_selic.loc[['selic_0', 'selic_4', 'selic_200'], ['valor', 'data_extracao']]
```

Out[18]:

	<b>valor</b>	<b>data_extracao</b>
<b>selic_0</b>	0.008442	2020-07-19
<b>selic_4</b>	0.008442	2020-07-19
<b>selic_200</b>	0.020872	2020-07-19

Antes de vermos a criação de filtros para o loc com condições booleanas, vale mencionar que existe também a propriedade iloc, a qual filtra as linhas considerando a posição que ocupam no objeto. Veja no exemplo a seguir, estamos usando o **iloc** para filtrar os 5 primeiros registros, usando a mesma notação do fatiamento de listas. Essa propriedade não possui a opção de também selecionar colunas. Veja um exemplo a seguir.

In [19]:

```
df_selic.iloc[:5]
```

Out[19]:

	<b>data</b>	<b>valor</b>	<b>data_extracao</b>	<b>responsavel</b>
<b>selic_0</b>	2020-07-17	0.008442	2020-07-19	AUTORA
<b>selic_1</b>	2020-07-16	0.008442	2020-07-19	AUTORA

	<b>data</b>	<b>valor</b>	<b>data_extracao</b>	<b>responsavel</b>
<b>selic_2</b>	2020-07-15	0.008442	2020-07-19	AUTORA
<b>selic_3</b>	2020-07-14	0.008442	2020-07-19	AUTORA
<b>selic_4</b>	2020-07-13	0.008442	2020-07-19	AUTORA

0

Ver anotações

## FILTROS COM TESTES BOOLEANOS

Podemos usar operadores relacionais e lógicos para fazer testes condicionais com os valores das colunas de um DF. Ao criarmos um teste condicional, internamente a biblioteca testa todas as linhas do DF ou da Series, retornando uma Series booleana, ou seja, composta por valores True ou False.

Observe a partir da entrada 20 a seguir. Estamos utilizando um operador relacional, para testar se os valores da coluna 'valor', são menores que < 0.01. Armazenamos o resultado em uma variável chamada teste. Veja que o tipo do teste é uma Series, e na linha 4 teste[:5], estamos imprimindo os cinco primeiros resultados do teste lógico.

In [20]:

```
teste = df_selic['valor'] < 0.01

print(type(teste))
teste[:5]
```

```
<class 'pandas.core.series.Series'>
```

Out[20]:

```
selic_0    True
selic_1    True
selic_2    True
selic_3    True
selic_4    True
Name: valor, dtype: bool
```

No código, entrada 21 a seguir, realizamos mais um teste lógico para ver se a data da taxa é do ano de 2020. Para isso, utilizamos o método `to_datetime()` para converter a string para data e então fazer a comparação.

In [21]:

```
teste2 = df_selic['data'] >= pd.to_datetime('2020-01-01')

print(type(teste2))
teste2[:5]
```

```
<class 'pandas.core.series.Series'>
```

Out[21]:

```
selic_0    True
selic_1    True
selic_2    True
selic_3    True
selic_4    True
Name: data, dtype: bool
```

O teste condicional pode ser construído também utilizando operadores lógicos. Para a operação lógica AND (E), em pandas, usa-se o caracter `&`. Para fazer a operação lógica OR (OU), usa-se o caracter `|`. Cada teste deve estar entre parênteses, senão ocorre um erro. Observe o código a seguir, na entrada 22 e a linha seguinte, temos a construção de dois novos testes, o primeiro usando a operação AND e o segundo usando OR.

In [22]:

0

[Ver anotações](#)

```
teste3 = (df_selic['valor'] < 0.01) & (df_selic['data'] >= pd.to_datetime('2020-01-01'))  
  
teste4 = (df_selic['valor'] < 0.01) | (df_selic['data'] >= pd.to_datetime('2020-01-01'))  
  
print("Resultado do AND:\n")  
print(teste3[:3])  
  
print("Resultado do OR:\n")  
print(teste4[:3])
```

Resultado do AND:

```
selic_0    True  
selic_1    True  
selic_2    True  
dtype: bool
```

Resultado do OR:

```
selic_0    True  
selic_1    True  
selic_2    True  
dtype: bool
```

Agora que já sabemos criar as condições, basta aplicá-las no DataFrame para criar o filtro. A construção é feita passando a condição para a propriedade loc. Observe o código a seguir. Na linha 1 estamos criando a condição do filtro (que é uma Series booleana) e na entrada 23, passamos como parâmetro para filtrar os registros. Esse filtro retornou somente 4 registros.

In [23]:

```
filtro1 = df_selic['valor'] < 0.01  
  
df_selic.loc[filtro1]
```

Ver anotações

Out[23]:

	<b>data</b>	<b>valor</b>	<b>data_extracao</b>	<b>responsavel</b>
<b>selic_0</b>	2020-07-17	0.008442	2020-07-19	AUTORA
<b>selic_1</b>	2020-07-16	0.008442	2020-07-19	AUTORA
<b>selic_2</b>	2020-07-15	0.008442	2020-07-19	AUTORA
<b>selic_3</b>	2020-07-14	0.008442	2020-07-19	AUTORA
<b>selic_4</b>	2020-07-13	0.008442	2020-07-19	AUTORA
<b>selic_5</b>	2020-07-10	0.008442	2020-07-19	AUTORA
<b>selic_6</b>	2020-07-09	0.008442	2020-07-19	AUTORA
<b>selic_7</b>	2020-07-08	0.008442	2020-07-19	AUTORA
<b>selic_8</b>	2020-07-07	0.008442	2020-07-19	AUTORA
<b>selic_9</b>	2020-07-06	0.008442	2020-07-19	AUTORA
<b>selic_10</b>	2020-07-03	0.008442	2020-07-19	AUTORA
<b>selic_11</b>	2020-07-02	0.008442	2020-07-19	AUTORA
<b>selic_12</b>	2020-07-01	0.008442	2020-07-19	AUTORA
<b>selic_13</b>	2020-06-30	0.008442	2020-07-19	AUTORA
<b>selic_14</b>	2020-06-29	0.008442	2020-07-19	AUTORA
<b>selic_15</b>	2020-06-26	0.008442	2020-07-19	AUTORA
<b>selic_16</b>	2020-06-25	0.008442	2020-07-19	AUTORA
<b>selic_17</b>	2020-06-24	0.008442	2020-07-19	AUTORA
<b>selic_18</b>	2020-06-23	0.008442	2020-07-19	AUTORA
<b>selic_19</b>	2020-06-22	0.008442	2020-07-19	AUTORA
<b>selic_20</b>	2020-06-19	0.008442	2020-07-19	AUTORA
<b>selic_21</b>	2020-06-18	0.008442	2020-07-19	AUTORA
<b>selic_7606</b>	1990-03-16	0.000000	2020-07-19	AUTORA
<b>selic_7607</b>	1990-03-15	0.000000	2020-07-19	AUTORA
<b>selic_7608</b>	1990-03-14	0.000000	2020-07-19	AUTORA

0

Ver anotações

Na entrada 23, criamos primeiro a condição, guardamos na variável e depois aplicamos, mas poderíamos ter passado a condição direta:

`df_selic.loc[df_selic['valor'] < 0.01]`. Cabe ao desenvolvedor escolher a sintaxe que se sente mais a vontade. Nesse livro vamos adotar a sintaxe de criar os filtros e guardar em variáveis por questões didáticas e de legibilidade do código.

Na entrada 24 (linha 1), criamos uma condição para exibir o registro das datas apenas do mês de janeiro de 2020. Primeiro criamos duas variáveis para armazenar as datas, na linha 4 criamos o filtro e na linha 6 o aplicamos no DF, guardando o resultado (que é um DF) em um novo DF. Todo esse filtro poderia ter sido escrito em uma única linha: `df_selic.loc[(df_selic['data'] >= pd.to_datetime('2020-01-01')) & (df_selic['data'] <= pd.to_datetime('2020-01-31'))]`, mas veja como fica mais difícil ler e interpretar o filtro, ainda mais para quem for dar manutenção no código.

In [24]:

```
data1 = pd.to_datetime('2020-01-01')
data2 = pd.to_datetime('2020-01-31')

filtro_janeiro_2020 = (df_selic['data'] >= data1) & (df_selic['data'] <= data2)

df_janeiro_2020 = df_selic.loc[filtro_janeiro_2020]
df_janeiro_2020.head()
```

Out[24]:

	<b>data</b>	<b>valor</b>	<b>data_extracao</b>	<b>responsavel</b>
<b>selic_114</b>	2020-01-31	0.017089	2020-07-19	AUTORA
<b>selic_115</b>	2020-01-30	0.017089	2020-07-19	AUTORA
<b>selic_116</b>	2020-01-29	0.017089	2020-07-19	AUTORA
<b>selic_117</b>	2020-01-28	0.017089	2020-07-19	AUTORA
<b>selic_118</b>	2020-01-27	0.017089	2020-07-19	AUTORA

Ver anotações

Vamos criar mais um filtro e um novo DF para que possamos ver a importância dos filtros. No código a seguir, vamos criar um novo DF contendo as informações do mês de janeiro do ano de 2019.

In [25]:

```
data1 = pd.to_datetime('2019-01-01')
data2 = pd.to_datetime('2019-01-31')

filtro_janeiro_2019 = (df_selic['data'] >= data1) & (df_selic['data'] <= data2)

df_janeiro_2019 = df_selic.loc[filtro_janeiro_2019]
df_janeiro_2019.head()
```

Out[25]:

	<b>data</b>	<b>valor</b>	<b>data_extracao</b>	<b>responsavel</b>
<b>selic_367</b>	2019-01-31	0.02462	2020-07-19	AUTORA
<b>selic_368</b>	2019-01-30	0.02462	2020-07-19	AUTORA
<b>selic_369</b>	2019-01-29	0.02462	2020-07-19	AUTORA
<b>selic_370</b>	2019-01-28	0.02462	2020-07-19	AUTORA
<b>selic_371</b>	2019-01-25	0.02462	2020-07-19	AUTORA

Agora que temos três DFs, um completo e dois com filtros vamos utilizar métodos da estatística descritiva para extrair informações sobre o valor da taxa selic em cada DF. Queremos saber, qual o valor máximo e mínimo registrado em cada caso e qual a média.

Ao selecionar uma coluna, temos uma Series, então basta utilizar o método solicitado, conforme códigos a seguir.

In [26]:

0

Ver anotações

```
print('Mínimo geral = ', df_selic['valor'].min())
print('Mínimo janeiro de 2019 = ', df_janeiro_2019['valor'].min())
print('Mínimo janeiro de 2020 = ', df_janeiro_2020['valor'].min(), '\n')
```

```
0
print('Máximo geral = ', df_selic['valor'].max())
print('Máximo janeiro de 2019 = ', df_janeiro_2019['valor'].max())
print('Máximo janeiro de 2020 = ', df_janeiro_2020['valor'].max(), '\n')
```

```
Ver anotações
print('Média geral = ', df_selic['valor'].mean())
print('Média janeiro de 2019 = ', df_janeiro_2019['valor'].mean())
print('Média janeiro de 2020 = ', df_janeiro_2020['valor'].mean(), '\n')
```

Mínimo geral = 0.0

Mínimo janeiro de 2019 = 0.02462

Mínimo janeiro de 2020 = 0.017089

Máximo geral = 3.626

Máximo janeiro de 2019 = 0.02462

Máximo janeiro de 2020 = 0.017089

Média geral = 0.2863543465855944

Média janeiro de 2019 = 0.02461999999999999

Média janeiro de 2020 = 0.017089000000000003

Veja como os filtros permitem começar a tirar respostas para áreas de negócios.

No ano de 2019 tanto a mínima quanto a máxima foram superiores que no ano de 2020. A máxima geral é bem superior a máxima desses meses, assim como a média geral, que é bem superior, ou seja, nesses meses a taxa média foi inferior a média geral, sendo que em janeiro de 2020 foi ainda pior.

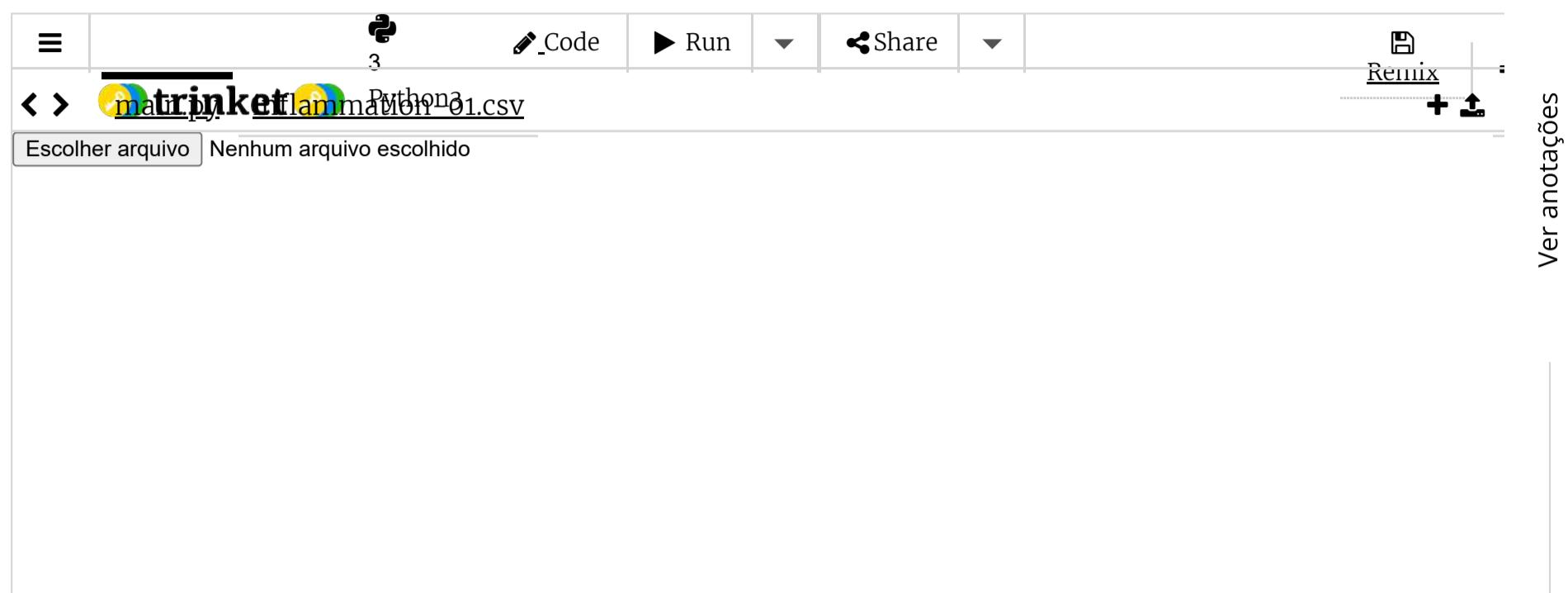
Todas essas transformações que fizemos podem ser persistidas fisicamente em um arquivo, por exemplo, csv. Veja o código a seguir, que salva um arquivo chamado dados\_selic.csv, na pasta de trabalho, com os parâmetros padrão:

~~separado por vírgula, com cabeçalho, com a inclusão dos índices, dentre outros.~~

In [27]:

```
df_selic.to_csv('dados_selic.csv')
```

Que tal utilizar o simulador a seguir e testar as funcionalidades aprendidas?



Poderíamos extrair diversas outras informações dos dados. Todo esse trabalho faz parte do cotidiano nos engenheiros, cientistas e analistas de dados. Os engenheiros de dados mais focados na preparação e disponibilização dos dados, os cientistas focados em responder questões de negócio, inclusive utilizando modelos de machine learning e deep learning e os analistas, também respondendo a perguntas de negócios e apresentando resultados. Se você gostou do que fizemos quem sabe não deva investir mais nessa área?!

#### SAIBA MAIS

Existem diversos portais que disponibilizam dados, por exemplo, o próprio kaggle, os portais brasileiros <https://www.portaldatransparencia.gov.br/> e <https://www.dados.gov.br/dataset>, o portal <https://archive.ics.uci.edu/ml/datasets.php>, ou o <https://vincentarelbundock.github.io/Rdatasets/datasets.html>. Enfim, são inúmeros os repositórios que podem ser explorados, até mesmo o git hub.

## BANCO DE DADOS COM PANDAS

Para finalizar nossa aula vamos aprender que além dos métodos para acessar arquivos, a biblioteca pandas possui dois métodos que permitem executar instruções SQL em banco de dados. Os métodos são:

- `pandas.read_sql(sql, con, index_col=None, coerce_float=True, params=None, parse_dates=None, columns=None, chunksize=None)`
- `pandas.read_sql_query(sql, con, index_col=None, coerce_float=True, params=None, parse_dates=None, chunksize=None)`

O mínimo de parâmetros que ambos métodos exigem é a instrução SQL e uma conexão com um banco de dados (con). A conexão com o banco de dados, deve ser feita usando uma outra biblioteca, por exemplo, sqlalchemy (suporte para diversos bancos), pyodbc para SQL Server, cx\_Oracle para Oracle, psycopg2 para Postgresql, dentre outras. Seguindo as recomendações da PEP 249 todas bibliotecas precisam fornecer um método "connect", o qual recebe a string de conexão. A sintaxe da string de conexão depende da biblioteca e do banco de dados.

A seguir apresentamos de maneira didática dois exemplos, o primeiro uma conexão com um banco postgresql e outro com um banco mysql. Veja que a única diferença entre eles é a importação da biblioteca específica e a string de conexão. Dessa forma, ao estabelecer conexão com um banco de dados e armazenar a instância em uma variável, basta passá-la como parâmetro do método da biblioteca pandas.

```
import psycopg2
```

```
host = 'XXXXXX'  
port = 'XXXXXX'  
database = 'XXXXXX'  
username = 'XXXXXX'  
password = 'XXXXXX'
```

```
conn_str = fr"dbname='{database}' user='{username}' host='{host}'  
password='{password}' port='{port}'"  
conn = psycopg2.connect(conn_str)
```

0

```
query = "select * from XXX.YYYY"  
df = pd.read_sql(query, conn)import mysql.connector
```

Ver anotações

```
host = 'XXXXXX'  
port = 'XXXXXX'  
database = 'XXXXXX'  
username = 'XXXXXX'  
password = 'XXXXXX'
```

```
conn_str = fr"host={host}, user={username}, passwd={password}, database=  
{database}"  
conn = mysql.connector.connect(host="localhost", user="root", passwd="",  
database="bd")
```

```
query = "select * from XXX.YYYY"  
df = pd.read_sql(query, conn)
```

## REFERÊNCIAS E LINKS ÚTEIS

Kaggle. Titanic: Machine Learning from Disaster. Disponível em  
(<https://www.kaggle.com/c/titanic>). Acesso em: 20 jun. 2020.

PyPI. Python Package Index. Disponível em: <https://pypi.org/>. Acesso em: 17 jun.  
2020.

pandas Team. DataFrame. Disponível em: <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html>. Acesso em: 17 jun. 2020.

Pandas Team. DataFrame. Disponível em: <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html>. Acesso em: 17 jun. 2020.

Pandas Team. pandas documentation. Disponível em:  
<https://pandas.pydata.org/pandas-docs/stable/index.html>. Acesso em: 17 jun.  
2020.

Pandas Team. Series. Disponível em: <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.html>. Acesso em: 17 jun. 2020.

0

Ver anotações

## FOCO NO MERCADO DE TRABALHO

# INTRODUÇÃO A MANIPULAÇÃO DE DADOS EM PANDAS

Vanessa Cadan Scheffer

### TRANSFORMAÇÃO DOS DADOS E EXTRAÇÃO DE INFORMAÇÕES

A biblioteca pandas possui métodos capazes de fazer a leitura dos dados e o carregamento em um DataFrame, além de recursos como a aplicação de filtros.



Fonte: Shutterstock.

#### Deseja ouvir este material?

Áudio disponível no material digital.

### DESAFIO

Como desenvolvedor em uma empresa de consultoria de software, você foi alocado em um projeto para uma empresa de geração de energia. Essa empresa tem interesse em criar uma solução que acompanhe as exportações de etanol no Brasil. Esse tipo de informação está disponível no site do governo brasileiro <http://www.dados.gov.br/dataset>, em formatos CSV, JSON, dentre outros.

No endereço <http://www.dados.gov.br/dataset/importacoes-e-exportacoes-de-etanol> é possível encontrar várias bases de dados (datasets), contendo informações de importação e exportação de etanol. O cliente está interessado em obter informações sobre a Exportação Etano Hidratado (barris equivalentes de petróleo) 2012-2020, cujo endereço é <http://www.dados.gov.br/dataset/importacoes-e-exportacoes-de-etanol/resource/ca6a2afe-def5-4986-babc-b5e9875d39a5>. Para a análise será necessário fazer o download do arquivo.

O cliente deseja uma solução que extraia as seguintes informações:

- Em cada ano, qual o menor e o maior valor arrecadado da exportação?
- Considerando o período de 2012 a 2019, qual a média mensal de arrecadamento com a exportação?

0

Ver anotações

- Considerando o período de 2012 a 2019, qual ano teve o menor arrecadamento? E o menor?

Como parte das informações técnicas sobre o arquivo, foi lhe informado que se trata de um arquivo delimitado CSV, cujo separador de campos é ponto-e-vírgula e a codificação do arquivo está em ISO-8859-1. Como podemos obter o arquivo? Como podemos extrair essas informações usando a linguagem Python? Serão necessários transformações nos dados para obtermos as informações solicitadas?

0

Ver anotações

## RESOLUÇÃO

Para começar a resolver o desafio, precisamos fazer o download do arquivo com os dados. Podemos acessar o endereço

<http://www.dados.gov.br/dataset/importacoes-e-exportacoes-de-etanol/resource/ca6a2afe-def5-4986-babc-b5e9875d39a5> e clicar no botão "ir para recurso" ou então digitar o endereço

<http://www.anp.gov.br/arquivos/dadosabertos/iee/exportacao-etanol-hidratado-2012-2020-bep.csv> que fará o download do arquivo de modo automático. Após obter o arquivo, basta copiá-lo para a pasta do projeto.

Conforme orientações, o arquivo é delimitado, mas seu separador padrão é o ";" e a codificação do arquivo foi feita em ISO-8859-1. Portanto, teremos que passar esses dois parâmetros para a leitura do arquivo usando a biblioteca pandas, uma vez que o delimitar padrão da biblioteca é o ",". No código a seguir, estamos fazendo a importação dos dados. Veja que temos 9 linhas e 8 colunas.

In [28]:

```
import pandas as pd

df_etanol = pd.read_csv('exportacao-etanol-hidratado-2012-2020-bep.csv',
sep=';', encoding="ISO-8859-1")

print(df_etanol.info())
df_etanol.head(2)
```

```
<class 'pandas.core.frame.DataFrame'>

RangeIndex: 9 entries, 0 to 8
Data columns (total 17 columns):
ANO           9 non-null int64
PRODUTO        9 non-null object
MOVIMENTO COMERCIAL 9 non-null object
UNIDADE        9 non-null object
JAN           9 non-null object
FEV           9 non-null object
MAR           9 non-null object
ABR           9 non-null object
MAI           8 non-null object
JUN           8 non-null object
JUL           8 non-null object
AGO           8 non-null object
SET           8 non-null object
OUT           8 non-null object
NOV           8 non-null object
DEZ           8 non-null object
TOTAL          9 non-null object
dtypes: int64(1), object(16)
memory usage: 1.3+ KB
None
```

0

Ver anotações

Out[28]:

	<b>ANO</b>	<b>PRODUTO</b>	<b>MOVIMENTO COMERCIAL</b>	<b>UNIDADE</b>	JAN	FEV	MAR	ABR	MAI
0	2012	ETANOL HIDRATADO (bep)	EXPORTACAO	bep	87231,41132	141513,5186	122157,3385	98004,42926	153286,6078
1	2013	ETANOL HIDRATADO (bep)	EXPORTACAO	bep	673419,9767	387331,6487	96929,59201	54390,05046	115092,482

Agora que temos os dados, vamos dividir nossa solução em duas etapas: a de transformação dos dados e a de extração de informações.

## ■ ETAPA DE TRANSFORMAÇÕES

Vamos começar removendo as colunas que sabemos que não serão utilizadas, afinal, quanto menos dados na memória RAM, melhor. Veja no código a seguir a remoção de três colunas, com o parâmetro `inplace=True`, fazendo com que a transformação seja salva no próprio objeto.

In [29]:

```
df_etanol.drop(columns=['PRODUTO', 'MOVIMENTO COMERCIAL', 'UNIDADE'],
inplace=True)

df_etanol.head(2)
```

Out[29]:

ANO	JAN	FEV	MAR	ABR	MAI	JUN	JUL	AGO
0	201287231,41132	141513,5186	122157,3385	98004,42926	153286,6078	144373,6894	384743,6142	244861,0289
1	2013673419,9767	387331,6487	96929,59201	54390,05046	115092,482	387498,3792	339162,21	354343,2858

Agora vamos redefinir os índices do DF, usando a coluna ANO. Esse passo será importante para a fase de extração de informações. Veja que também optamos em remover a coluna do DF (drop=True).

Ver anotações

In [30]:

```
df_etanol.set_index(keys='ANO', drop=True, inplace=True)

df_etanol.head(2)
```

Out[30]:

	JAN	FEV	MAR	ABR	MAI	JUN	JUL	AGO
ANO								
2012	87231,41132	141513,5186	122157,3385	98004,42926	153286,6078	144373,6894	384743,6142	244861,0289
2013	673419,9767	387331,6487	96929,59201	54390,05046	115092,482	387498,3792	339162,21	354343,2858

Como os dados são de origem brasileira, a vírgula é usada como separador decimal, o que não condiz com o padrão da biblioteca pandas. Precisamos converter todas as vírgulas em ponto. Para isso vamos utilizar uma estrutura de repetição que filtra cada coluna, criando uma Series, o que nos habilita a utilizar a funcionalidade str.replace(',', '.') para a substituição.

In [31]:

```
for mes in 'JAN FEV MAR ABR MAI JUN JUL AGO SET OUT NOV DEZ TOTAL'.split():
    df_etanol[mes] = df_etanol[mes].str.replace(',', '.')

print(df_etanol.dtypes)
df_etanol.head(2)
```

JAN	object
FEV	object
MAR	object
ABR	object
MAI	object
JUN	object
JUL	object
AGO	object
SET	object
OUT	object
NOV	object
DEZ	object
TOTAL	object
dtype:	object

Out[31]:

	JAN	FEV	MAR	ABR	MAI	JUN	JUL	AGC
ANO								
2012	87231.41132	141513.5186	122157.3385	98004.42926	153286.6078	144373.6894	384743.6142	244861.0289
2013	673419.97670	387331.6487	96929.59201	54390.05046	115092.4820	387498.3792	339162.2100	354343.2858

Mesmo trocando a vírgula por ponto, a biblioteca ainda não conseguiu identificar como ponto flutuante. Portanto, vamos fazer a conversão usando o método `astype(float)`.

In [32]:

```
df_etanol = df_etanol.astype(float)
print(df_etanol.dtypes)

df_etanol.head(2)
```

JAN	float64
FEV	float64
MAR	float64
ABR	float64
MAI	float64
JUN	float64
JUL	float64
AGO	float64
SET	float64
OUT	float64
NOV	float64
DEZ	float64
TOTAL	float64
	dtype: object

Out[32]:

	JAN	FEV	MAR	ABR	MAI	JUN	JUL	AGC
ANO								
2012	87231.41132	141513.5186	122157.33850	98004.42926	153286.6078	144373.6894	384743.6142	244861.0289
2013	673419.97670	387331.6487	96929.59201	54390.05046	115092.4820	387498.3792	339162.2100	354343.2858

Ver anotações

**PESQUISE MAIS**

Poderíamos ter usado a biblioteca locale para fazer parte desse trabalho, que tal se aprofundar e pesquisar mais?!

0

**| ETAPA DE EXTRAÇÃO DE INFORMAÇÕES**

Ver anotações

Agora que preparamos os dados, podemos começar a etapa de extração das informações solicitadas. Vamos começar extraiendo o menor e maior valor arrecadado em cada ano. Como nosso índice é o próprio ano, podemos usar a função loc para filtrar e então os métodos min() e max(). Para que a extração seja feita para todos os anos, usamos uma estrutura de repetição.

Nas linhas print(f"Menor valor = {minimo:.0f}".replace(',', '.')) print(f"Maior valor = {maximo:.0f}".replace(',', '.')) do código a seguir, estamos fazendo a impressão dos valores solicitados. Para que fique mais claro a leitura, formatamos a exibição. O código minimo:.0f faz com que seja exibida somente a parte inteira e o separador de milhar seja feito por vírgula. Em seguida substituimos a vírgula por ponto que é o padrão brasileiro.

In [33]:

```
# Em cada ano, qual o menor e o maior valor arrecadado da exportação?

for ano in range(2012, 2021):
    ano_info = df_etanol.loc[ano]
    minimo = ano_info.min()
    maximo = ano_info.max()
    print(f"Ano = {ano}")
    print(f"Menor valor = {minimo:.0f}".replace(',', '.'))
    print(f"Maior valor = {maximo:.0f}".replace(',', '.'))
    print("-----")
```

```
Ano = 2012
Menor valor = 87.231
Maior valor = 4.078.157
-----
Ano = 2013
Menor valor = 54.390
Maior valor = 4.168.543
-----
Ano = 2014
Menor valor = 74.303
Maior valor = 2.406.110
-----
Ano = 2015
Menor valor = 31.641
Maior valor = 3.140.140
-----
Ano = 2016
Menor valor = 75.274
Maior valor = 3.394.362
-----
Ano = 2017
Menor valor = 2.664
Maior valor = 1.337.427
-----
Ano = 2018
Menor valor = 4.249
Maior valor = 2.309.985
-----
Ano = 2019
Menor valor = 14.902
Maior valor = 2.316.773
-----
Ano = 2020
Menor valor = 83.838
Maior valor = 298.194
-----
```

0

Ver anotações

Agora, vamos implementar o código para extrair a média mensal, considerando o período de 2012 a 2019. Novamente, podemos usar o loc para filtrar os anos requisitados e, para cada coluna, extrair a média. Na linha 5 fazemos a extração, mas veja que está dentro de uma estrutura de repetição, mês a mês. Na linha 6 fazemos a impressão do resultado, também formatando a saída. Veja que o mês de abril apresenta um rendimento bem inferior aos demais!

In [34]:

```
# Considerando o período de 2012 a 2019, qual a média mensal de arrecadamento
# com a exportação

print("Média mensal de rendimentos:")

for mes in 'JAN FEV MAR ABR MAI JUN JUL AGO SET OUT NOV DEZ'.split():
    media = df_etanol.loc[2012:2019, mes].mean()

    print(f"{mes} = {media:.0f}".replace(',', '.'))
```

Média mensal de rendimentos:

JAN = 248.380  
FEV = 210.858  
MAR = 135.155  
ABR = 58.929  
MAI = 106.013  
JUN = 244.645  
JUL = 295.802  
AGO = 276.539  
SET = 354.454  
OUT = 376.826  
NOV = 266.748  
DEZ = 319.588

0

Ver anotações

Agora precisamos descobrir qual ano teve a menor e a maior quantia em exportação, considerando o período de 2012 a 2019. Para isso vamos usar o método `idxmin()` para descobrir o mínimo e `idxmax()` para o máximo.

In [35]:

```
# Considerando o período de 2012 a 2019, qual ano teve o menor arrecadamento? E
# o menor?

ano_menor_arrecadacao = df_etanol.loc[2012:2019, 'TOTAL'].idxmin()
ano_maior_arrecadacao = df_etanol.loc[2012:2019, 'TOTAL'].idxmax()

print(f"Ano com menor arrecadação = {ano_menor_arrecadacao}")
print(f"Ano com maior arrecadação = {ano_maior_arrecadacao}")
```

```
Ano com menor arrecadação = 2017
Ano com maior arrecadação = 2013
```

Agora é com você, que tal agora organizar as códigos em funções e deixar a solução pronta para ser usada pela equipe?!

## DESAFIO DA INTERNET

Ganhar habilidade em programação exige estudo e treino (muito treino). Acesse o endereço <https://www.kaggle.com/datasets>, faço seu cadastro e escolha uma base de dados para treinar e desenvolver seu conhecimento com a biblioteca pandas.

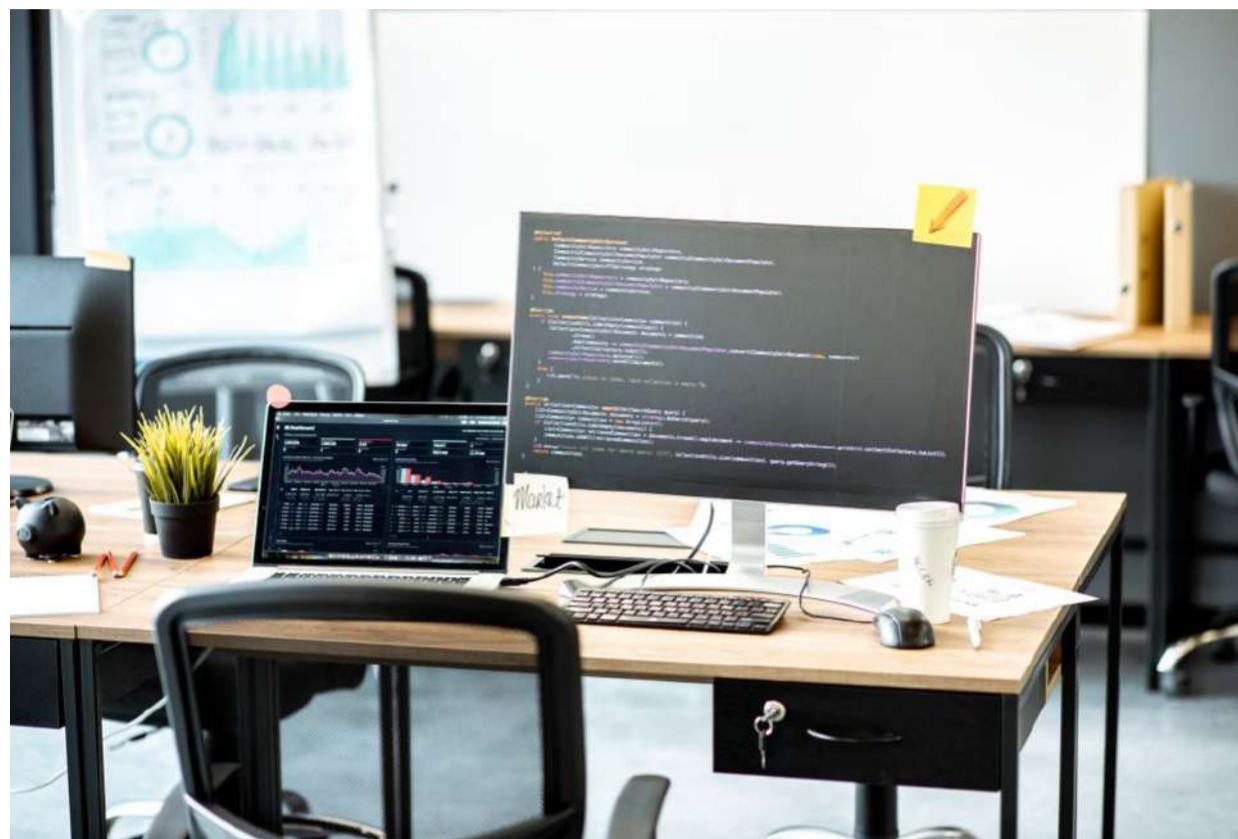
# NÃO PODE FALTAR

## VISUALIZAÇÃO DE DADOS EM PYTHON

Vanessa Cadan Scheffer

### BIBLIOTECAS E FUNÇÕES PARA CRIAÇÃO DE GRÁFICOS

Para a criação de gráficos em Python são utilizadas as bibliotecas matplotlib e outras baseadas na matplotlib, e também funções que permitem criar e personalizar os gráficos.



Fonte: Shutterstock.

### Deseja ouvir este material?

Áudio disponível no material digital.

### INTRODUÇÃO A VISUALIZAÇÃO DE DADOS EM PYTHON

Visualização de dados ou DataViz é um dos pilares dos profissionais que trabalham com dados. Existem profissionais que se especializam e inclusive tiram certificações para atuar nessa área específica. Após uma análise de dados e extração de informações é interessante que a entrega de resultados para a área de negócios seja feita de maneira visual, ou seja, por meio de gráficos. Um gráfico bem elaborado "fala" por si só e ajuda aos que assistem a entenderem os resultados.

A linguagem Python, conta com uma série de bibliotecas que permitem a criação de gráficos, os quais podem ser estáticos (sem iteração) ou dinâmicos, que apresentam iteração, como por exemplo, responder a eventos de clique do mouse. Nessa aula, nossa objetivo é apresentar um pouco desse universo de possibilidades.

0

Ver anotações

## MATPLOTLIB

Ao se falar em criação de gráficos em Python, o profissional precisa conhecer a biblioteca **matplotlib**, pois diversas outras são construídas a partir desta. A criação e grande parte da evolução dessa biblioteca se deve a John Hunter, que a desenvolveu como uma opção ao uso dos softwares gnuplot e MATLAB (MCGREGGOR, 2015). Com a utilização da linguagem Python na área científica para trabalhar com dados, após a extração dos resultados, o cientista precisava criar seus gráficos nos outros softwares mencionados, o que se tornava incoveniente, motivando a criação da biblioteca em Python.

A instalação da biblioteca pode ser feita via pip install: `pip install matplotlib`, lembrando que em ambientes como o projeto Anaconda e o Colab esse recurso já está disponível. O módulo **pyplot** possui uma coleção de funções que permitem criar e personalizar os gráficos

([https://matplotlib.org/api/\\_as\\_gen/matplotlib.pyplot.html](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.html)). Existem duas sintaxes que são amplamente adotadas para importar essa biblioteca para o projeto:

- `import matplotlib.pyplot as plt`
- `from matplotlib import pyplot as plt`

Em ambas formas de importação utilizamos o apelido "plt" que é uma convenção adotada para facilitar o uso das funções. Ao trabalharmos no jupyter notebook com o kernel do IPython (o kernel do IPython é o backend de execução do Python para o Jupyter), podemos habilitar uma opção de fazer a impressão do gráfico "inline", ou seja, no próprio notebook. Essa opção faz parte do "Built-in magic commands", cuja documentação pode ser acessada no endereço <https://ipython.readthedocs.io/en/stable/interactive/magics.html>. Para habilitar utiliza-se a sintaxe: `%matplotlib inline`. Portanto, projetos no jupyter notebook, que utilizem o matplotlib sempre terão no começo, os comandos a seguir.

In [1]:

```
from matplotlib import pyplot as plt

%matplotlib inline
```

Os gráficos são uma forma visual de "contar" a história dos dados. Em breve contaremos várias histórias, mas como nosso primeiro gráfico, que tal um pouco de arte abstrata? Vamos criar duas listas aleatórias de valores inteiros com o módulo random e então plotar um gráfico de linhas, com a função `plt.plot()` do módulo pyplot. Veja a seguir, após criar duas listas com valores aleatórios, a função `plot()` as recebe como parâmetros, utilizando-as para os valores dos eixos horizontal (x) e vertical (y) e já cria o gráfico. Mágico não? Mas como "ele" sabia que era para criar um gráfico de linhas? Por que colocou a cor azul? Por que fez desse tamanho? Todos esses parâmetros e muitos outros, podem ser configurados!

In [2]:

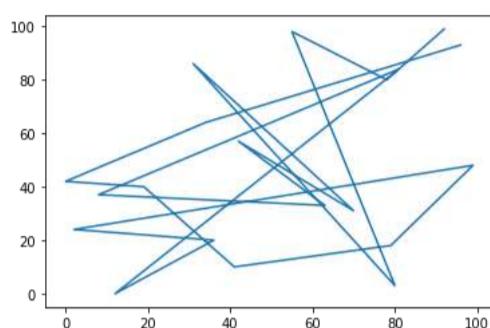
```
import random

dados1 = random.sample(range(100), k=20)
dados2 = random.sample(range(100), k=20)

plt.plot(dados1, dados2) # pyplot gerencia a figura e o eixo
```

Out[2]:

[&lt;matplotlib.lines.Line2D at 0x1cf32995940&gt;]



Existem essencialmente duas maneiras de usar o Matplotlib:

1. Deixar para o pyplot criar e gerenciar automaticamente figuras e eixos, e usar as funções do pyplot para plotagem.
2. Criar explicitamente figuras e eixos e chamar métodos sobre eles (o "estilo orientado a objetos (OO)").

No gráfico que criamos, nós utilizamos a opção 1, ou seja, foi o próprio módulo que criou o ambiente da figura e do eixo. Ao utilizar a segunda opção, podemos criar uma figura com ou sem eixos, com a função `plt.subplots()`, que quando invocada sem parâmetros, cria um layout de figura com 1 linha e 1 coluna.

## FIGURA COM EIXO COMO VARIÁVEL

Vamos explorar o estilo orientado a objetos, começando pela criação de eixos de forma explícita, ou seja com atribuição a uma variável. Vamos criar uma figura com 1 linha 2 duas colunas, ou seja, teremos dois eixos. Pense nos eixos como uma matriz, na qual cada eixo é uma posição que pode ter uma figura alocada. Vale ressaltar que sobre um eixo (sobre uma figura), podem ser plotados diversos gráficos. Para criar essa estrutura usamos a sintaxe: `fig, ax = plt.subplots(1, 2)`, onde `fig` e `ax` são os nomes das variáveis escolhidas. A variável `ax`, é do tipo `array numpy`, ou seja, os eixos nada mais são, que uma matriz de contêiners para se criar os plots. Como a figura possui dois eixos, temos que especificar em qual vamos plotar, para isso informamos qual contêiner vamos usar: `ax[0]` ou `ax[1]`.

Veja o código a seguir. Na linha 6 criamos a figura, com 1 linha 2 colunas e o eixo que vamos utilizar e ainda definimos o tamanho das figuras por meio do parâmetro `figsize`. Nas linhas 8, 9 e 10, imprimimos algumas informações para entendermos esse mecanismo do pyplot. `ax` é do tipo '`numpy.ndarray`', como já havíamos mencionado. Ao imprimir o conteúdo de `ax[0]` e `ax[1]`, podemos ver as coordenadas alocadas para realizar a impressão da figuras. Das linhas 12 a 18, imprimimos 3 gráficos sobre o primeiro eixo (que será posicionado uma figura do

lado esquerdo), definimos os rótulos dos eixos, o título do gráfico e pedimos para exibir a legenda, que é construída a partir do parâmetro "label" na função plot(). Das linhas 20 a 26, criamos novos 3 gráficos, mas agora sobre o segundo eixo (que será posicionado uma figura do lado direito). Nesse novo conjunto de gráficos, configuraremos a aparência das linhas, com os parâmetros 'r--' (red tracejado), 'b--' (blue tracejado) e 'g--' (green tracejado). Observe o resultado dos gráficos. Ao criar uma estrutura com 1 linha e 2 colunas, os gráficos ficam posicionados lado a lado, e se tivéssemos criado com 2 linhas e 1 coluna?

o

Ver anotações

In [3]:

```
import numpy as np

x = range(5)
x = np.array(x) # temos que converter para um array numpy, senão o plot não
consegue fazer operações.

fig, ax = plt.subplots(1, 2, figsize=(12, 5)) # Cria figura com subplots: 1
linha, 2 colunas e eixos

print("Tipo de ax = ", type(ax))
print("Conteúdo de ax[0] = ", ax[0])
print("Conteúdo de ax[1] = ", ax[1])

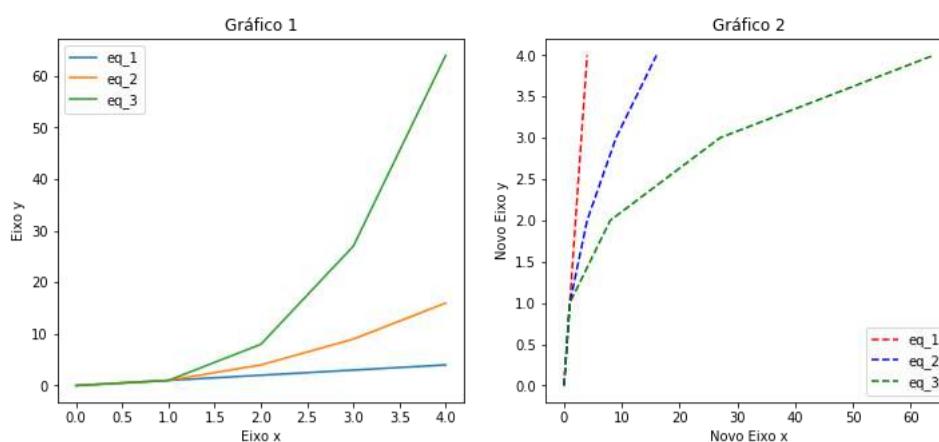
ax[0].plot(x, x, label='eq_1') # cria gráfico sobre eixo 0
ax[0].plot(x, x**2, label='eq_2') # cria gráfico sobre eixo 0
ax[0].plot(x, x**3, label='eq_3') # cria gráfico sobre eixo 0
ax[0].set_xlabel('Eixo x')
ax[0].set_ylabel('Eixo y')
ax[0].set_title("Gráfico 1")
ax[0].legend()

ax[1].plot(x, x, 'r--', label='eq_1') # cria gráfico sobre eixo 1
ax[1].plot(x**2, x, 'b--', label='eq_2') # cria gráfico sobre eixo 1
ax[1].plot(x**3, x, 'g--', label='eq_3') # cria gráfico sobre eixo 1
ax[1].set_xlabel('Novo Eixo x')
ax[1].set_ylabel('Novo Eixo y')
ax[1].set_title("Gráfico 2")
ax[1].legend()
```

```
Tipo de ax = <class 'numpy.ndarray'>
Conteúdo de ax[0] = AxesSubplot(0.125,0.125;0.352273x0.755)
Conteúdo de ax[1] = AxesSubplot(0.547727,0.125;0.352273x0.755)
```

Out[3]:

```
<matplotlib.legend.Legend at 0x1cf32a6cb00>
```



**FIGURA SEM EIXO COMO VARIÁVEL**

Também podemos criar uma figura, sem atribuir o eixo a uma variável. Nesse caso, temos que usar a função `plt.subplots(n_rows, n_cols, plot_number)`, para definir onde será plotado o gráfico. Veja no código a seguir. Na linha 4 criamos uma figura, mas agora sem eixo e sem especificar o grid. Na linha 5, estamos adicionando um subplot com 1 linha, 2 colunas e o primeiro gráfico (121). O primeiro parâmetro do método `subplot()` é a quantidade de linhas; o segundo parâmetro é a quantidade de colunas; o terceiro é número do plot dentro da figura, deve começar em 1 e ir até a quantidade de plots que se tem. Como o eixo não está atribuído a nenhuma variável, agora usamos o próprio para acessar a função `plot()`. Veja que na linha 14 estamos adicionando um subplot de 1 linha, 2 colunas a mesma figura, mas agora especificando que plotaremos a segunda figura (122).

In [4]:

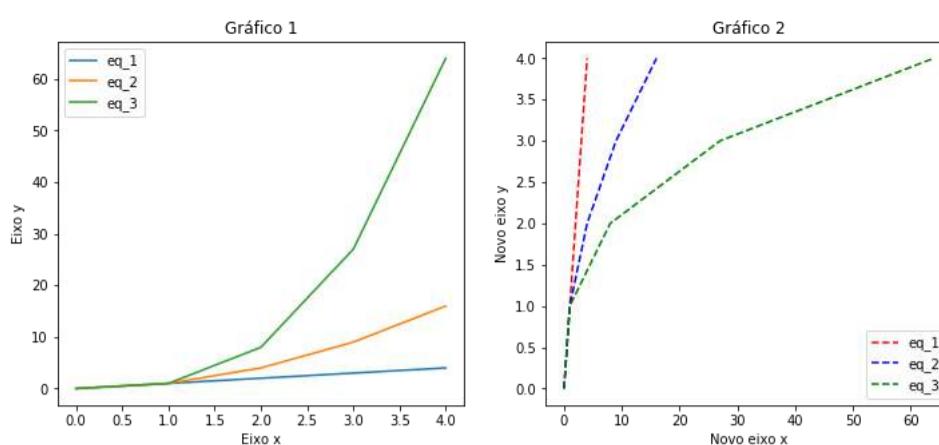
```
x = range(5)
x = np.array(x) # temos que converter para um array numpy, senão o plot não consegue fazer operações.

fig = plt.subplots(figsize=(12, 5)) # Cria figura sem eixo
plt.subplot(121) # Adiciona um grid de subplots a figura: 1 linha, 2 colunas -
Figura 1
plt.plot(x, x, label='eq_1')
plt.plot(x, x**2, label='eq_2')
plt.plot(x, x**3, label='eq_3')
plt.title("Gráfico 1")
plt.xlabel('Eixo x')
plt.ylabel('Eixo y')
plt.legend()

plt.subplot(122) # Adiciona um grid de subplots a figura: 1 linha, 2 colunas -
Figura 2
plt.plot(x, x, 'r--', label='eq_1')
plt.plot(x**2, x, 'b--', label='eq_2')
plt.plot(x**3, x, 'g--', label='eq_3')
plt.title("Gráfico 2")
plt.xlabel('Novo eixo x')
plt.ylabel('Novo eixo y')
plt.legend()
```

Out[4]:

```
<matplotlib.legend.Legend at 0x1cf32b44c50>
```



0

Ver anotações

Naturalmente obtivemos o mesmo resultado anterior, pois criamos a mesma estrutura com uma sintaxe diferente. Ao optar em utilizar eixos como variáveis ou não, o desenvolvedor deve ficar atento somente as regras de sintaxe e as funções disponíveis para cada opção. Podemos então resumir que:

- `plt.subplots()` é usado para criar um layout de figura e subplots.  
([https://matplotlib.org/api/\\_as\\_gen/matplotlib.pyplot.subplots.html#matplotlib.pyplot.subplots](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.subplots.html#matplotlib.pyplot.subplots)).
- `plt.subplot()` é usado para adicionar um subplot em um figura existente.  
([https://matplotlib.org/api/\\_as\\_gen/matplotlib.pyplot.subplot.html#matplotlib.pyplot.subplot](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.subplot.html#matplotlib.pyplot.subplot)).

Que tal treinar e explorar utilizando o simulador a seguir. Veja os exemplos que a própria ferramenta utiliza em sua página principal.

## ■ BIBLIOTECA PANDAS

As principais estruturas de dados da biblioteca pandas (Series e DataFrame) possuem o método `plot()`, construído com base no matplotlib e que permite criar gráficos a partir dos dados nas estruturas. Vamos começar criando um DataFrame a partir de um dicionário, com a quantidade de alunos em três turmas distintas.

In [5]:

```
import pandas as pd

dados = {
    'turma':['A', 'B', 'C'],
    'qtde_alunos':[33, 50, 45]
}
df = pd.DataFrame(dados)

df
```

Out[5]:

	<b>turma</b>	<b>qtde_alunos</b>
<b>0</b>	A	33
<b>1</b>	B	50
<b>2</b>	C	45

A partir de um DataFrame, podemos invocar o método: `df.plot(*args, **kwargs)` para criar os gráficos. Os argumentos dessa função, podem variar, mas existem três que são triviais: os nomes das colunas com os dados para os eixos x e y, bem como o tipo de gráfico (kind). Veja o código a seguir, os valores da coluna 'turma' serão usados no eixo x, da coluna 'qtde\_alunos' no eixo y e o tipo de gráfico será o de barras (bar). Nas entradas 7 e 8, repetimos a mesma construção, entretanto mudando o tipo de gráfico para barra na horizontal (barh) e linha (line).

o

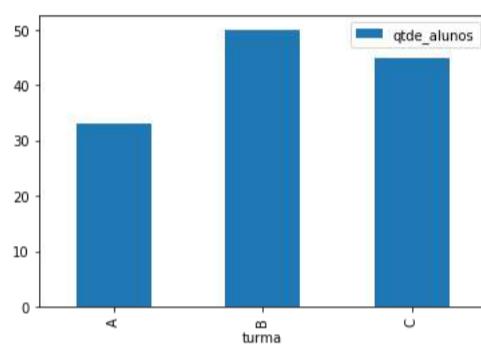
Ver anotações

In [6]:

```
df.plot(x='turma', y='qtde_alunos', kind='bar')
```

Out[6]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1cf339fe7b8>
```

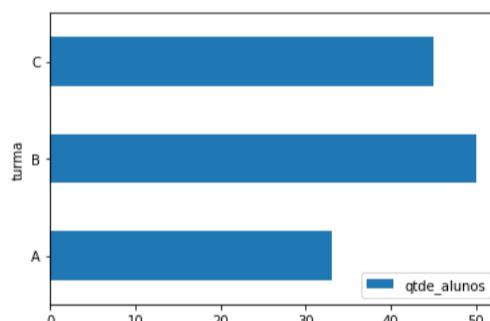


In [7]:

```
df.plot(x='turma', y='qtde_alunos', kind='barh')
```

Out[7]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1cf32bd2470>
```

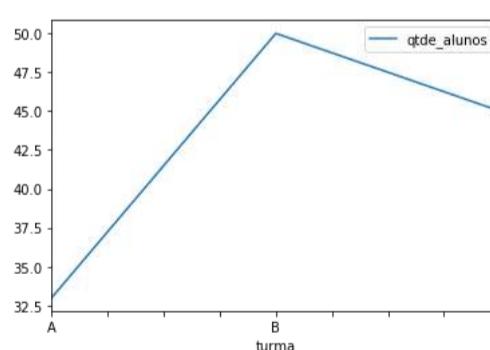


In [8]:

```
df.plot(x='turma', y='qtde_alunos', kind='line')
```

Out[8]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1cf33aeb668>
```



No endereço <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.plot.html> você pode encontrar a lista

com todos os tipos de gráficos possíveis de serem construídos com o método `plot()` da biblioteca. Para construir um gráfico do tipo pizza (pie), precisamos definir como índice os dados que serão usados como legenda. Veja a seguir, fazemos a transformação no DF seguido do plot com o tipo pie. Esse tipo de sintaxe é

chamado de encadeamento, pois ao invés de fazermos a transformação, salvar em um novo objeto e depois plotar, fazemos tudo em uma única linha, sem precisar criar o novo objeto.

0

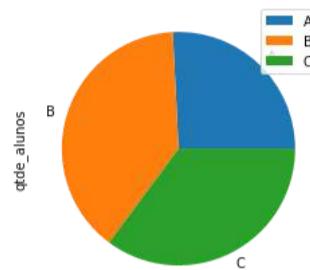
In [9]:

```
df.set_index('turma').plot(y='qtde_alunos', kind='pie')
```

Ver anotações

Out[9]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1cf33b7fd0>
```



Vale ressaltar que para todos os gráficos criados, a biblioteca oferece uma segunda opção de sintaxe, que é invocar o tipo de gráfico como método, por exemplo:

- `df.plot.bar(x='turma', y='qtde_alunos')`
- `df.plot.line(x='turma', y='qtde_alunos')`
- `df.set_index('turma').plot.pie(y='qtde_alunos')`

#### EXEMPLIFICANDO

Vamos utilizar os dados sobre a exportação de etanol hidratado (barris equivalentes de petróleo) 2012-2020, disponível no endereço:

<https://www.anp.gov.br/arquivos/dadosabertos/iee/exportacao-etanol-hidratado-2012-2020-bep.csv>, para analisarmos e extrairmos informações de forma visual. Para conseguir utilizar o método plot nessa base, teremos que transformar a vírgula em ponto (padrão numérico) e converter os dados para os tipos float e int. Veja o código a seguir em que preparamos os dados

In [10]:

```
df_etanol = pd.read_csv('exportacao-etanol-hidratado-2012-2020-bep.csv',  
sep=';', encoding="ISO-8859-1")  
  
# Apaga colunas que não usaremos  
df_etanol.drop(columns=['PRODUTO', 'MOVIMENTO COMERCIAL', 'UNIDADE'],  
inplace=True)  
  
# Substitui a vírgula por ponto em cada coluna  
for mes in 'JAN FEV MAR ABR MAI JUN JUL AGO SET OUT NOV DEZ  
TOTAL'.split():  
    df_etanol[mes] = df_etanol[mes].str.replace(',', '.')  
  
# Converte os valores para float  
df_etanol = df_etanol.astype(float)  
  
# Converte o ano para inteiro  
df_etanol['ANO'] = df_etanol['ANO'].astype(int)  
  
df_etanol.head(2)
```

Out[10]:

ANO	JAN	FEV	MAR	ABR	MAI	JUN	JUL
0	201287231.41132	141513.5186	122157.33850	98004.42926	153286.6078	144373.6894	384743.614224
1	2013673419.97670	387331.6487	96929.59201	54390.05046	115092.4820	387498.3792	339162.210035

Agora com os dados corretos podemos usar o método de plotagem para que, de forma visual, possamos identificar o ano que teve a menor e maior arrecadação para o mês de janeiro. Veja o código a seguir. No eixo x, vamos usar a informação de ano, e no y todos os valores da coluna 'JAN'; (kind) o tipo do gráfico será de barras; (figsize) a figura terá um tamanho de 10 polegadas na horizontal por 5 na vertical; (rot) a legenda no eixo x não deve ser rotacionada; (fontsize) o tamanho das fontes na figura deve ser 12 pt; (legend) não queremos legenda nesse gráfico. Pelo gráfico, podemos identificar com facilidade que o ano que teve menor arrecadação nesse mês, foi 2017 e o maior 2013. Também plotamos o mesmo gráfico, mas em linhas, veja como é possível explicar os dados agora sobre outra perspectiva: o comportamento da variação da arrecadação ao longo do tempo.

In [11]:

0

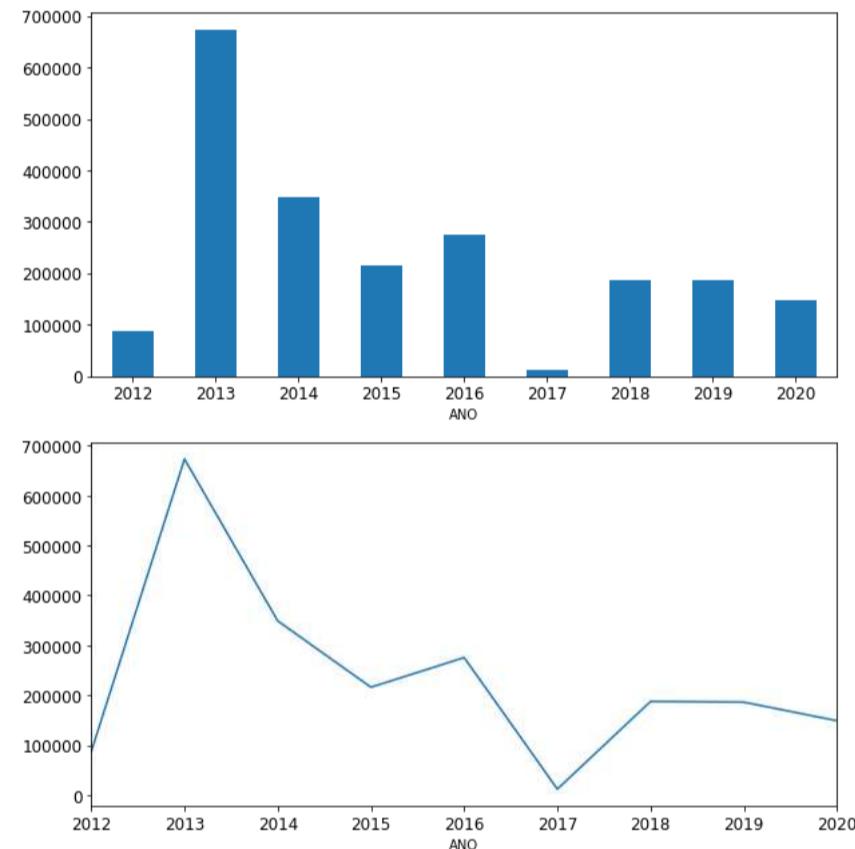
Ver anotações

```
df_etanol.plot(x='ANO',
                y='JAN',
                kind='bar',
                figsize=(10, 5),
                rot=0,
                fontsize=12,
                legend=False)

df_etanol.plot(x='ANO',
                y='JAN',
                kind='line',
                figsize=(10, 5),
                rot=0,
                fontsize=12,
                legend=False)
```

Out[11]:

&lt;matplotlib.axes.\_subplots.AxesSubplot at 0x1cf34c166a0&gt;



Queremos criar um gráfico, que nos possibilite comparar a arrecadação entre os meses de janeiro e fevereiro. Veja no código a seguir. Estamos selecionando três colunas do nosso DF e encadeando o método plot(), agora passando como parâmetro somente o valor x, o tipo, o tamanho, a rotação da legenda e o tamanho da fonte. Os valores para o eixo y, serão usados das colunas. Com essa construção se torna possível avaliar, visualmente, o desempenho nos meses. Veja que em determinados anos, a discrepância entre eles é considerável.

In [12]:

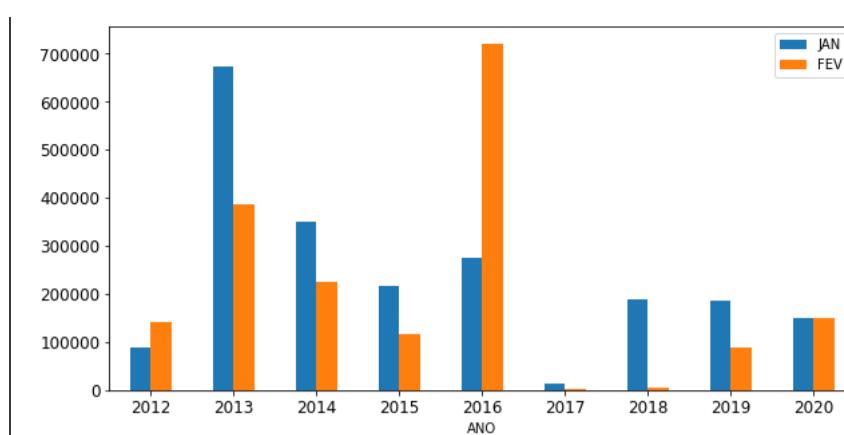
```
df_etanol[['ANO', 'JAN', 'FEV']].plot(x='ANO', kind='bar', figsize=(10, 5), rot=0, fontsize=12)
```

Out[12]:

&lt;matplotlib.axes.\_subplots.AxesSubplot at 0x1cf34c708d0&gt;

0

Ver anotações



0

Ver anotações

## BIBLIOTECA SEABORN

Seaborn é outra biblioteca Python, também baseada na matplotlib, que foi desenvolvida especificamente para criação de gráficos. Seaborn pode ser instalado via pip install: pip install seaborn, e para utilizar no projeto existe uma convenção para sintaxe: import seaborn as sns. A biblioteca conta com um repositório de datasets que podem ser usados para explorar as funcionalidades e estão disponíveis no endereço: <https://github.com/mwaskom/seaborn-data>. Vamos carregar os dados sobre gorjetas (tips) para nosso estudo. Veja no código a seguir, utilizamos a função load\_dataset(), cujo retorno é um DataFrame pandas, para carregar a base de dados. A seguir imprimimos as informações básicas que foram carregadas. Temos 244 linhas e 7 colunas, cujos dados são do tipo ponto flutuante, categóricos e um inteiro.

In [13]:

```
import seaborn as sns

# Configurando o visual do gráfico. Leia mais em
# https://seaborn.pydata.org/generated/seaborn.set.html#seaborn.set
sns.set(style="whitegrid") # opções: darkgrid, whitegrid, dark, white, ticks

df_tips = sns.load_dataset('tips')

print(df_tips.info())
df_tips.head()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 244 entries, 0 to 243
Data columns (total 7 columns):
total_bill    244 non-null float64
tip          244 non-null float64
sex           244 non-null category
smoker        244 non-null category
day           244 non-null category
time          244 non-null category
size          244 non-null int64
dtypes: category(4), float64(2), int64(1)
memory usage: 7.2 KB
None
```

Out[13]:

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

0

Ver anotações

O tipo de dados que uma coluna possui é muito importante para a biblioteca seaborn, uma vez que as funções usadas para construir os gráficos são divididas em grupos: relacional, categóricos, distribuição, regressão, matriz e grids (<https://seaborn.pydata.org/api.html>).

## FUNÇÃO BARPLOT()

Dentro do grupo de funções para gráficos de variáveis categóricas, temos o **barplot()**, que permite criar gráficos de barras, mas por que usariamos essa função e não a da biblioteca pandas? A resposta está nas opções de parâmetros que cada biblioteca suporta. Veja o construtor da função barplot:

```
seaborn.barplot(x=None, y=None, hue=None, data=None, order=None, hue_order=None,
estimator=function mean, ci=95, n_boot=1000, units=None, seed=None, orient=None,
color=None, palette=None, saturation=0.75, errcolor='.26', errwidth=None,
capsize=None, dodge=True, ax=None, **kwargs). Esse construtor possui uma série de
parâmetros estatísticos, que dão muita flexibilidade e poder aos cientista de
dados, vamos falar sobre o parâmetro "estimator", que por default é a função
média. Isso significa que cada barra do gráfico, exibirá a média dos valores de uma
determinada coluna, o que pode não fazer sentido, uma vez que queremos exibir a
quantidade dos valores (len) ou a soma (sum).
```

Para entender como esse parâmetro pode "afetar" a construção do gráfico, veja o código a seguir. Usamos o matplotlib para construir uma figura e um eixo com três posições. Nosso primeiro gráfico de barras (linha 3), utiliza o estimator padrão, que sabemos que é a média. O segundo (linha 4), utiliza a função de soma como estimator e o terceiro (linha 5), a função len, para contar. Veja no resultado como os gráficos são diferentes. No primeiro, nos conta que a o valor médio da conta entre homens e mulheres é próximo, embora os homens gastem um pouco mais. Já o segundo gráfico, nos mostra que os homens gastam "muito mais", será que é verdade? A soma da conta dos homens, de fato é superior a das mulheres, mas será que não existem muito mais homens do que mulheres na base? O terceiro gráfico nos conta isso, quantos homens e mulheres possuem na base.

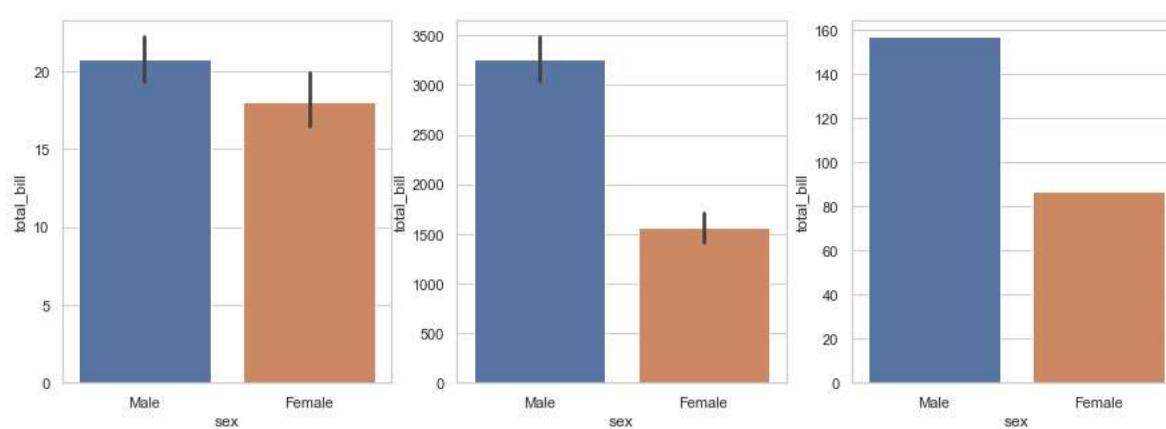
In [14]:

```
fig, ax = plt.subplots(1, 3, figsize=(15, 5))

sns.barplot(data=df_tips, x='sex', y='total_bill', ax=ax[0])
sns.barplot(data=df_tips, x='sex', y='total_bill', ax=ax[1], estimator=sum)
sns.barplot(data=df_tips, x='sex', y='total_bill', ax=ax[2], estimator=len)
```

Out[14]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1cf35b313c8>
```



0

Ver anotações

Construir gráficos não é somente plotar imagens bonitas, existem muitos conceitos estatísticos envolvidos e a biblioteca seaborn fornece mecanismos para que essas informações estejam presentes nos resultados visuais.

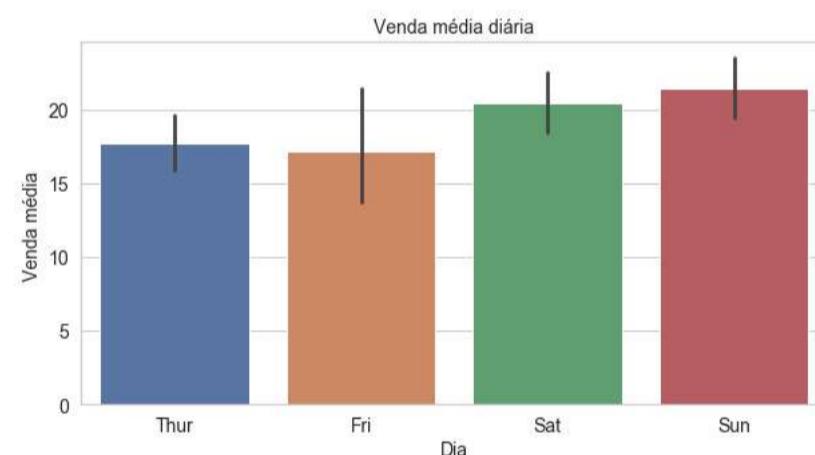
A biblioteca seaborn integra totalmente com a matplotlib. Vamos criar um gráfico que nos mostre o valor médio diário de venda. Veja o código a seguir, a criação do gráfico está na linha 3, mas configuramos seu tamanho na linha 1, e das linhas 5 a 8, configuramos os rótulos e os tamanhos.

In [15]:

```
plt.figure(figsize=(10, 5))

ax = sns.barplot(x="day", y="total_bill", data=df_tips)

ax.axes.set_title("Venda média diária", fontsize=14)
ax.set_xlabel("Dia", fontsize=14)
ax.set_ylabel("Venda média ", fontsize=14)
ax.tick_params(labelsize=14)
```



## ■ FUNÇÃO COUNTPLOT()

Conseguimos plotar a contagem de uma variável categórica, com a função barplot e o estimator len, entretanto, a biblioteca seaborn possui uma função específica para esse tipo de gráfico: `seaborn.countplot(x=None, y=None, hue=None, data=None, order=None, hue_order=None, orient=None, color=None, palette=None, saturation=0.75, dodge=True, ax=None, **kwargs)`.

Esse método não aceita que sejam passados valores de x e y ao mesmo tempo, pois a contagem será feita sobre uma variável categórica, portanto devemos especificar x ou y, a diferença será na orientação do gráfico. Se informamos x, teremos uma gráfico na vertical, se y, na horizontal.

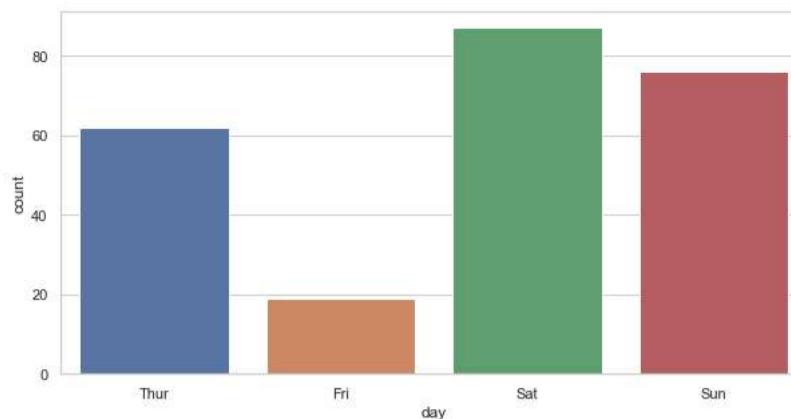
In [16]:

```
plt.figure(figsize=(10, 5))

sns.countplot(data=df_tips, x="day")
```

Out[16]:

&lt;matplotlib.axes.\_subplots.AxesSubplot at 0x1cf35ad2e80&gt;



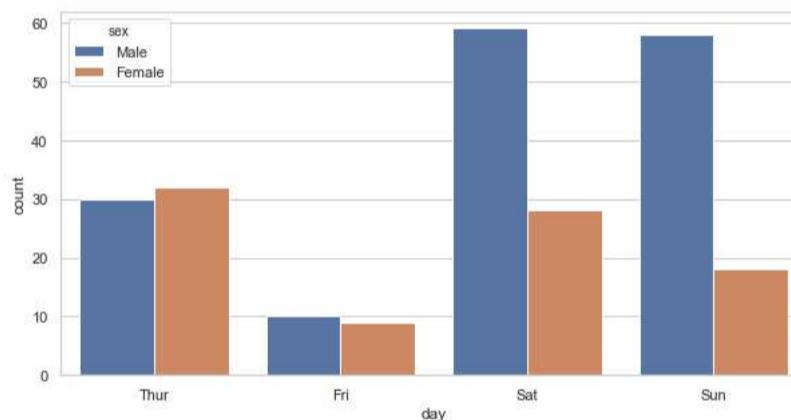
Vamos plotar mais um gráfico de contagem para mostrar o poder de um único parâmetro. O parâmetro **hue** é usado como entrada de dados, pois irá discriminar no gráfico a variável atribuída ao parâmetro. Para entendermos, vamos plotar a quantidade de pessoas por dia, mas discriminado por gênero, quantos homens e mulheres estiveram presentes em cada dia? Veja no código a seguir, a única diferença é o parâmetro.

In [17]:

```
plt.figure(figsize=(10, 5))
sns.countplot(data=df_tips, x="day", hue="sex")
```

Out[17]:

&lt;matplotlib.axes.\_subplots.AxesSubplot at 0x1cf35ba9240&gt;



## FUNÇÃO SCATTERPLOT()

Os gráficos do grupo relacional, permitem avaliar, de forma visual a relação entre duas variáveis: x, y. A função possui a seguinte sintaxe: `seaborn.scatterplot(x=None, y=None, hue=None, style=None, size=None, data=None, palette=None, hue_order=None, hue_norm=None, sizes=None, size_order=None, size_norm=None, markers=True, style_order=None, x_bins=None, y_bins=None, units=None, estimator=None, ci=95, n_boot=1000, alpha='auto', x_jitter=None, y_jitter=None, legend='brief', ax=None, **kwargs)`.

Vamos construir um gráfico que permita avaliar se existe uma relação entre o valor da conta e da gorjeta. Será que quem gastou mais também deu mais gorjeta? Veja o código a seguir, invocamos a função passando o valor da conta como parâmetro para x e a gorjeta para y. Agora vamos avaliar o resultado. Cada "bolinha" representa uma conta paga e uma gorjeta, por exemplo, a bolinha mais a direita, podemos interpretar que para uma conta de aproximadamente 50 e poucos dólares foi dada uma gorjeta de 10. Olhando para o gráfico, parece quanto maior o

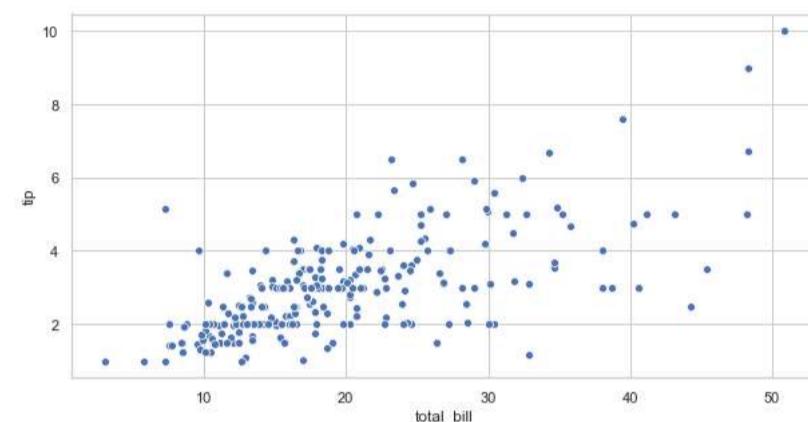
valor da conta, maior foi o valor da gorjeta. Esse comportamento é chamado de relação linear, pois conseguimos traçar uma reta entre os pontos, descrevendo seu comportamento através de uma função linear.

In [18]:

```
plt.figure(figsize=(10, 5))  
sns.scatterplot(data=df_tips, x="total_bill", y="tip")
```

Out[18]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1cf35c7e908>
```



O gráfico scatterplot é muito utilizado por cientistas de dados que estão buscando por padrões nos dados. O padrão observado no gráfico, que mostra a relação entre o valor da conta e da gorjeta, pode ser um forte indício que, caso o cientista precise escolher um algoritmo de aprendizado de máquina para prever a quantidade de gorjeta que um cliente dará, ele poderá uma regressão linear.

O universo dos dados é cada vez mais requisitado nas empresas, se você gostou do que aprendemos, não deixe de investir mais tempo em estudo e treinamento, pois há muito o que se aprender!

#### REFERÊNCIAS E LINKS ÚTEIS

IPython Development Team. Built-in magic commands. Disponível em:

<https://ipython.readthedocs.io/en/stable/interactive/magics.html>. Acesso em: 17 jun. 2020.

Matplotlib development team. matplotlib.pyplot. Disponível em:

[https://matplotlib.org/api/\\_as\\_gen/matplotlib.pyplot.html](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.html)) Acesso em: 27 jun. 2020.

Michael Waskom. seaborn. Disponível em: (<https://seaborn.pydata.org/api.html>) Acesso em: 27 jun. 2020.

Pandas Team. pandas.DataFrame.plot. Disponível em:

<https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.plot.html>). Acesso em: 27 jun. 2020.

Secretaria de Tecnologia da Informação, Ministério do Planejamento, Desenvolvimento e Gestão. Conjuntos de dados. Disponível em:  
<https://www.dados.gov.br/dataset>. Acesso em: 27 jun. 2020.

0

Ver anotações

# FOCO NO MERCADO DE TRABALHO

## VISUALIZAÇÃO DE DADOS EM PYTHON

Vanessa Cadan Scheffer

0

Ver anotações

### ANÁLISE E APRESENTAÇÃO DE DADOS

As bibliotecas pandas, matplotlib e seaborn podem ser utilizadas para o carregamento dos dados e a geração dos gráficos.



Fonte: Shutterstock.

### Deseja ouvir este material?

Áudio disponível no material digital.

### DESAFIO

Como desenvolvedor em uma empresa de consultoria de software, você foi alocado em um projeto para uma empresa de telecomunicações. Essa empresa tem interesse em habilitar um novo serviço, mas antes precisa entender qual a

disponibilidade dos satélites autorizados a operar no Brasil. Para a primeira sprint (período de 15 dias de trabalho), você foi encarregado de apresentar, uma análise preliminar da situação dos satélites.

Nessa primeira entrega, você deve apresentar a comparação da quantidade de satélites que são brasileiros, dos que são estrangeiros. Dentre os satélites brasileiros, você deve discriminar a quantidade de cada operadora comercial, bem como a quantidade de satélites operando em cada banda. As mesmas análises devem ser feitas para os satélites que pertencem a outros países.

Onde esses dados podem ser encontrados? Qual a melhor forma de apresentar os resultados, basta levar os números? Qual biblioteca pode ser usada para resolver o desafio?

## RESOLUÇÃO

Um dos grandes desafios nessa primeira entrega é encontrar uma fonte confiável de dados.

No endereço <https://www.dados.gov.br/dataset>, existe uma categoria específica para esse tipo de informação: Agência Nacional de Telecomunicações - Anatel.

Dentro dessa categoria encontramos um arquivo delimitado (csv) com a relação de satélites autorizados a operar no Brasil: <https://www.dados.gov.br/dataset/relacao-de-satelites-geoestacionarios-autorizados-a-operar-no-brasil>, basta clicar no recurso e fazer download para a pasta do projeto.

Agora que identificamos uma fonte confiável podemos usar as bibliotecas pandas, matplotlib e seaborn para carregar os dados e gerar gráficos que contemplam as informações solicitadas.

In [19]:

```
import pandas as pd  
import matplotlib.pyplot as plt  
import seaborn as sns
```

0

Ver anotações

Vamos carregar os dados em um DataFrame pandas, chamado df\_satelites. Os dados possuem como delimitador ";", logo é preciso informar ao método read\_csv esse parâmetro. Também é preciso garantir que linhas duplicadas sejam removidas (linha 2) e para ter os índices variando de 0 até N, vamos resetar (linha 3).

o

Ver anotações

Como resultado temos um DF com 68 linhas e 7 colunas.

In [20]:

```
df_satelites = pd.read_csv('satelites_operando_comercialmente.csv', sep=';')
df_satelites.drop_duplicates(inplace=True)
df_satelites.reset_index(drop=True, inplace=True)

print(df_satelites.info())
df_satelites.head()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 68 entries, 0 to 67
Data columns (total 7 columns):
Satelite operando      68 non-null object
Órbita                  68 non-null object
Bandas                  68 non-null object
Status do Satélite     68 non-null object
Pos. Orbital            68 non-null object
Direito                 68 non-null object
Operadora Comercial    68 non-null object
dtypes: object(7)
memory usage: 3.8+ KB
None
```

Out[20]:

	<b>Satelite operando</b>	<b>Órbita</b>	<b>Bandas</b>	<b>Status do Satélite</b>	<b>Pos. Orbital</b>	<b>Direito</b>	<b>Operador Comercial</b>
0	EUTELSAT 65 West A	Satélite Geoestacionário (GEO)	C (AP30B), Ku (AP30B), Ka	operação comercial	65°O	Brasileiro	EUTELSAT DO LTDA
1	HISPASAT 74W-1	Satélite Geoestacionário (GEO)	Ku (AP30/30A)	operação comercial	74°O	Brasileiro	HISPAMAR SATELLITE S.A.
2	YAH 3	Satélite Geoestacionário (GEO)	Ka	operação comercial	20°O	Brasileiro	YAH TELECOMUNICAÇÕES LTDA
3	SGDC	Satélite Geoestacionário (GEO)	Ka	operação comercial	75°O	Brasileiro	TELECOMUNICACOES BRASILEIRAS SA TELEBRAS
4	SES-14	Satélite Geoestacionário (GEO)	C (não planejada), Ku (não planejada), Ka	operação comercial	47,5°O	Brasileiro	SES DTH DO BRASIL LTDA

Agora vamos criar um gráfico que faz a contagem e visualmente, faz a comparação entre a quantidade de satélites brasileiros e estrangeiros. Podemos usar o countplot(), passando como parâmetros o DF e a coluna 'Direito', como variável categórica a ser contada. Também podemos usar os recursos da biblioteca matplotlib para configurar o tamanho da figura e dos textos nos eixos.

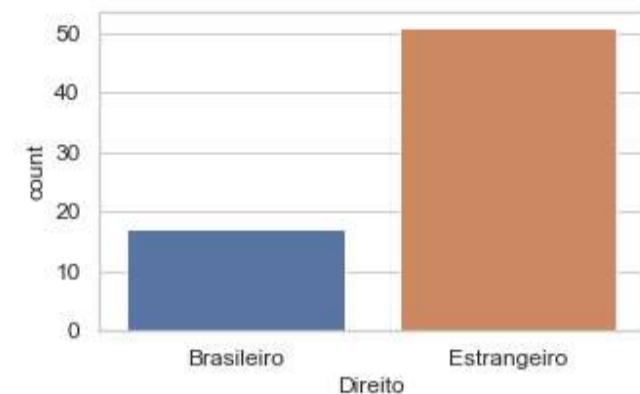
In [21]:

```
# Quantos satélites são brasileiros e quantos são estrangeiro?
```

```
plt.figure(figsize=(5,3))
plt.tick_params(labelsize=12)
sns.countplot(data=df_satelites, x='Direito')
```

Out[21]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1cf35cfdbe0>
```



Agora vamos extrair as informações sobre os satélites brasileiros. Para facilitar, vamos criar um novo DataFrame aplicando um filtro. Veja na linha 13, que o DF df\_satelites\_brasileiros, será um filtro do DF df\_satelites onde somente os brasileiros estarão presentes. Agora, podemos usar o countplot no df\_satelites\_brasileiros para contar quantos satélites cada operadora comercial no Brasil possui (linha 8). Como o nome das operadoras é longo, vamos pedir para exibir na vertical, por isso configuramos a rotação do "xticks" na linha 6. Na linha 7 configuramos o tamanho dos textos nos eixos.

In [22]:

```
# quantos satélites cada operadora brasileira possui operando?
```

```
df_satelites_brasileiros = df_satelites.loc[df_satelites['Direito'] ==
    'Brasileiro']
```

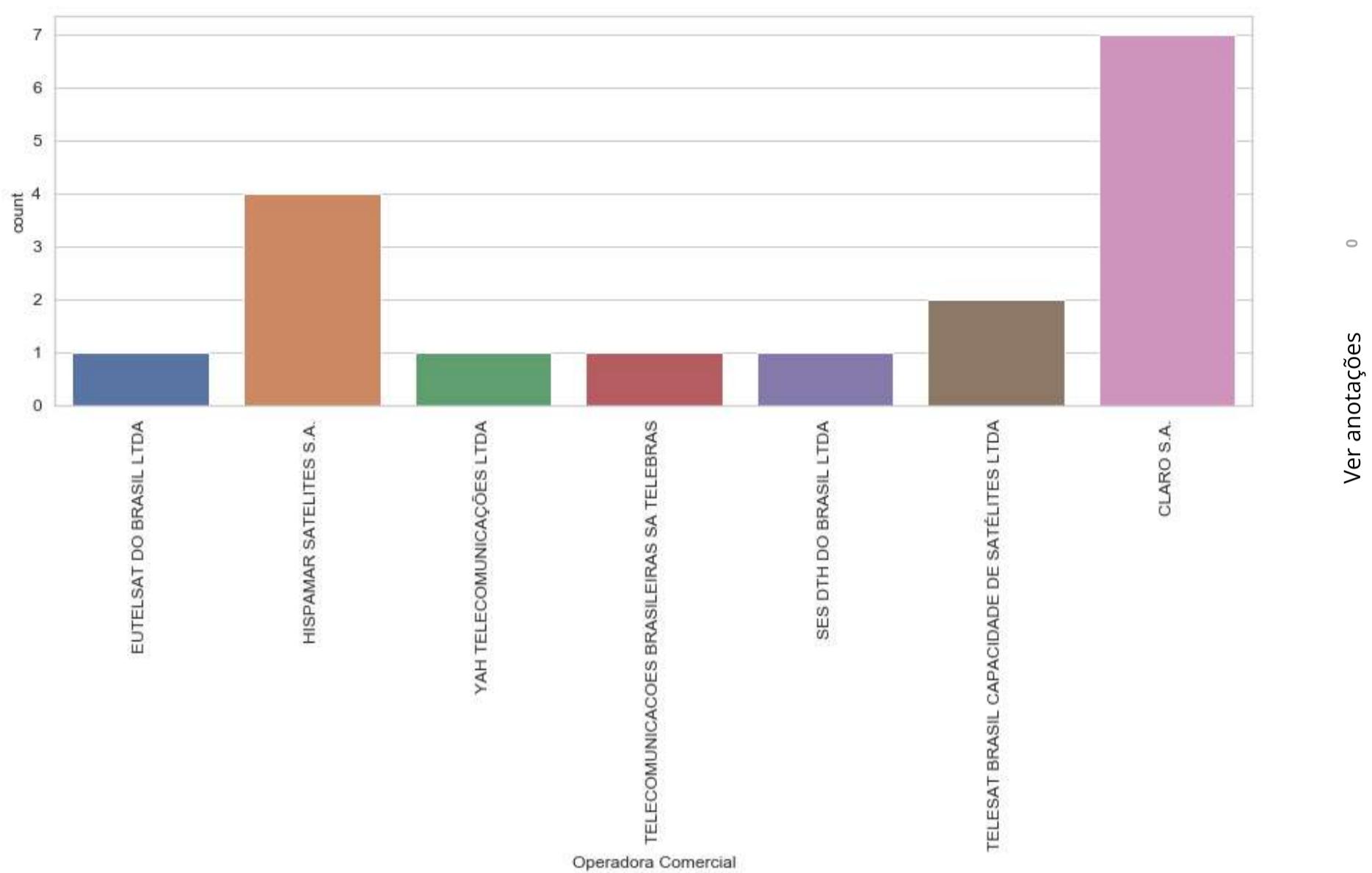
```
plt.figure(figsize=(15,5))
plt.xticks(rotation=90)
plt.tick_params(labelsize=12)
sns.countplot(data=df_satelites_brasileiros, x='Operadora Comercial')
```

Out[22]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1cf36eb4e10>
```

0

Ver anotações



Para saber quantos satélites brasileiros estão operando em cada banda, vamos usar o countplot, passando como parâmetro o df\_satelites\_brasileiros e a coluna 'Bandas'. Novamente foi necessário configurar o texto nos eixos.

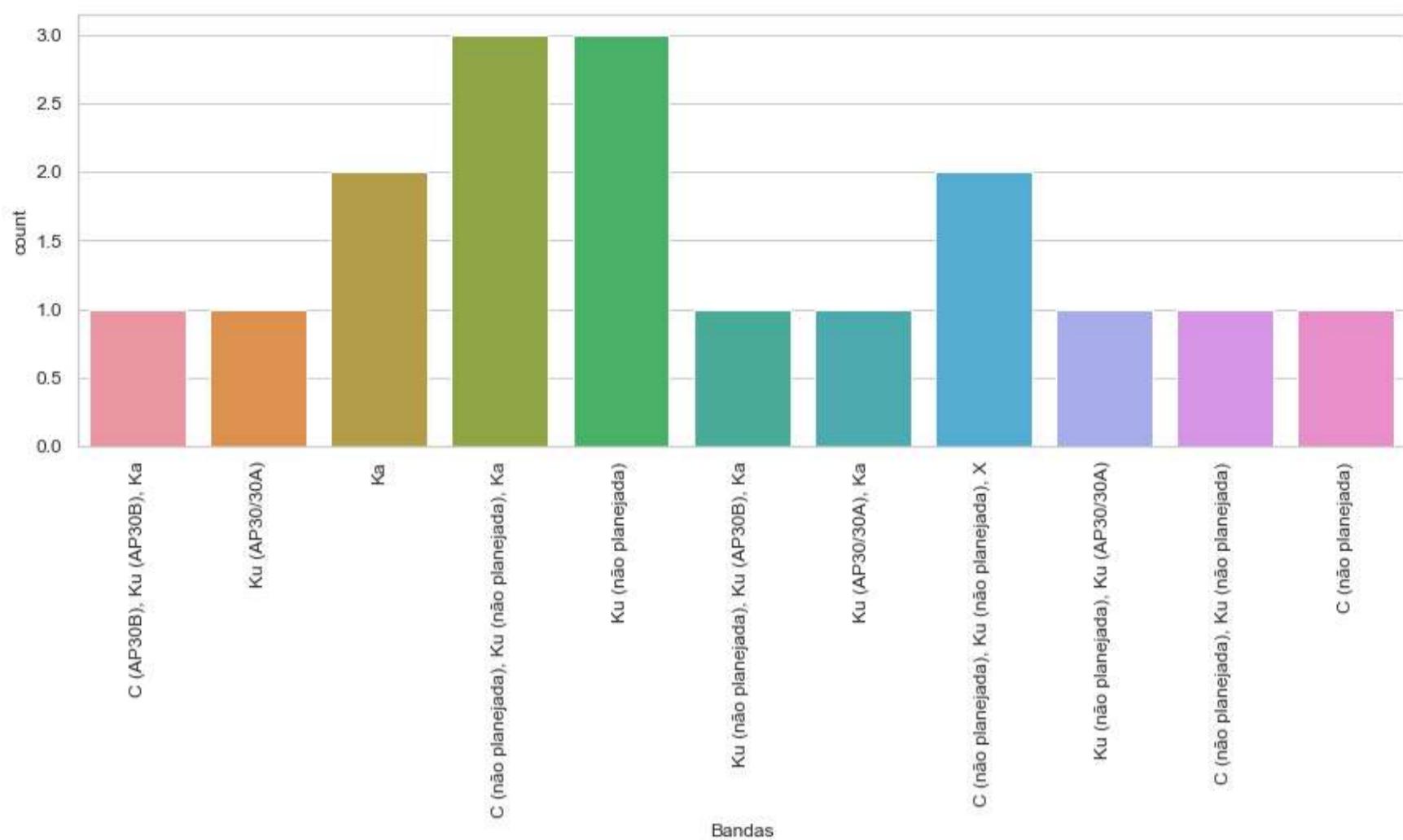
In [23]:

```
# Quantos satélites brasileiros estão operando em cada banda?

plt.figure(figsize=(15,5))
plt.xticks(rotation=90)
plt.tick_params(labelsize=12)
sns.countplot(data=df_satelites_brasileiros, x='Bandas')
```

Out[23]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1cf33aeb588>
```



[Ver anotações](#) 0

Agora vamos repetir os mesmos processos para os satélites estrangeiros, começando pela criação de um DataFrame que contenha somente as informações sobre eles (linha 3). Esse primeiro gráfico mostra quantos satélites cada operadora estrangeira possui em operação.

In [24]:

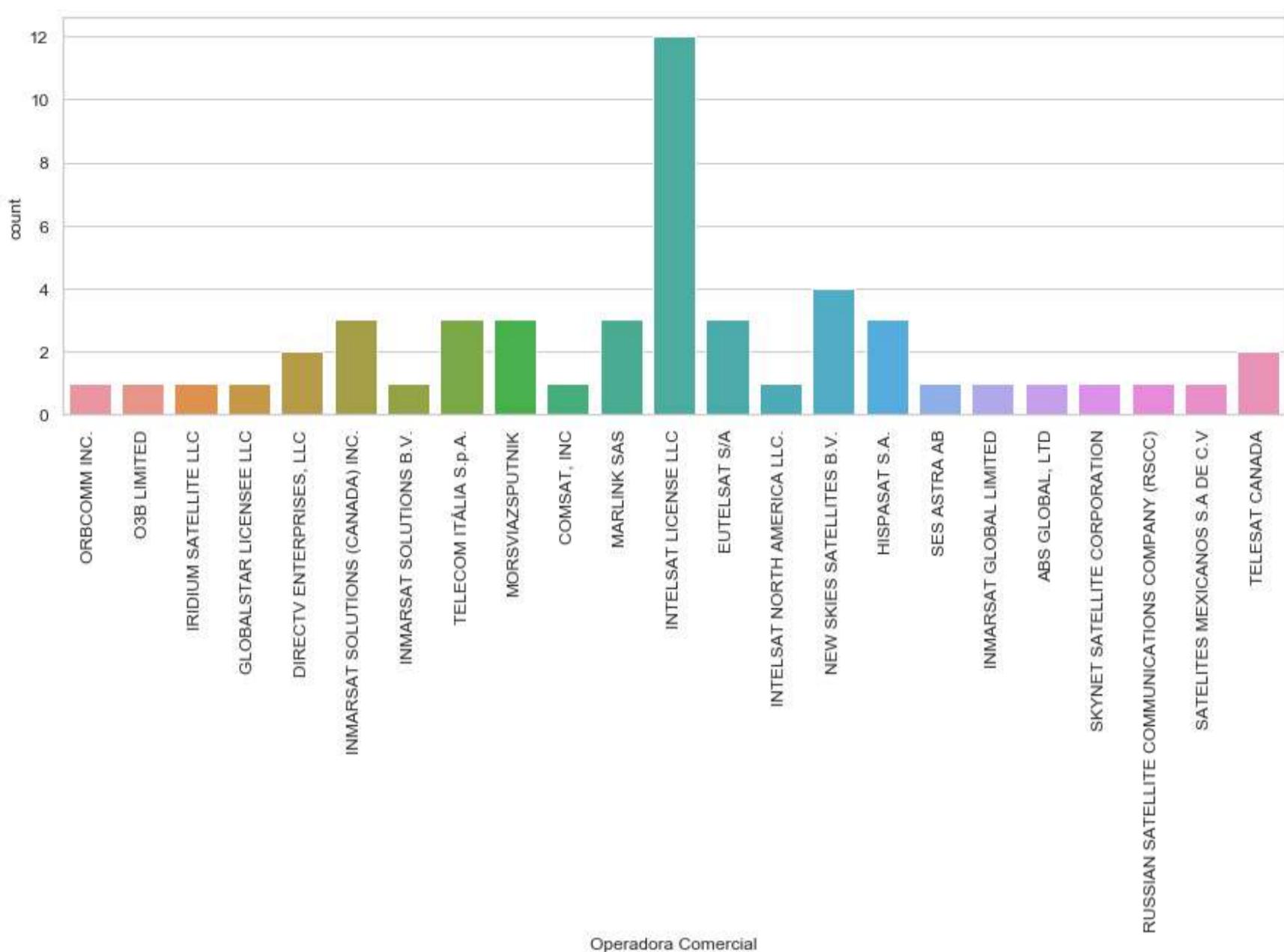
```
# Quantos satélites cada operadora estrangeira possui operando?
```

```
df_satelites_estrangeiros = df_satelites.loc[df_satelites['Direito'] ==  
'Estrangeiro']
```

```
plt.figure(figsize=(15,5))  
plt.xticks(rotation=90)  
plt.tick_params(labelsize=12)  
sns.countplot(data=df_satelites_estrangeiros, x='Operadora Comercial')
```

Out[24]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1cf36f5ab70>
```



Ver anotações 0

Agora vamos plotar quantos satélites estrangeiros estão operando em cada banda.

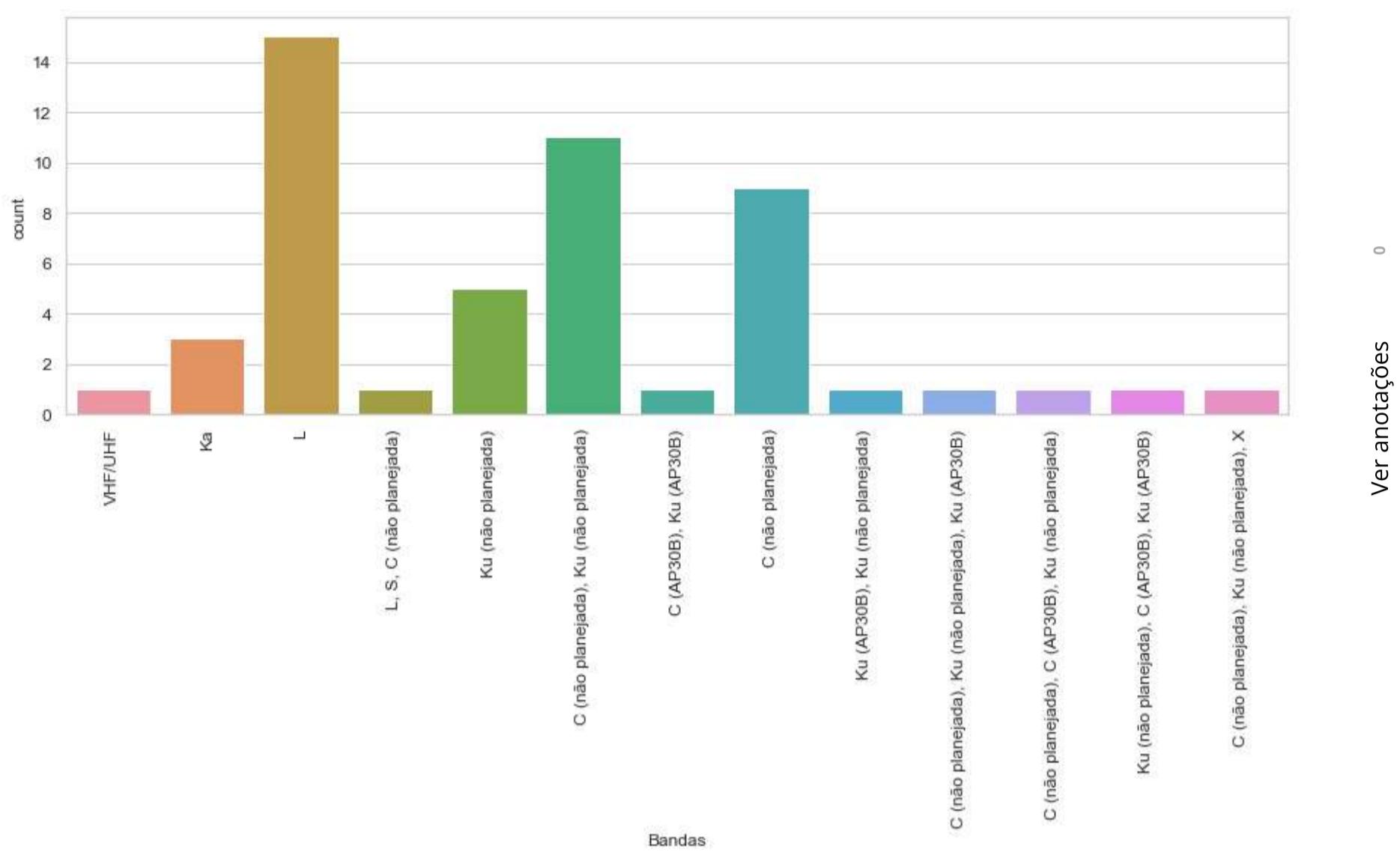
In [25]:

```
# Quantos satélites brasileiros estão operando em cada banda?

plt.figure(figsize=(15,5))
plt.xticks(rotation=90)
plt.tick_params(labelsize=12)
sns.countplot(data=df_satelites_estrangeiros, x='Bandas')
```

Out[25]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1cf36f4da20>
```



Com essas informações o cliente começará a ter argumentos para a escolha de uma operadora e banda que deseja contratar. Que tal personalizar um pouco mais o tamanho das legendas, criar títulos para os gráficos?

## DESAFIO DA INTERNET

Ganhar habilidade em programação exige estudo e treino (muito treino). Acesse o endereço <https://www.kaggle.com/datasets>, faço seu cadastro e escolha uma base de dados para treinar e desenvolver seu conhecimento sobre visualização de dados.