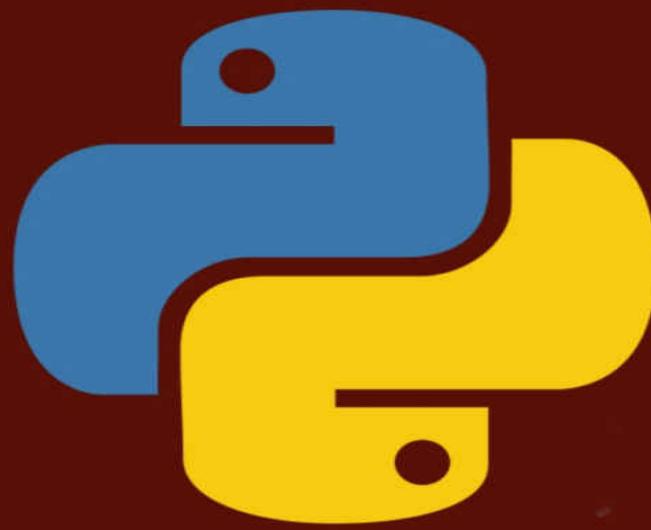


SÉRIE CIENTISTA DE DADOS

# PYTHON

Guia prático do  
básico ao avançado



RAFAEL FVC SANTOS

*Segunda edição*

**E-book**

# **Python**

*Guia prático do básico ao avançado*

Por:

*Rafael F. V. C. Santos*

2<sup>a</sup> Edição.

**COPYRIGHT© 2020 BY RAFAEL F V C SANTOS**

Todos os direitos reservados e protegidos pela Lei no 9.610 de 19/02/1998. Nenhuma parte desta edição pode ser utilizada e/ou reproduzida - em qualquer meio ou forma, seja mecânico ou eletrônico, fotocópia, gravação etc. - nem apropriada ou estocada em sistema de bancos de dados, sem a expressa autorização do autor.

2<sup>a</sup> Edição, 2020.

# Sumário

## Capítulo 1

[História do Python](#)

[Porque usar o Python?](#)

[Versões do Python](#)

[Ambiente de Desenvolvimento](#)

[Download e Instalação do Anaconda](#)

[As principais bibliotecas Python](#)

[Básicas:](#)

[Gráficas:](#)

[Álgebra simbólica:](#)

[Estatística:](#)

[Outras](#)

[Como instalar pacotes](#)

[Jupyter Lab](#)

[Funcionalidades do Jupyter Notebook](#)

[Abrindo um serviço local no Jupyter](#)

[Diferença entre Code e Markdown](#)

[Programação com Python](#)

[Indentação](#)

[Comentários](#)

[Dicas de Estudos](#)

## Capítulo 2

[Variáveis, tipos, estruturas e operações](#)

[Como estruturar um programa](#)

[Operações matemáticas e lógicas](#)

[Funções \*built-in\*](#)

[Operadores matemáticos, relacionais e lógicos](#)

[Variáveis e tipos](#)

[Variável Inteira](#)

[Float e double](#)

[Variável String](#)

[Variável Booleana](#)

[Declarações Múltiplas](#)

[Principais erros de declaração de variáveis](#)

[Trabalhando com variáveis](#)

[Indexação em Python](#)

[Funções Built-in das Strings](#)

[Variáveis para armazenamento de dados](#)

[Listas](#)

[Deletando elemento de uma lista](#)

[Aninhamento de Listas](#)

[Junção de Listas e avaliações de elementos](#)

[Cópia e criação de Listas](#)

[List Comprehension](#)

[Dicionários](#)

[Principais métodos dos dicionários](#)

[Tuplas](#)

[Vetores e matrizes](#)

[Vetores](#)

[Matrizes](#)

[Operações básicas com Vetores e Matrizes](#)

[Capítulo 3](#)

[Condicionais, laços de repetição e listas](#)

[Condicionais if/else/elif](#)

[Condicional if](#)

[Condicional else](#)

[Condicional Elif](#)

[Laços de Repetição for e while](#)

[Loop for](#)

[Loop while](#)

[Capítulo 4](#)

[Funções](#)

[Variáveis locais e globais - funções](#)

[Funções úteis no Python](#)

[Funções com parâmetros indeterminados](#)

[Tópico especial: Funções Lambda](#)

[Capítulo 5](#)

[I/O de arquivos](#)

[Arquivos CSV](#)

[Trabalhando com diretórios](#)

[Data e hora](#)

[Capítulo 6](#)

[Geração de gráficos e diagramas com o Python](#)

## [Gerando Gráficos com o Matplotlib](#)

[Gráfico de Barras](#)

[Histogramas](#)

[Gráficos de Pizza](#)

## [Capítulo 7](#)

[Funções especiais](#)

[Função Map](#)

[Função Reduce](#)

[Função Filter](#)

[Função Zip](#)

[Função Enumerate](#)

## [Capítulo 8](#)

[Mensagens de erros, warnings](#)

[Erro de Sintaxe \(SyntaxError\)](#)

[Erro de Identação \(IndentationError\)](#)

[Erro de nome \(NameError\)](#)

[Erro de Valor \(ValueError\)](#)

[Erro de Index \(IndexError\)](#)

[Erro de chave \(KeyError\)](#)

[Erro de Tipo \(TypeError\)](#)

[Diferença entre erro e exceção](#)

[Erros x Exceções](#)

[Tratando erros e exceções - Try, Except e Finally](#)

## [Capítulo 9](#)

[Classes e objetos com Python](#)

[Principais características da POO](#)

[Abstração](#)

[Encapsulamento](#)

[Herança](#)

[Polimorfismo](#)

[Classes e Objetos no Python](#)

[Em Python tudo é um objeto](#)

[Herança em Python](#)

[Métodos especiais em Python](#)

## [Capítulo 10](#)

[Conexão com banco de dados](#)

[Linguagem SQL](#)

[SQLite](#)

[Entidades](#)  
[Registros](#)  
[Chaves](#)  
[Relacionamentos](#)  
[Integridade referencial](#)  
[Normalização de dados](#)  
[Python e SQLite](#)  
[Agradecimentos](#)

## Sobre o Autor

Rafael F V C Santos ([rafaelfvcs@gmail.com](mailto:rafaelfvcs@gmail.com))

**Site:** <https://analistasquant.wordpress.com/>

**Twitter:** @rafaelfvcs

Especialista em gestão estratégica de riscos aplicados ao mercado financeiro. Trabalha com o desenvolvimento de estratégias automatizadas de investimentos (Robôs de investimentos - *Expert Advisor*) utilizando *machine learning* e estatística espacial. Formado em Engenharia Química pela Universidade Federal de Pernambuco (UFPE). Possui mestrado e doutorado em Engenharia Civil (UFPE) nas áreas de caracterização, modelagem e simulação estocástica, aplicadas a poços e reservatórios de petróleo. Possui livros na área de mercado financeiro e diversos artigos, com o tema de estatística aplicada, publicados em revistas e congressos nacionais e internacionais.

Também, é professor na plataforma de ensino a distância Udemy. Seus principais cursos são:

- Python Fundamentos (<http://bit.ly/python-fundamentos>)
- Estatística Descritiva com Python (<http://bit.ly/stats-descritiva-python>)
- Álgebra Linear com Python (<http://bit.ly/alg-linear-python>)
- Crie seu 1º Robô de Investimentos com MQL5 (<http://bit.ly/1-robo-invest-mql5>)
- Estratégias avançadas para robôs invest. (<http://bit.ly/mql5-avancado-1>)
- Como criar indicadores técnicos de Invest. (<http://bit.ly/ind-tecnicos-mql5>)
- Matemática para Data Science – Pré-Cálculo (<http://bit.ly/mat-pre-calcu>)

- Manipulação e análise de dados com Pandas Python (<http://bit.ly/pandas-python>)
- Fundamentos da linguagem R – Data Science Tool (<http://bit.ly/aprenda-linguagem-r>)
- Análise técnica para investidores e analistas Quant

## Prefácio

Uma das ferramentas mais importantes para os analistas de dados e cientistas em geral são as linguagens de comunicação com os computadores. Aprender a se comunicar com o computador é obrigação de todos na era digital e do big data. Atualmente temos a nosso alcance um número muito grande de linguagens de programação de computadores. Essas ferramentas estão à nossa disposição em um nível de comunicação cada vez mais alto. Alto nível para linguagem de programação significa uma maneira de se comunicar com as máquinas de um jeito mais próximo da linguagem humana.

No início da década de 1970, a grande maioria das linguagens de computadores era dita de baixo nível. Isso significava que para darmos ordens aos computadores precisávamos dominar algo bem mais difícil de compreender comparado ao que precisamos hoje em dia. Naquela época as linguagens apresentavam uma estrutura mais próxima das máquinas (binária).

Felizmente, estamos na era digital e agora podemos escolher, a partir de nossas necessidades e preferências, a linguagem de programação que mais nos agrada. Agora conseguimos ter à nossa disposição aquela linguagem que mais se dispõe a resolver nossos problemas com o mínimo esforço computacional. Graças aos diversos blogs, fóruns e sites de educação a distância a aprendizagem dessas linguagens se tornou democraticamente acessível a praticamente todos. Com apenas um smartphone já é possível instalar e aprender várias linguagens de programação.

Este e-book traz uma apresentação, do básico ao avançado, de uma das mais poderosas e simples linguagens de programação. Iremos conhecer e aprender a dominar todos os principais elementos para que possamos desenvolver nossos projetos e pôr nossas ideias para o mundo dos negócios digitais.

Estamos vivendo na era do ***big data***. Quando nos referenciamos ao big data queremos dizer que um **volume** muito grande de dados em uma **variedade** (texto, fotos, vídeos, áudios) e **velocidade** nunca antes vista está sendo gerado para nós. É fato que esses dados guardam uma enorme riqueza de informações e conhecimentos. Mas para chegarmos a tais benefícios precisamos armazenar, manipular, caracterizar, modelar, analisar e simular esses dados da maneira adequada. A maneira adequada quem vai dizer é o nosso modelo de negócio ou investigação do problema.

Precisamos de ferramentas aptas a nos ajudar com as atividades para trabalhar com esse novo grande volume de dados. E é ai que entram as linguagens de programação de computadores. E é com a união de ciências como estatística, matemática e computação que nasce a ciência de dados.

A área da ciência de dados nunca cresceu tanto e esteve tão carente com relação a profissionais qualificados. Python é uma das linguagens de programação chave para todo e qualquer analista que ouse trabalhar com grandes volumes de dados.

Foi pensando em ajudar cientistas, analistas e futuros profissionais do mundo do big data surgiu a ideia de escrever esse material.

Portanto, vamos em frente conhecer as façanhas, facilidades e aplicações que Python tem a nos oferecer. Mão à obra!

Rafael FVC Santos  
Brasil, abril de 2020



# **Capítulo 1**

## História do Python

No ano de 1989, o pesquisador do Instituto Nacional de Matemática Computacional de Amsterdam na Holanda (*National Research Institute for Mathematics and Computer Science in Amsterdam - CWI*) chamado Guido Van Rossum teve a necessidade de uma linguagem de alto-nível para desempenhar suas atividades de pesquisa. Como na época nenhuma linguagem de programação oferecia o que ele precisava, então iniciou a criação e desenvolvimento da linguagem Python.

Rossum era acostumado a desenvolver no paradigma funcional estrutural, mas, já sabendo que o futuro aguardava a chamada programação orientada a objetos, fez do Python uma linguagem multiparadigma, assim podemos programar tanto no modo funcional quanto orientado a objetos.

Ele queria uma forma de comunicação com as máquinas mais simplificada e menos verbosa que as existentes até o momento. Queria minimizar o máximo possível das estruturas, retirando, por exemplo, as chaves e parênteses excessivos dentro dos algoritmos. Foi assim que ele começou a desenvolver a linguagem Python.

Ele deu esse nome a linguagem por conta do seriado de comédia bastante popular na época chamado de Tropa Monty Python.



Figura 1.1 – Série britânica televisiva The Monte Python que fez sucesso nos anos 70.

O projeto de desenvolvimento do Python foi *Open Source* e sua primeira versão foi disponibilizada no início do ano de 1991. Até hoje a comunidade de desenvolvedores vem ganhando adeptos do mundo todo. Muitas grandes empresas do ramo tecnológico vêm fazendo investimentos expressivos no desenvolvimento de melhorias e novas bibliotecas e frameworks<sup>[1]</sup> com Python.

## Porque usar o Python?

Investimento em educação nunca é demais ou perdido. No entanto, antes de investirmos nosso escasso dinheiro e tempo no aprendizado de algo, é importante termos bons motivos. Certamente, aprender Python é algo indispensável para aqueles que estão interessados em desenvolver projetos tecnológicos inovadores. O slogan: '***programe ou seja programado***' nunca esteve tão evidente. Python pode ser um dos principais caminhos para nos tirar da ignorância com relação a uma comunicação mais próxima com as máquinas.

Veja alguns dos principais benefícios que você irá ter a sua disposição ao aprender Python:

- Softwares limpos e claros;
- *Open Source*;
- Produtividade elevada;
- Curva de aprendizagem pequena (fácil de aprender e usar);
- Grandes empresas utilizando e investindo em melhorias;
- Linguagem multiplataforma e paradigma;
- Uma das melhores linguagens para trabalhar com inteligência artificial (*machine learning* e *deep learning*);
- Ampla variedade de uso (*games*, imagens, robótica, paralelização etc.).

Algo importante com relação ao aprendizado de alguma tecnologia é a quantidade de usuários e documentação disponível a respeito. No mercado editorial existe uma infinidade de livros para todos os níveis de conhecimento e aprofundamento. Na internet também podem ser encontrados diversos *sites*, blogs e fóruns com muita informação e conhecimentos importantes para o aprendizado da linguagem Python.

Veja abaixo um *rank* mostrando o posicionamento da linguagem Python frente às outras mais utilizadas no mercado em 2019.

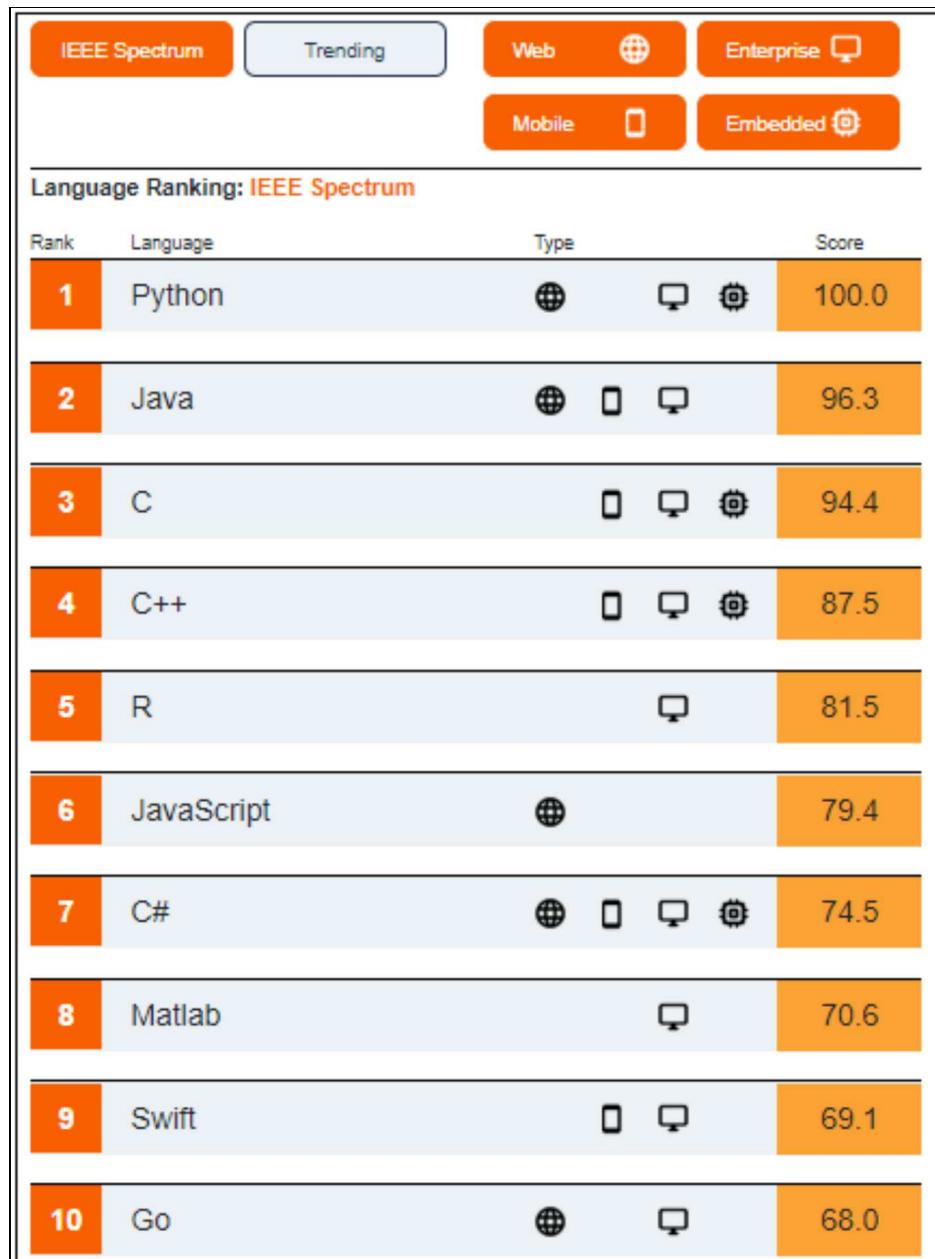


Figura 1.2 - Fonte da imagem:<https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2019>

Segue abaixo outro *rank*, agora da Tiobe<sup>[2]</sup> também para 2019:

Dec 2019	Dec 2018	Change	Programming Language	Ratings	Change
1	1		Java	17.253%	+1.32%
2	2		C	16.086%	+1.80%
3	3		Python	10.308%	+1.93%
4	4		C++	6.196%	-1.37%
5	6	▲	C#	4.801%	+1.35%
6	5	▼	Visual Basic .NET	4.743%	-2.38%
7	7		JavaScript	2.090%	-0.97%
8	8		PHP	2.048%	-0.39%
9	9		SQL	1.843%	-0.34%
10	14	▲	Swift	1.490%	+0.27%

Figura 1.3 – Ranking do índice TIOBE para dezembro de 2019.

Atualmente (2019) a linguagem Python apresenta a posição de número 4 (*subiu 3 posições em relação a 2018*) segundo a quantidade de usuários que comentam e postam dúvidas a respeito do Python no repositório do *Stackoverflow*. Esse é um dos maiores repositórios de códigos do mundo, utilizado por programadores de diversas áreas. Veja uma lista das principais tecnologias mais badaladas no *Stackoverflow*<sup>[3]</sup>.

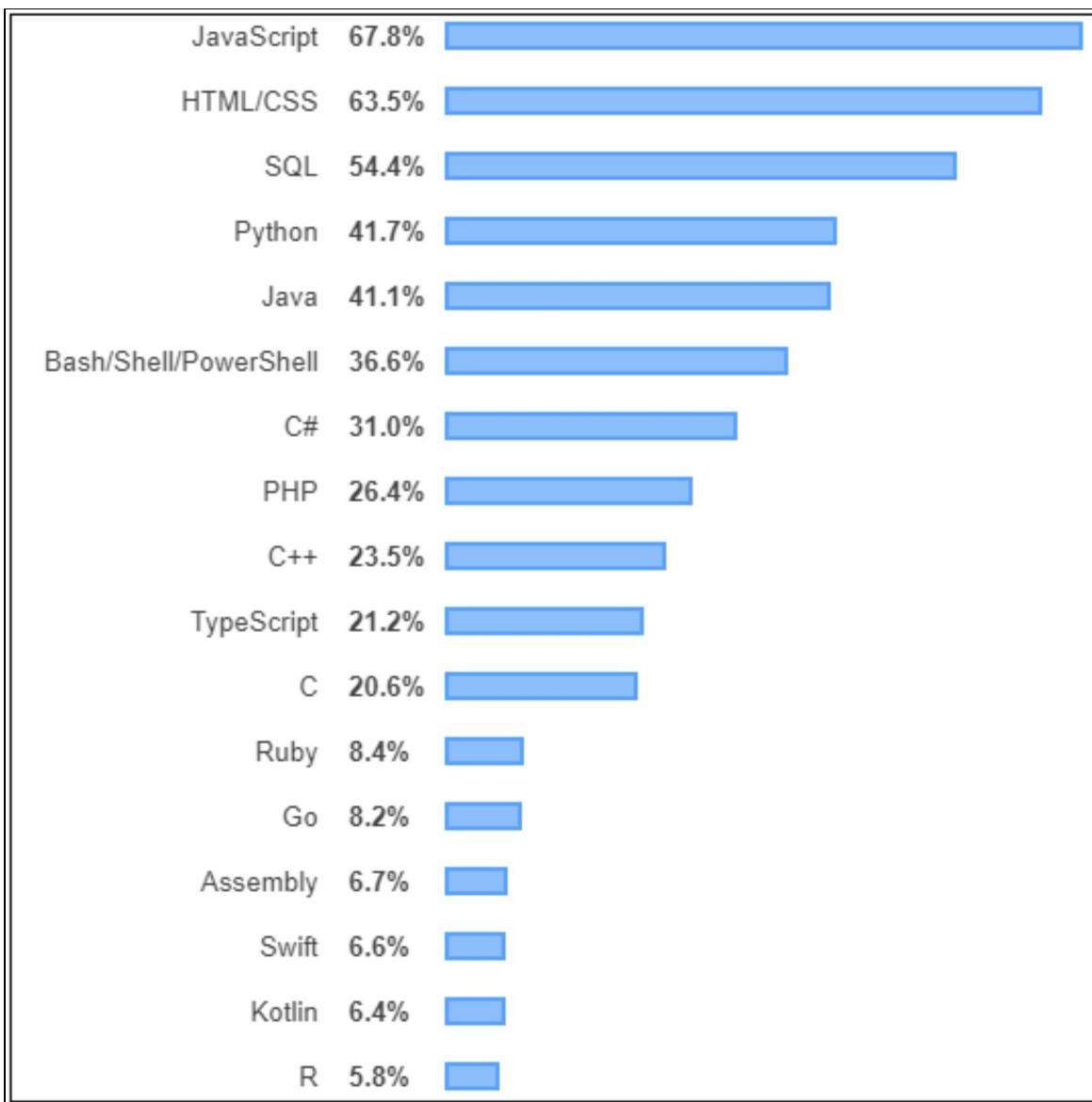


Figura 1.4 – Ranking de linguagens de programação de atividades nos repositórios do *Stackoverflow*.

O GitHub<sup>[4]</sup>, que é conhecido como a rede social do programador e facilita nossas vidas com um serviço de gerenciamento de versões de códigos, coloca o Python como a 2<sup>a</sup> linguagem com maior número de *commits* e projetos em desenvolvimento. Caso o leitor não conheça o GitHub, chegou a hora de ir pesquisar no Google algo a respeito e abrir um conta.

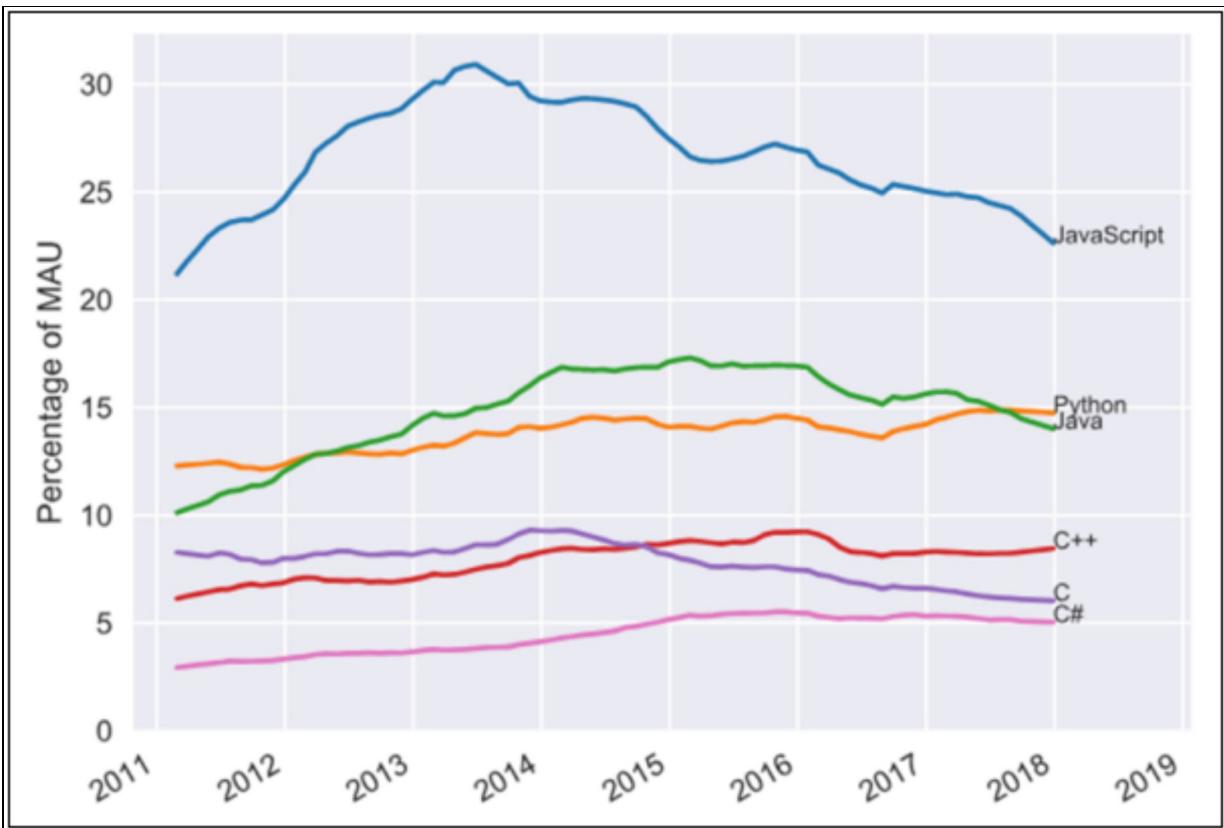


Figura 1.5 - Linguagens mais populares do Github.

Aqui vão mais algumas razões da tão grande popularidade da linguagem Python:

- É uma linguagem que suporta múltiplos paradigmas de programação. Isso quer dizer que com o Python você pode fazer programas/algoritmos complexos utilizando o paradigma procedural, funcional e até mesmo orientado a objetos.
- Os comandos e estruturas de programação são curtos. Python possui uma sintaxe bastante simplificada capaz de expressar comandos complexos escrevendo poucas linhas de código. Esse tipo de estrutura possibilita aumentar a produtividade na implementação e manutenção dos códigos. A interpretação ou entendimento do código também é bastante fácil. Python quer que você gaste tempo com a solução do problema e não com a transcrição do mesmo para as máquinas entenderem. Ela

possui tipagem dinâmica que simplifica muito o desenvolvimento dos códigos.

- c) Fornece uma enorme quantidade de bibliotecas de códigos (mais de 211 mil em janeiro de 2020<sup>[5]</sup>), cuidadosamente desenvolvidos, para trabalhar com as mais variadas áreas do conhecimento. Em tópicos seguintes, iremos conhecer algumas das principais bibliotecas do Python.
- d) Python fornece excelentes *frameworks* para trabalhar com o desenvolvimento web tais como Django, web2py, TurboGears, CubicWeb, Reahl, dentre outros. Esses *frameworks* ajudam os programadores a criar suas aplicações sem a necessidade de começar do zero. O programador se torna um jogador de quebra-cabeça que precisa encontrar as peças adequadas e colocá-las em seus devidos lugares.
- e) A linguagem Python permite a criação de interfaces gráficas com o usuário, as chamadas GUI (*Graphical User Interface*) que podem ser utilizadas nos principais sistemas operacionais do mercado como o Windows, Mac OX, Linux e Unix.
- f) Com Python podemos manipular e analisar, com extrema facilidade e agilidade, dados com diversas estruturas. Por exemplo, o analista de dados precisa lidar frequentemente com o uso de séries temporais, que é bastante facilitado a partir do uso das bibliotecas: Pandas, Scipy e Numpy.
- g) Pode ser utilizada para desenvolvimento de códigos para dispositivos móveis (*Mobile App Development*). Por exemplo, *frameworks* como o Kivy permite ao programador desenvolver com agilidade aplicativos para dispositivos móveis iOS, Android e Windows Phone.
- h) Possui o *Garbage Collection* que gerencia o esvaziamento de memórias das variáveis e classes sem a necessidade de especificação por parte dos usuários.
- i) E, para fechar com chave de ouro, alguns desses principais benefícios de utilização da linguagem Python, temos uma enorme comunidade desenvolvedora, bastante ativa e atenciosa. Através de cooperação mútua, a comunidade torna

a linguagem largamente documentada e continuamente aprimorada. Tudo isso por ser um projeto *Open Source*<sup>[6]</sup>.

## Versões do Python

Atualmente, janeiro de 2020, estamos com a versão do Python 3.8.1. Iremos utilizar a versão 3.7 para os estudos deste livro. A diferença da versão 3.7 para a 3.8 são praticamente inexistentes para os nossos estudos de fundamentos da linguagem. Infelizmente, várias utilidades das versões anteriores 2.6 e 2.7 não funcionam nesta versão 3.7 ou 3.8. Portanto, muito cuidado com a versão instalada na sua máquina, caso já tenha instalado.

Então o leitor, para acompanhar os exemplos do livro sem grandes problemas, preferencialmente, deverá utilizar a versão 3.x (3. *alguma coisa*). No entanto, isso não impede a utilização de versões anteriores, desde que o leitor tenha conhecimento das diferenças entre as versões.

Aqui estão algumas das versões passadas, com seus respectivos anos de lançamento:

Python 0.9.0 released in 1991 (*first release*)

Python 1.0 *released* em 1994

Python 2.0 *released* em 2000

Python 2.6 *released* em 2008

Python 2.7 *released* em 2010

Python 3.0 *released* em 2008

Python 3.3 *released* em 2010

Python 3.4 *released* em 2014

Python 3.5 *released* em 2018

Python 3.8 *released* em 2019

## Ambiente de Desenvolvimento

Um IDE (*Integrated Development Environment*) ou ambiente integrado de desenvolvimento de linguagem de programação é bastante utilizado pela praticidade e dinamicidade da criação dos algoritmos. Atualmente, temos acesso a uma infinidade de IDEs gratuitos e pagos.

Não vale a pena discutir sobre o melhor IDE para se programar. Cada programador vai ter seu preferido e defendê-lo com unhas e dentes. Todavia, nosso interesse com esse livro é mais conceitual, ou seja, não iremos desenvolver projetos complexos que necessitem de um ambiente para estruturar e organizar devidamente o nosso projeto. Por isso, estaremos deixando de lado o uso de um IDE mais completo.

Contudo, para fins de conhecimento, é importante que saibamos os principais e mais utilizados IDEs do mercado. Vamos a eles:

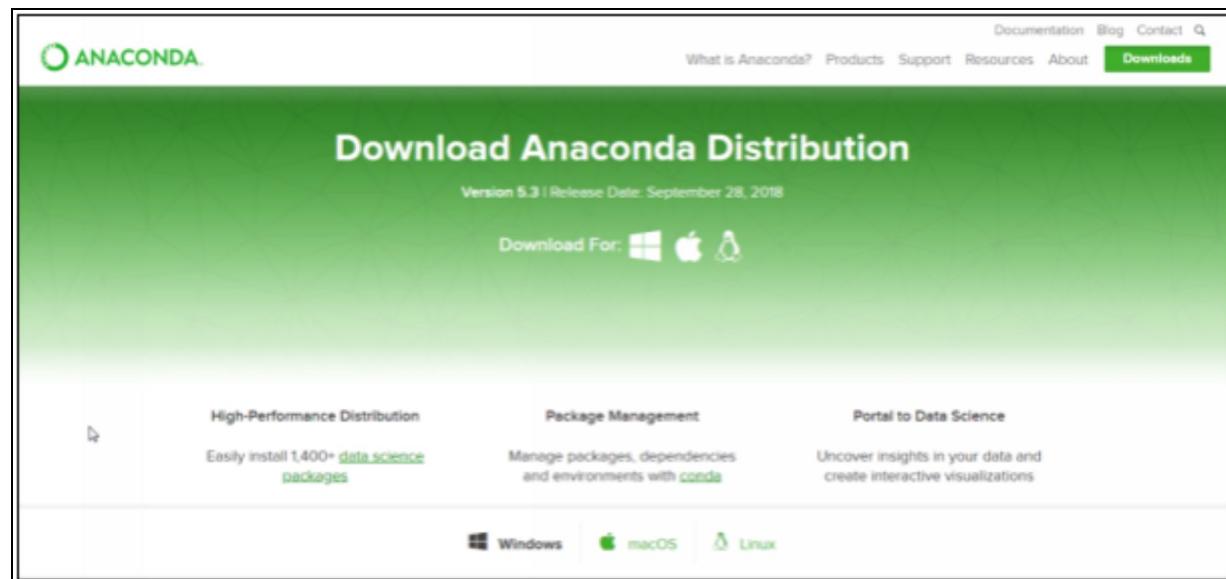
- Anaconda - Spyder (<https://anaconda.org/anaconda/python>)
- PyCharm (<https://www.jetbrains.com/pycharm/>) {amplamente utilizado no mercado}
- Sublime (<https://www.sublimetext.com/>)
- Atom (<https://atom.io/packages/ide-python>)
- PyDev (<http://www.pydev.org/>)
- Rideo (<http://rideo.yhat.com>) {espécie de versão do R Studio para Python}

Após o *download* e instalação do Python 3.7, iremos ter acesso ao IDLE (Python GUI) que é uma interface gráfica bastante útil para nos ajudar no desenvolvimento de projetos pequenos e médios. Assim, para os objetivos deste e-book, o IDLE já seria suficiente. Mas iremos mostrar como instalar um IDE mais completo, no caso iremos utilizar o ambiente Anaconda<sup>[7]</sup>.

## Download e Instalação do Anaconda

O ambiente de desenvolvimento (IDE) escolhido para os estudos deste livro foi Jupyter Notebook que já vem dentro do Anaconda. O leitor pode se sentir à vontade para acompanhar os exemplos com o IDE de sua preferência. A vantagem de instalar o Anaconda é que já teremos acesso ao Python que já vem com os IDEs Spyder e Jupyter, bem como um conjunto de mais de 1.400 pacotes para trabalhar com análise de dados.

O processo de instalação do Anaconda é bastante simples. Primeiro devemos fazer o download no site: <https://www.anaconda.com/download/>



Devemos escolher a versão do Anaconda Python 3.7 ou superior e efetuar o *download* dos arquivos relativos aos bits do processador (32 bits - computadores mais antigos, ou 64 bits para os computadores mais atuais).



Feito isso, siga as etapas do gerenciador de instalação. Não se preocupe, pois, a instalação leva certo tempo. Isso porque o Anaconda é bastante pesado (atenção para o espaço em disco necessário). Portanto, esteja atento em usar uma máquina com no mínimo 4 GB de Ram e processador i3 Intel (equivalente ou superior).

# **As principais bibliotecas Python**

A linguagem Python fornece uma gama enorme de bibliotecas para trabalhar nas mais diversas áreas científicas e tecnológicas. As principais e mais utilizadas bibliotecas são:

## Básicas:

NumPy -(<http://www.numpy.org/>)

SciPy (<https://www.scipy.org/> )

Pandas (<https://pandas.pydata.org/> )

## Gráficas:

Matplotlib (<https://matplotlib.org/>)

Plotly (<https://plot.ly/python/>)

Bokeh (<https://bokeh.pydata.org/en/latest/>)

VPython - para gráficos animados em 3D (<http://vpython.org/>)

## Álgebra simbólica:

Sympy (<http://www.sympy.org/pt/index.html>)

## Estatística:

Statsmodels (<https://www.statsmodels.org>)

Scikit-Learn (<http://scikit-learn.org/stable/>)

PyMC - para análise Baysiana de Dados (<https://github.com/pymc-devs>)

Pystan (<http://pystan.readthedocs.io/en/latest/>)

Keras (<https://keras.io/>)

## Outras

NetworkX (<https://networkx.github.io/>)

Wakari - para desenvolvimentos em computação na nuvem  
(<https://anaconda.org/wakari>)

PyCuda, PyOpenCL - processamento paralelo  
(<https://documentacion.de/pycuda/>)

Theano - processamento paralelo e otimização  
(<http://deeplearning.net/software/theano/>)

CVXPY - otimização convexa com Python (<http://www.cvxpy.org/>)

PyTables - manipulação de dados em tabelas  
(<https://www.pytables.org/>)

Numba - permite que o Python execute processos em velocidade de código em linguagem de máquina nativa (baixo nível)  
(<https://numba.pydata.org/>)

Jupyter - permite utilizar o Python diretamente de seu browser  
(<http://jupyter.org/>)

Tensor Flow - ampla biblioteca para trabalhar com inteligência artificial (<https://www.tensorflow.org/?hl=pt-br>)

## Como instalar pacotes

Este subtópico é muito importante, pois retrata uma dúvida recorrente de muitos usuários Python: a instalação de novos pacotes e bibliotecas. Caso você não tenha escolhido o Anaconda para acompanhar esse livro e esteja utilizando apenas o Python puro, em alguns momentos será preciso instalar algumas bibliotecas. O processo de instalação pode se dar a partir de diversos caminhos. Tudo vai depender de como as **variáveis de ambiente** estão configuradas na sua máquina.

Para minimizar problemas desse tipo, caso você não esteja por dentro do que está sendo dito, siga as etapas descritas a seguir quando precisar instalar alguma nova biblioteca no Python. Vamos pegar um exemplo de instalação da biblioteca NumPy (lembmando que esta já vem previamente instalada com o Anaconda). Estaremos aqui fazendo uma demonstração didática.

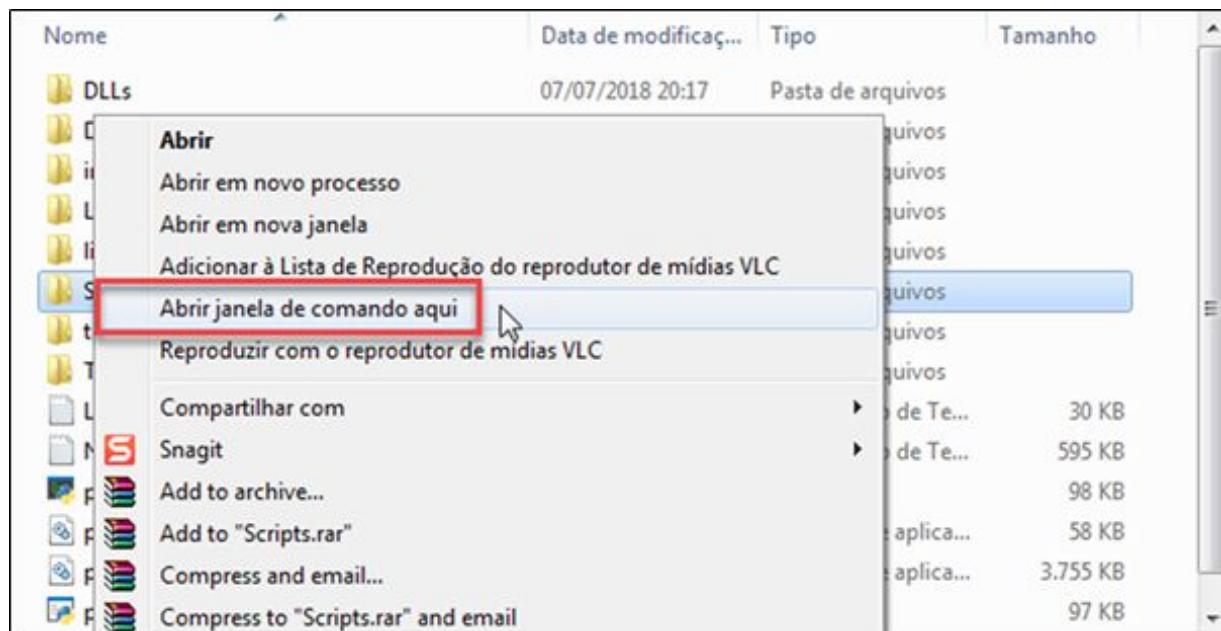
Podemos fazer a instalação do Numpy no Python 3.7 de diversas maneiras. Muitas vezes instalar pacotes e bibliotecas pode se tornar um problema. Mas você irá aprender uma maneira muito simplificada e rápida utilizando o **pip**. Assim, caso tenhamos mais de uma versão do Python instaladas, o modo de instalação que iremos conhecer a seguir irá facilitar bastante nossas vidas.

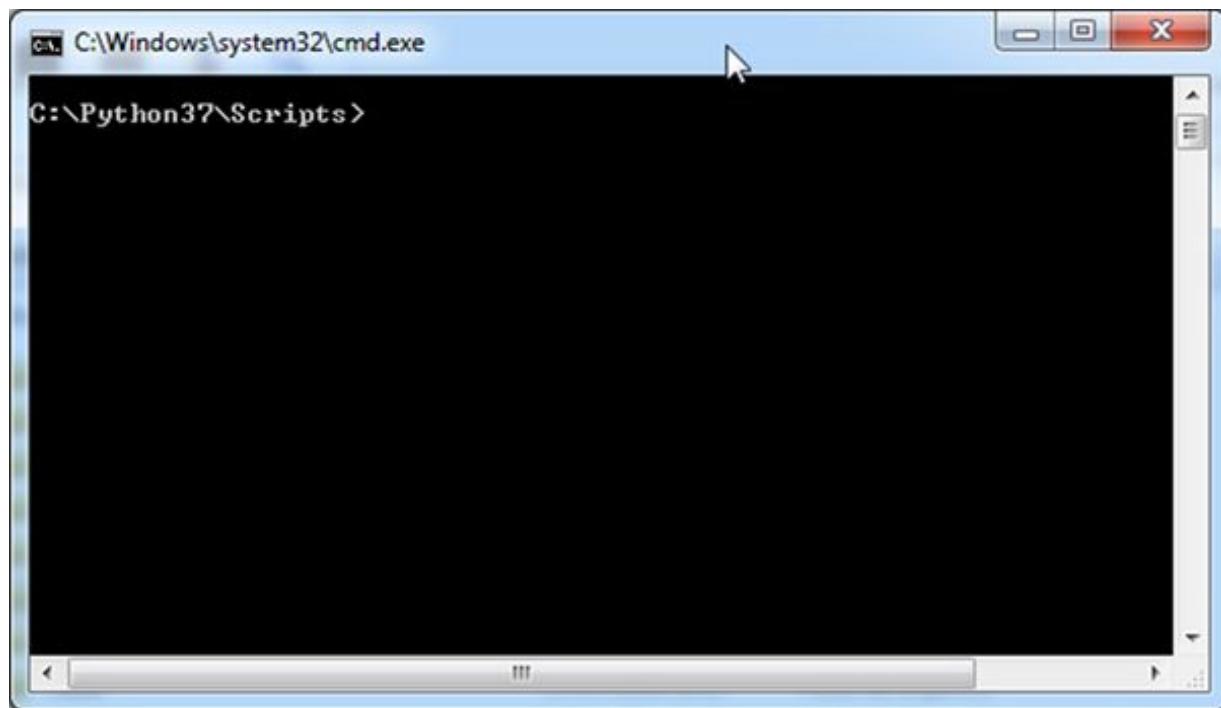
Vamos seguir os seguintes passos:

- 1 - Vamos ao diretório onde instalamos o Python 3.7. No nosso caso de exemplo foi: **C:\Python37**

Nome	Data de modificaç...	Tipo	Tamanho
DLLs	07/07/2018 20:17	Pasta de arquivos	
Doc	07/07/2018 20:17	Pasta de arquivos	
include	07/07/2018 20:16	Pasta de arquivos	
Lib	07/07/2018 20:17	Pasta de arquivos	
libs	07/07/2018 20:17	Pasta de arquivos	
Scripts	07/07/2018 20:18	Pasta de arquivos	
tcl	07/07/2018 20:17	Pasta de arquivos	
Tools	07/07/2018 20:17	Pasta de arquivos	
LICENSE	27/06/2018 05:03	Documento de Te...	30 KB
NEWS	27/06/2018 05:03	Documento de Te...	595 KB
python	27/06/2018 05:01	Aplicativo	98 KB
python3.dll	27/06/2018 05:00	Extensão de aplica...	58 KB
python37.dll	27/06/2018 05:00	Extensão de aplica...	3.755 KB
pythonw	27/06/2018 05:01	Aplicativo	97 KB

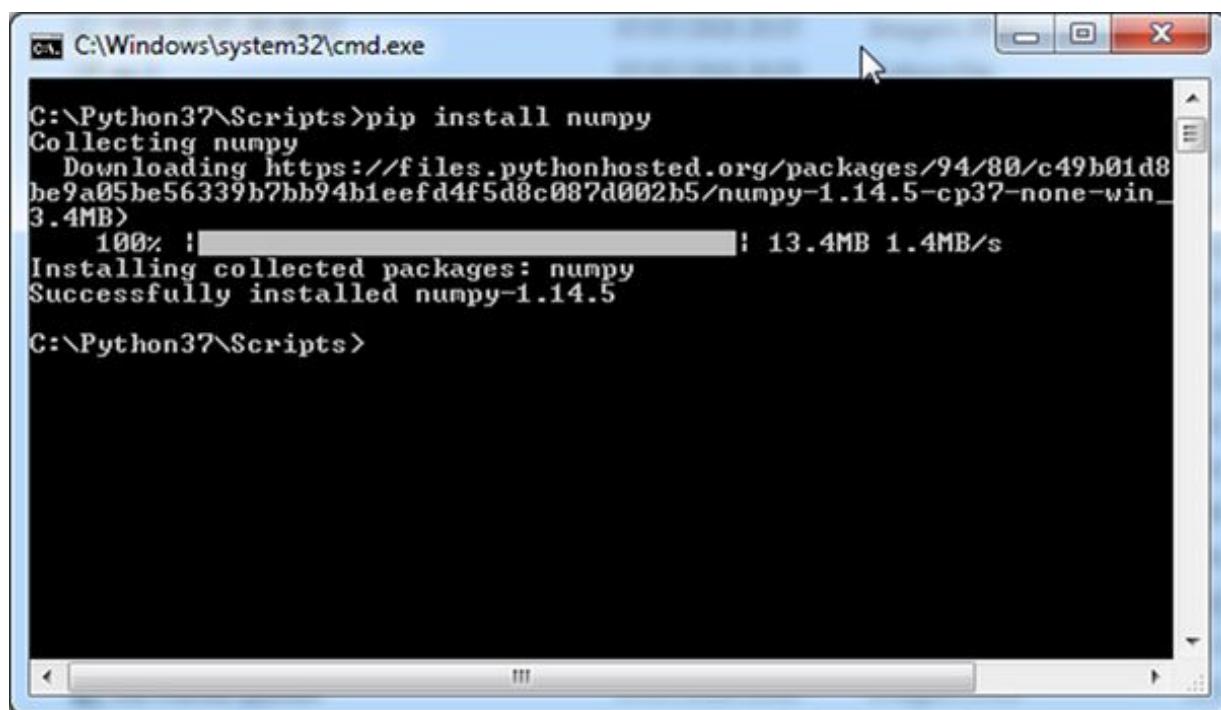
2 - Com o botão direito e segurando a tecla *shift* do teclado vamos clicar em cima da pasta Scripts e clicar em **Abrir janela de comando aqui**, como mostra a figura abaixo:





3 – Em seguida vamos dar o seguinte comando nesta janela: >> ***pip install numpy***

E clicar **Enter**:



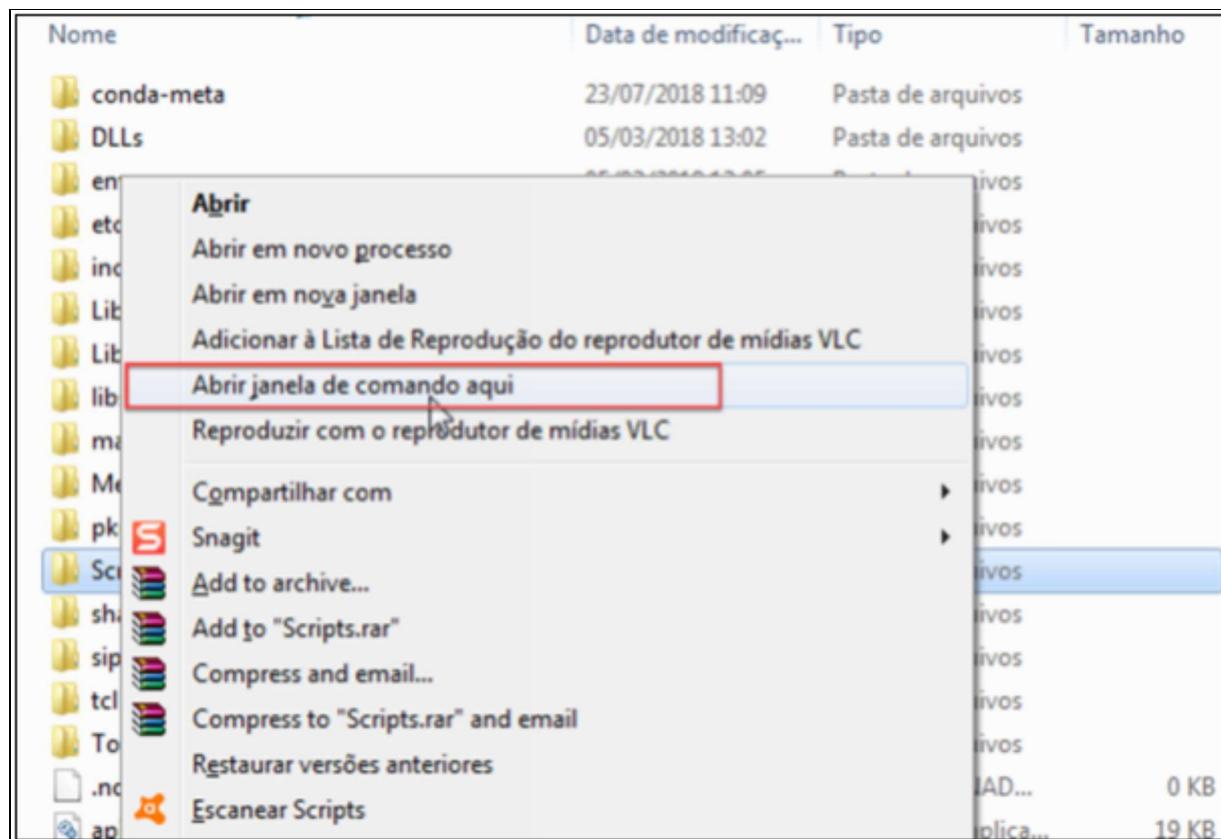
Tudo certo. Agora temos nossa biblioteca **NumPy** devidamente instalada. Vamos conhecer agora algumas funcionalidades do Python para facilitar a compreensão dos capítulos subsequentes.

## Jupyter Lab

Com a instalação do Anaconda também teremos acesso ao Jupyter Lab, porém numa versão desatualizada. Vamos agora aprender como atualizar a versão do Jupyter Lab.

Primeiro vamos até a pasta raiz de instalação do Anaconda para garantir a instalação correta.

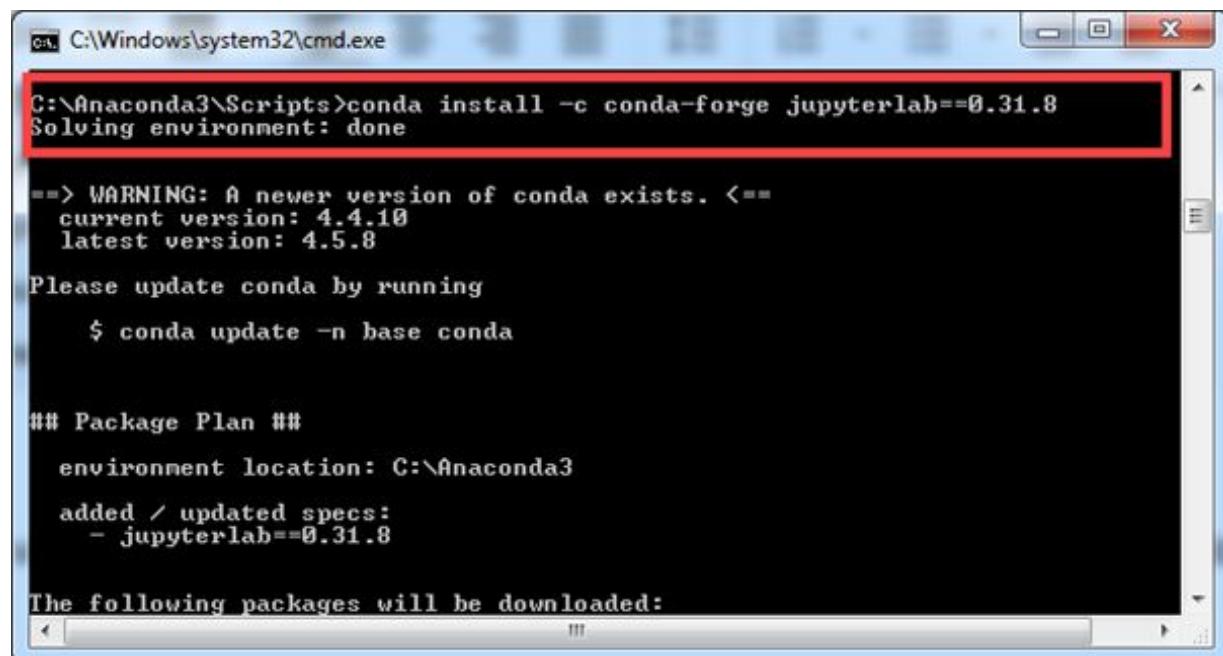
Nós escolhermos o seguinte diretório: **c:\Anaconda3** dentro desse diretório temos uma pasta chamada **Scripts**. Vamos segurar a tecla **shift** do teclado e clicar com o botão direito em cima dessa pasta pedindo para **Abrir janela de comando aqui**, como mostra a figura abaixo:



Logo em seguida vamos atualizar o nosso Jupyter Lab a partir do seguinte comando no terminal Windows: `conda install -c conda-`

*forge jupyterlab==0.31.8*

Aperte **Enter**. Depois, quando for requerido, aperte a tecla **[y]** e aguarde a instalação:



C:\Windows\system32\cmd.exe

```
C:\Anaconda3\Scripts>conda install -c conda-forge jupyterlab==0.31.8
Solving environment: done

=> WARNING: A newer version of conda exists. <=-
  current version: 4.4.10
  latest version: 4.5.8

Please update conda by running

$ conda update -n base conda

## Package Plan ##
environment location: C:\Anaconda3
added / updated specs:
- jupyterlab==0.31.8

The following packages will be downloaded:
```

## Funcionalidades do Jupyter Notebook

O Jupyter Notebook é um projeto *open-source* que se propõe a utilizar linguagens de programação a partir de um navegador de internet (*browser*). Assim, temos a opção de trabalhar com o desenvolvimento de aplicações e *scripts* com o navegador Chrome, por exemplo.

A palavra Jupyter surgiu a partir da junção das iniciais das palavras das linguagens: **Julia**, **Python** e **R**, que atualmente são as mais utilizadas pelos cientistas, analistas e engenheiros de dados.



Com o Jupyter Notebook podemos criar e compartilhar documentos contendo códigos, equações, gráficos, figuras e textos de maneira simples e rápida. Um navegador de internet é o recurso principal necessário. Hoje em dia todo mundo utiliza pelo menos um navegador de internet. Por isso a grande facilidade de compartilhamento.

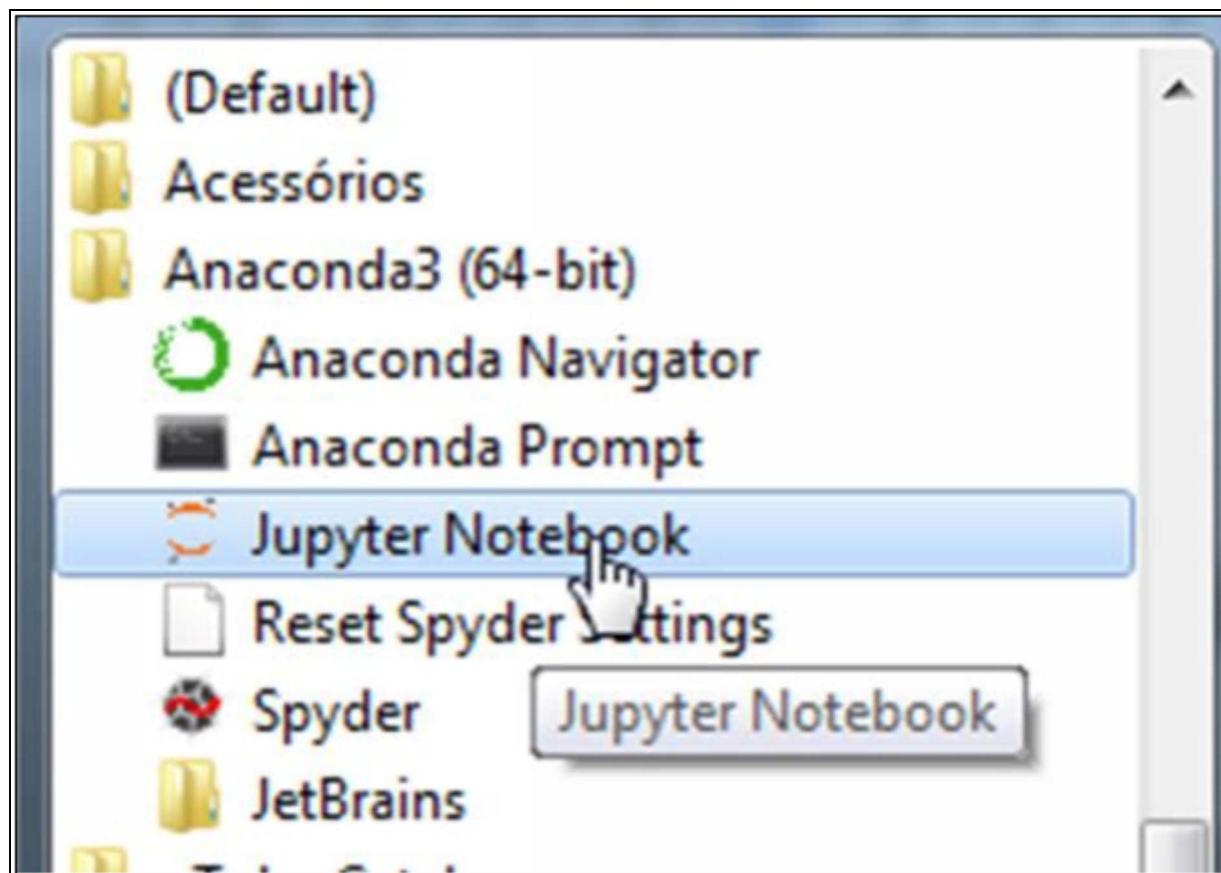


Esse projeto vem ganhando destaque entre os desenvolvedores do mundo inteiro devido a facilidade de criar e compartilhar tutoriais com simplicidade e clareza. Para saber mais a respeito desse excelente IDE entrar em <http://jupyter.org/documentation>.

Vamos conhecer agora as principais funcionalidades do Jupyter Notebook.

### Abrindo um serviço local no Jupyter

Quando instalamos o Anaconda em nossas máquinas o Jupyter Notebook (JN) também é automaticamente instalado. Podemos ir ao Menu iniciar e procurar por Anaconda:

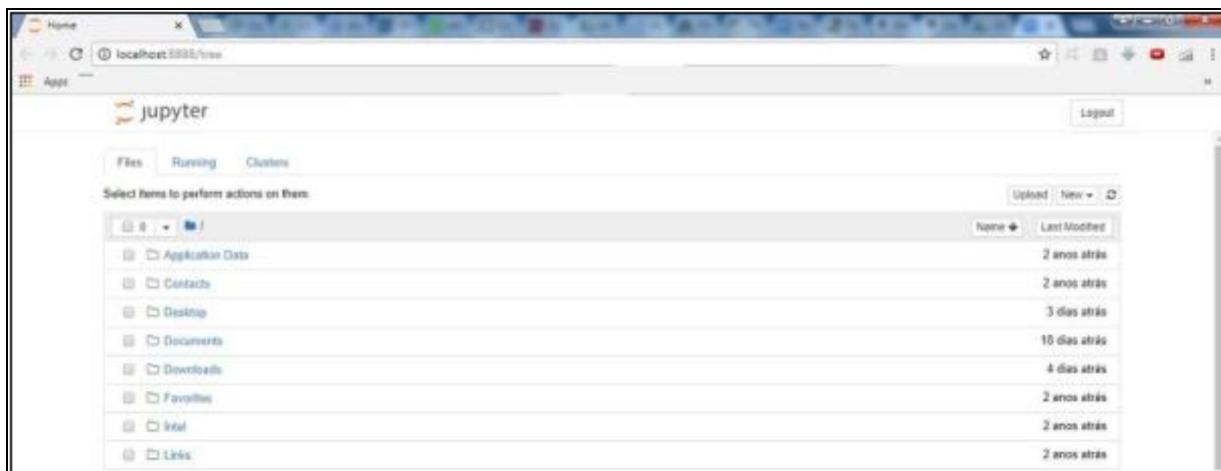


Após clicarmos, iremos ter acesso a um servidor local  
(<http://localhost:8888/> ).

```
[I 12:16:42.473 NotebookApp] JupyterLab beta preview extension loaded from C:\Anaconda3\lib\site-packages\jupyterlab
[I 12:16:42.490 NotebookApp] JupyterLab application directory is C:\Anaconda3\share\jupyter\lab
[I 12:16:43.232 NotebookApp] Serving notebooks from local directory: C:\Users\Computer
[I 12:16:43.232 NotebookApp] 0 active kernels
[I 12:16:43.233 NotebookApp] The Jupyter Notebook is running at:
[I 12:16:43.233 NotebookApp] http://localhost:8888/?token=1e6d4b39423f03eb9289cc3432c915f3b6c8712eb085d42
[I 12:16:43.233 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
[C 12:16:43.265 NotebookApp]

      Copy/paste this URL into your browser when you connect for the first time,
      to login with a token:
          http://localhost:8888/?token=1e6d4b39423f03eb9289cc3432c915f3b6c8712eb085d42
[I 12:16:44.477 NotebookApp] Accepting one-time-token-authenticated connection from ::1
```

A depender do seu navegador padrão, a janela mostrada abaixo será automaticamente exibida:



Se seu navegador for o Internet Explorer (IE) do Windows tente mudar para o Chrome, pois vários problemas são reportados quando utilizamos o Jupyter a partir do IE. Portanto, para os estudos deste livro, iremos utilizar o navegador Chrome.

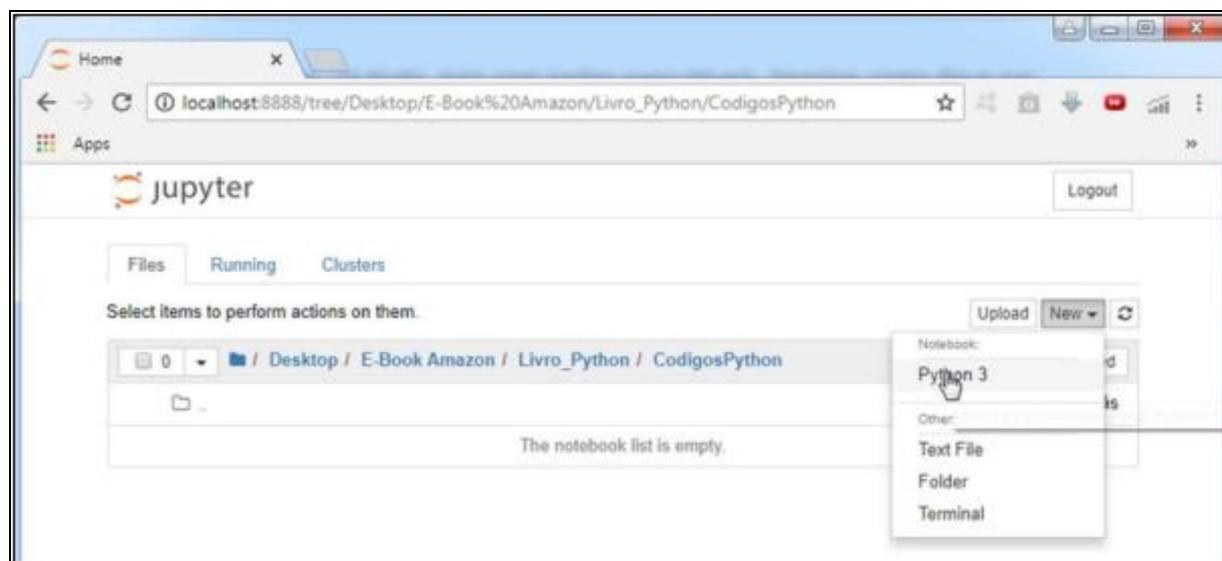
É importante que você crie um diretório em sua área de trabalho para irmos colocando os códigos. Uma vez criado o diretório, basta navegar até ele clicando nos campos mostrados na figura acima.

Podemos também abrir o Jupyter a partir do terminal de comandos do Windows. Para isso, basta digitar no terminal o comando:

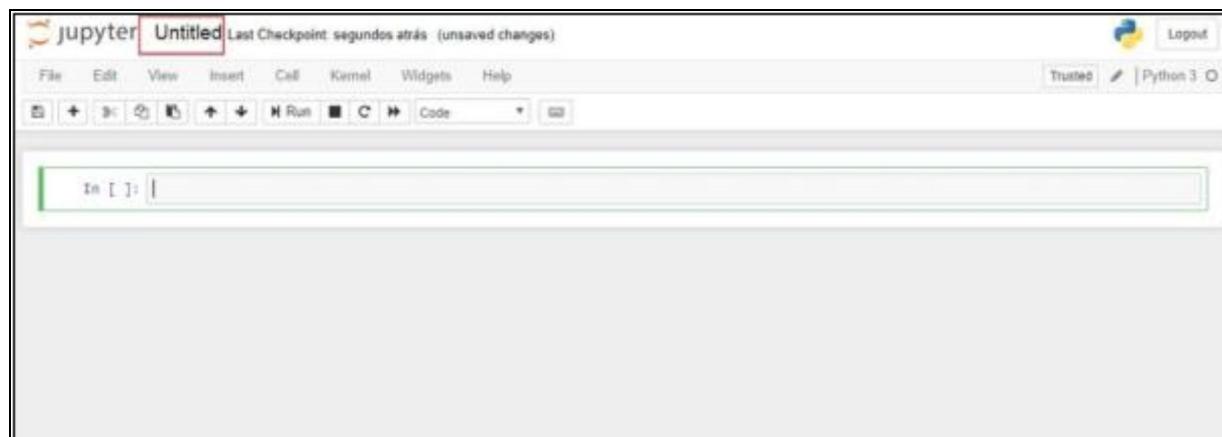
`> jupyter notebook.`

**Atenção:** caso você tenha problemas para abrir o Jupyter dessa maneira, significa que as variáveis de ambiente para o Jupyter não estão devidamente configuradas. Você então precisará ir ao painel de controle, configurações avançadas e variáveis de ambiente para fazer a configuração. Para mais detalhes com relação a isso faça uma pesquisa no Google: **como adicionar variáveis de ambiente no Windows**.

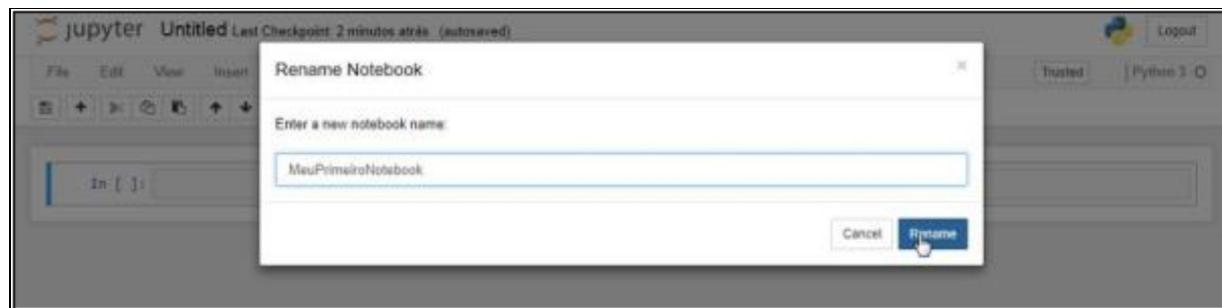
Vejamos como criar um arquivo do Jupyter (extensão .ipynb) dentro do diretório: Desktop/E-book Amazon/Livro\_Python/CodigosPython:



Vale ressaltar que se tivermos outras linguagens de programação instaladas (kernels), como, por exemplo, o R, iremos ter a opção de abrir um servidor local Jupyter Notebook com essa linguagem. No nosso caso temos apenas o Python 3 instalado, por isso apenas essa opção nos foi oferecida. Assim, após escolher um novo arquivo Python 3, teremos acesso aos seguintes campos mostrados abaixo:



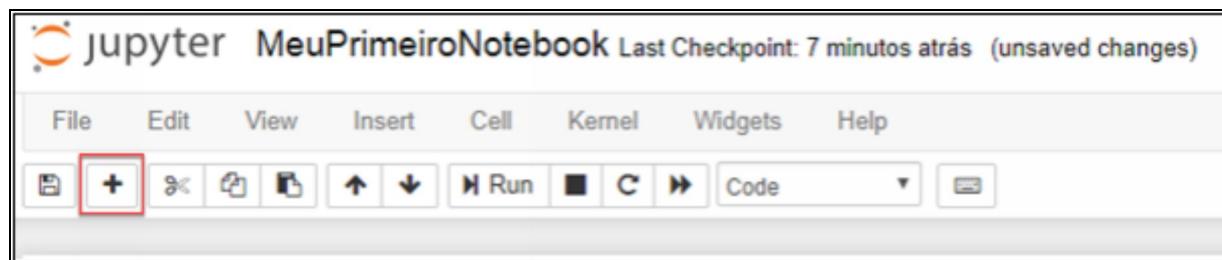
Podemos renomear o arquivo clicando no campo destacado na figura acima.



Agora é muito importante que você vasculhe entre os menus do Jupyter. É comum que após a instalação de um programa deixemos de lado a exploração dos menus. Devemos ter a curiosidade de conhecer bem nossa ferramenta de trabalho para ganharmos produtividade. Por isso, vale muito a pena gastar alguns minutinhos vasculhando o campo do Menu.

Essas ferramentas são sempre construídas pensando no usuário, ou seja, certamente foi pensada e desenvolvida para tornar sua vida de programador e estudante ainda mais fácil.

Podemos digitar os códigos nas células criadas a partir de um símbolo de mais (+) mostrado abaixo:

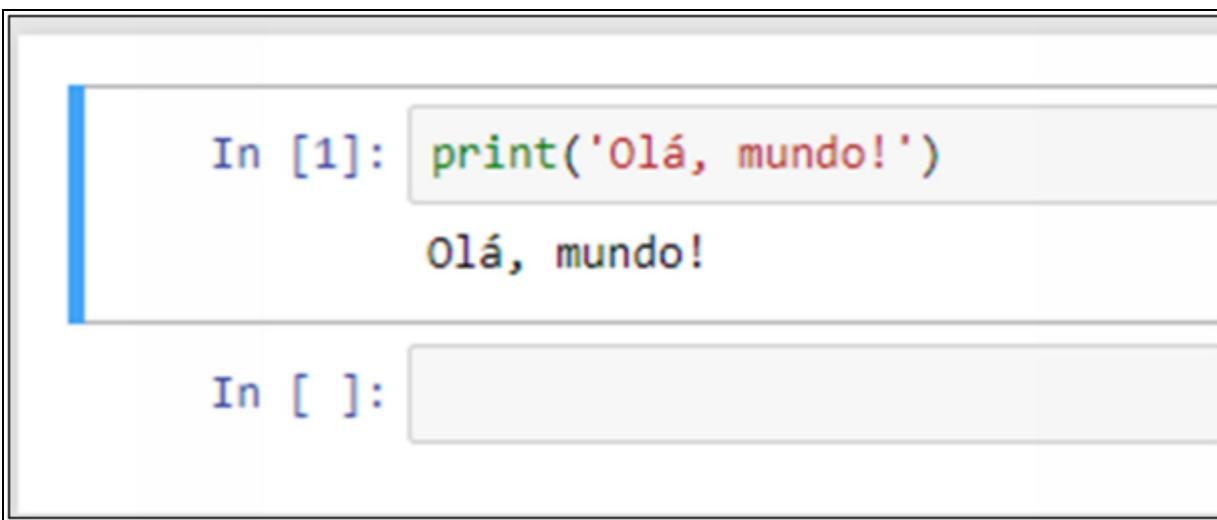


Ao clicarmos neste botão, iremos ter acesso a novas células para digitarmos nossos códigos. Como você pode ter notado uma célula inicial já é criada no momento em que pedimos um novo arquivo. O

que vou dizer agora você pode não conseguir visualizar na figura (devido falta de cores no **Kindle**), mas quando estiver utilizando seu computador para a leitura, poderá notar que as células no modo de edição (quando estamos digitando nossos códigos) ficam com a cor verde nas bordas. Veja figura abaixo:

A screenshot of a Jupyter Notebook interface. The title bar says "jupyter MeuPrimeiroNotebook Last Checkpoint: 11 minutes atrás (unsaved changes)". The menu bar includes File, Edit, View, Insert, Cell, Kernel, Widgets, Help. On the right, there are buttons for Trusted, Python 3, and Logout. Below the menu is a toolbar with various icons. A code cell is active, containing the Python code "In [ ]: print('Olá, mundo!')". The cell has a green border, indicating it is in edit mode.

Podemos digitar nossos códigos Python nela e pedir para que sejam executados a partir do botão ‘Run’ ou com o atalho do teclado (**Shift + Enter**). Após isso, teremos o seguinte resultado:

A screenshot of a Jupyter Notebook showing the execution result. The code cell from the previous image is still present. Below it, the output cell shows the result of the execution: "In [1]: print('Olá, mundo!')\nOlá, mundo!" The "In [1]" label is blue, and the output text is colored (Olá is blue, mundo! is orange). There is another empty code cell below, labeled "In [ ]:".

Note que mudou da cor **verde** para **azul** e apareceu o número **[1]** dentro de colchetes. À medida que vamos colocando nossos códigos e pedindo para rodar com o **Run** ou **Shift+Enter** novos números em sequência irão aparecendo automaticamente.

Abaixo você terá acesso aos princípios da filosofia Python, a partir do '*import this*'. Faça uma leitura cuidadosa para conhecer mais a

respeito do ‘**espírito**’ da linguagem.

```
In [1]: print("Olá, mundo!")
Olá, mundo!

In [2]: import this
The Zen of Python, by Tim Peters
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, It's a bad idea.
...
```

Foi dito a pouco que o Jupyter Notebook é uma excelente ferramenta para compartilhar tutoriais. E agora você vai entender o porquê disso. Vamos abrir um novo notebook e renomeá-lo para ‘Estudo\_notebook\_1’.

The screenshot shows a Jupyter Notebook window titled "MeuPrimeiroNotebook". The "File" menu is open, displaying options like "New Notebook", "Open...", "Make a Copy...", "Rename...", "Save and Checkpoint", "Revert to Checkpoint", "Print Preview", "Download as", "Trusted Notebook", and "Close and Halt". The "Kernel" dropdown shows "Python 3" is selected. The main area contains a code cell with the following content:

```
'Olá, mundo!'  
mundo!  
this  
en of Python, by Tim Peters  
ful is better than ugly.  
it is better than implicit.  
e is better than complex.  
x is better than complicated.  
s better than nested.  
e is better than dense.  
ility counts.  
Special cases aren't special enough to break the rules.
```

Algo muito útil para documentação de *scripts* e tutoriais é que o Jupyter aceita comando em HTML, Latex e Markdown. Isso mesmo, podemos utilizar linguagem de marcação HTML para auxiliar na estruturação e documentação dos nossos *scripts*.

Outros ambientes de desenvolvimentos de códigos permitem fazer comentários a partir de caracteres reservados como (# e "" "), no entanto, com o jupyter podemos criar células e escrever textos dentro dessas células.

Os textos escritos podem conter as linguagens de marcação do HTML e até mesmo CSS<sup>[8]</sup> (*Cascading Style Sheets*). Assim, podemos deixar o visual dos nossos Notebooks bem bonitos e atrativos para os utilizadores.

**Markdown** é uma linguagem mais simples de marcação que a HTML. Ela foi desenvolvida por John Gruber e Aaron Swartz.

Basicamente, a markdown converte textos em uma estrutura HTML que é facilmente compreendida pelo navegador.

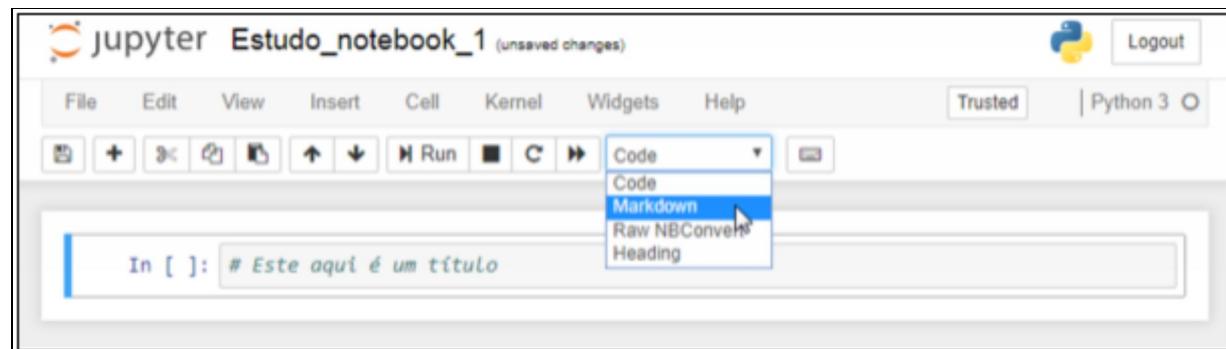
Vamos estudar agora a diferença entre **code** e **markdown**. Isso irá nos ajudar a fazer a nossa documentação dos códigos no Jupyter.

## Diferença entre Code e Markdown

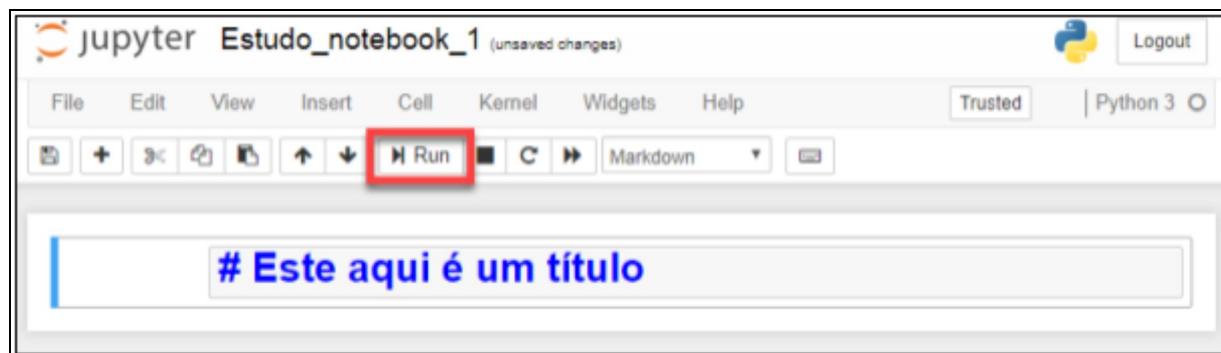
As células que podemos criar dentro do Jupyter Notebook podem ter as principais características de **code** ou **markdown**. Por padrão, as células vêm com o **code**, que é propriamente os códigos na linguagem em que estamos trabalhando. No nosso caso, estamos com a linguagem Python (Jupyter pode trabalhar com Julia e R como mencionado antes).

Por outro lado, **markdown** pode ser utilizada para que possamos escrever textos a partir de elementos visuais do próprio Jupyter ou HTML.

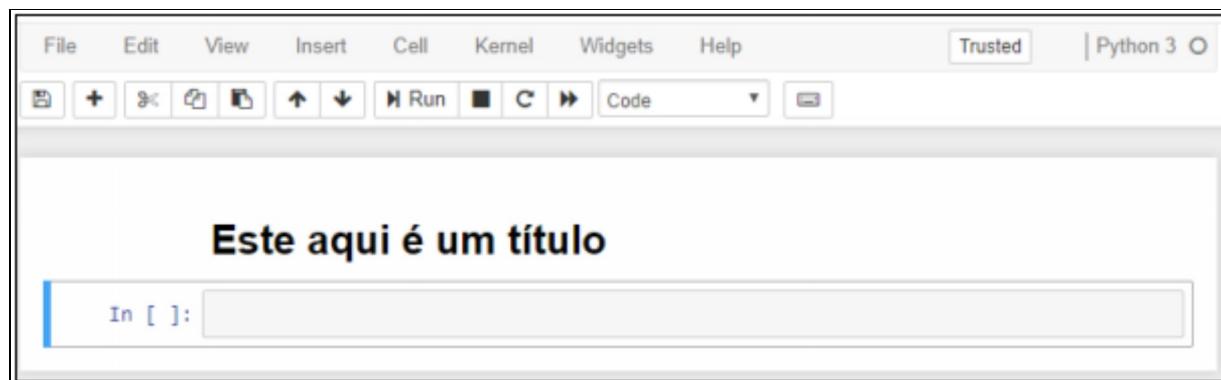
Por exemplo, veja abaixo como colocar um comentário utilizando o cerquilha (hashtag, ou jogo da velha) quando a opção da célula estiver marcada como **markdown**.



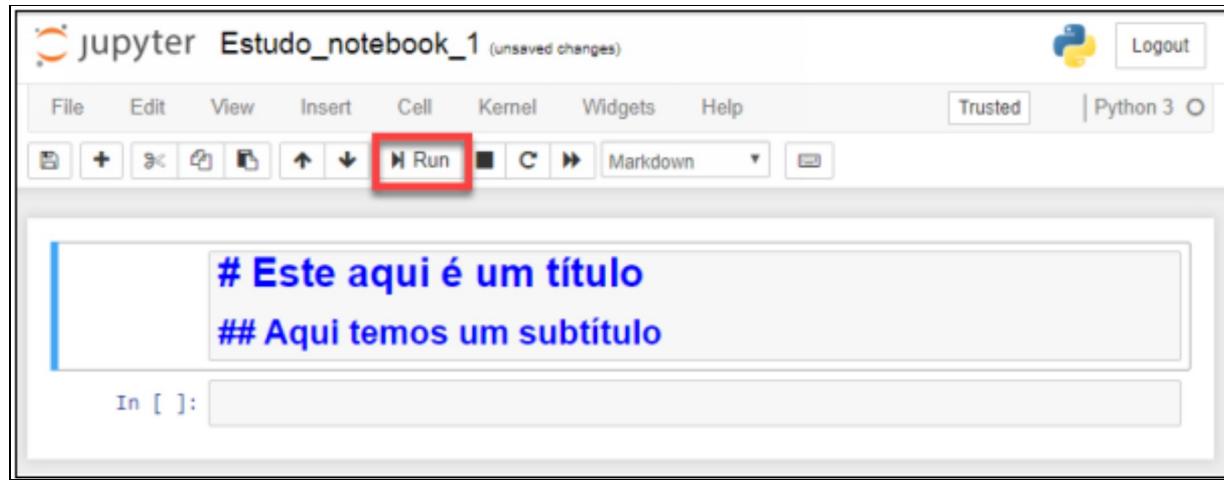
Ao definirmos a célula como **Markdown** podemos utilizar as *tags* dessa linguagem.



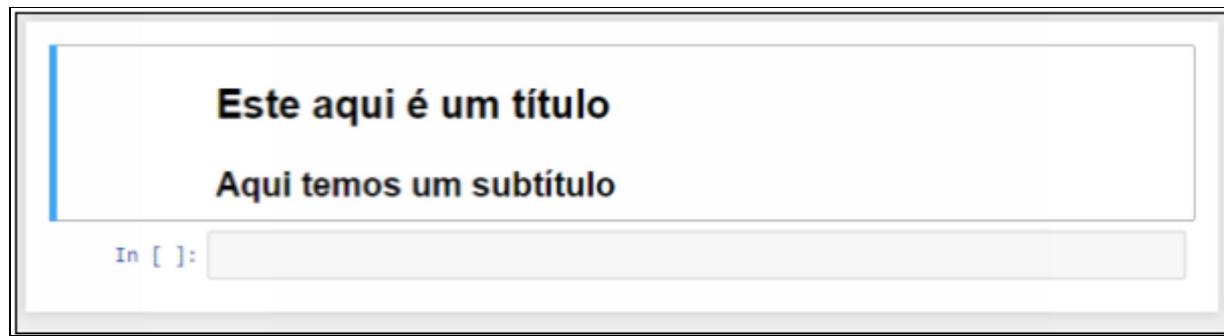
Se pedirmos para executar a célula com o **Run** ou **Shift+Enter**, teremos o seguinte resultado:



Interessante, concorda? Podemos digitar também dois ## e ter um comportamento de tamanho de texto distinto. Podemos dar dois clicks em cima da célula e alterar o seu **markdown** para:



Após executar:



Para quem conhece HTML, podemos utilizar todo o poder dessa linguagem para manipular e ornamentar nossos textos e comentários. Lembrando sempre que a célula precisa estar marcada como **Markdown**:

The screenshot shows a Jupyter Notebook interface. At the top, there's a toolbar with various icons: a file icon, a plus sign, a back arrow, a forward arrow, a refresh icon, an up arrow, a down arrow, a run icon, a stop icon, a clear icon, and a right arrow. To the right of these is a dropdown menu labeled "Markdown". Below the toolbar, there are two code cells. The first cell contains the following text:

```
# Este aqui é um título  
## Aqui temos um subtítulo
```

The second cell contains the following code and placeholder text:

```
<h1>Aqui é um título H1</h1>  
  
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor  
incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis  
nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. .  
  
<h2 style="color: red">Aqui é um título H2</h2>  
  
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor  
incididunt ut labore et dolore magna aliqua.  
  
<h3 style="color: green">Aqui é um título H3</h3>  
  
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor  
incididunt ut labore et dolore magna aliqua.
```

Após executar, temos:

The rendered output shows three levels of headings and their associated placeholder text. The first heading is in black, the second in red, and the third in green, all matching the colors used in the original code cells.

**Aqui é um título H1**

Placeholder text: Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. .

**Aqui é um título H2**

Placeholder text: Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.

**Aqui é um título H3**

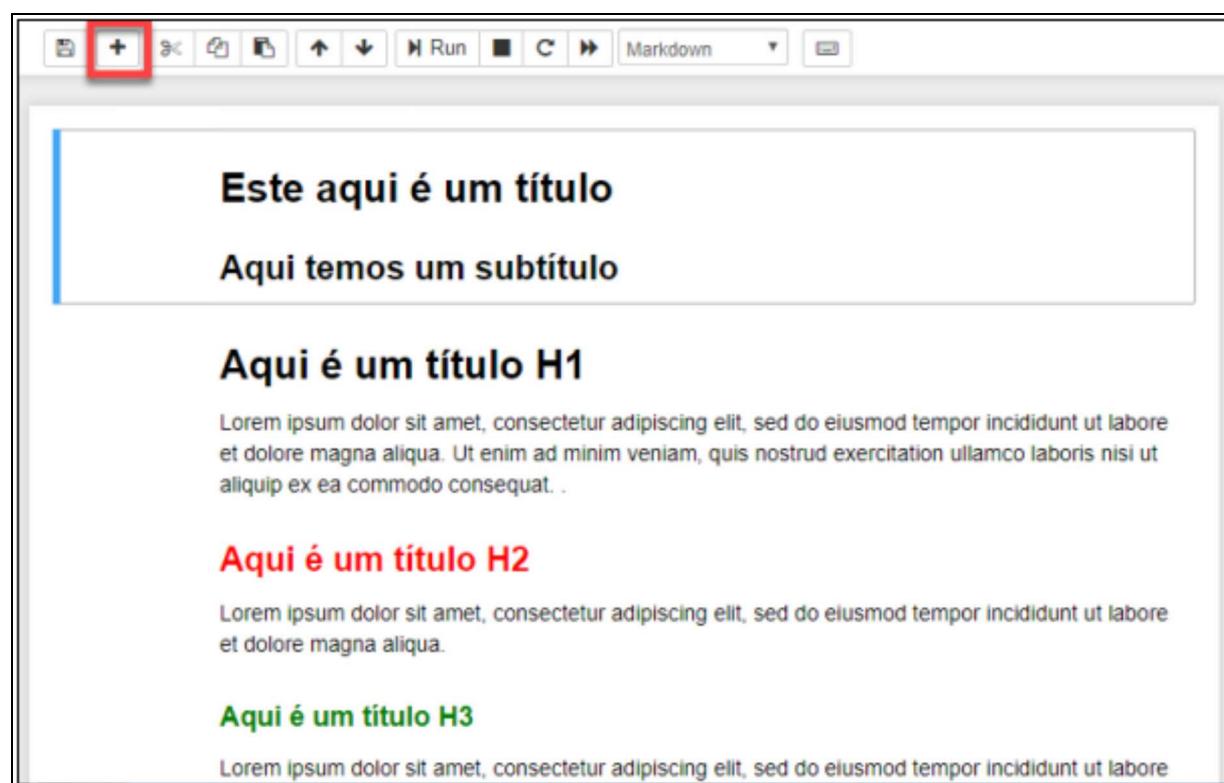
Placeholder text: Placeholder text: Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.

Com isso podemos ir inserindo texto no decorrer do nosso código com as células marcadas como **Markdown**. Os códigos com

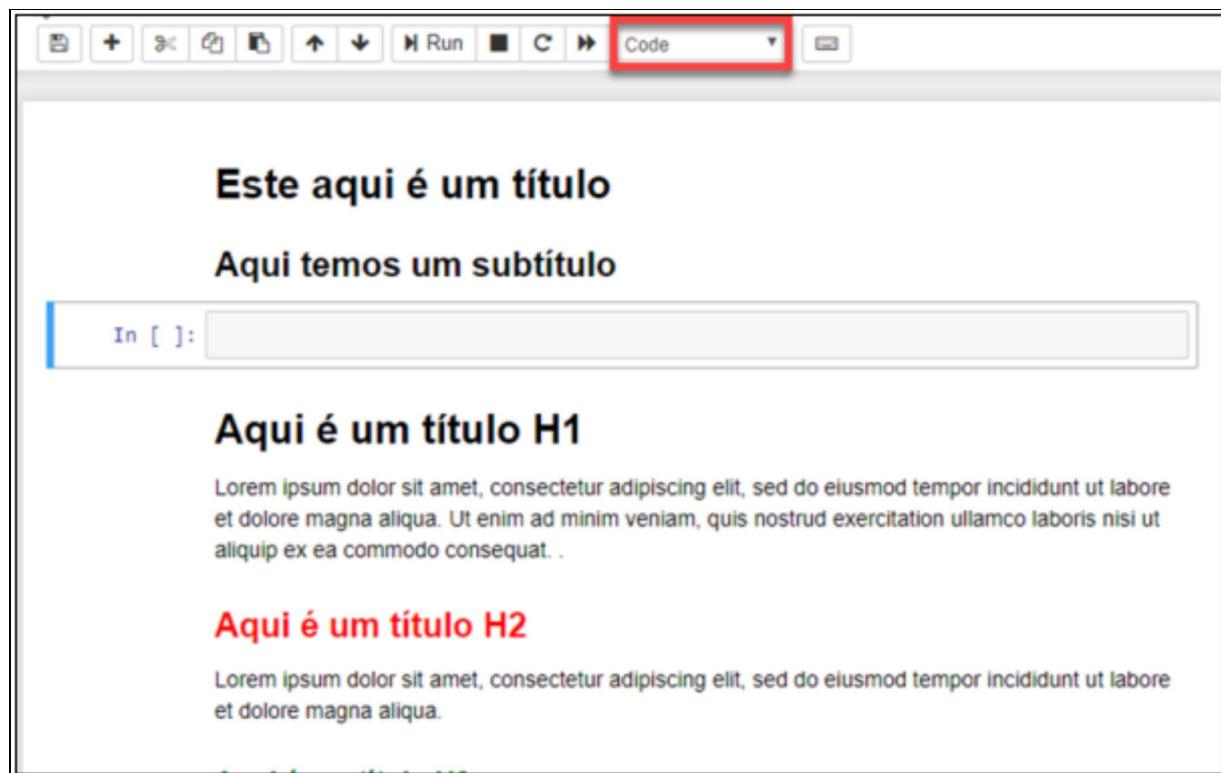
Python, claro, devemos ter a marcação **Code**, atenção a isso.

Podemos também inserir equações com a formatação do LaTeX (<https://pt.wikipedia.org/wiki/LaTeX>). Como não é nosso objetivo estudar LaTeX, os interessados podem fazer uma rápida pesquisa no Google e conhecer essa importante funcionalidade.

Vamos agora adicionar uma célula de códigos entre nossas duas colunas Markdown. Para isso devemos **clicar** na primeira célula e apertar no ‘ícone de mais’ destacado abaixo:



Devemos lembrar de modificar para **code** o tipo da célula que será criada, veja abaixo:



Podemos adicionar nosso código nessa nova célula criada uma vez que ela vem, previamente, marcada com **Code**:

As setinhas para cima e para baixo destacadas na figura abaixo permitem movimentar as células. O ícone de tesoura também destacado permite **excluir** ou remover uma célula selecionada qualquer.

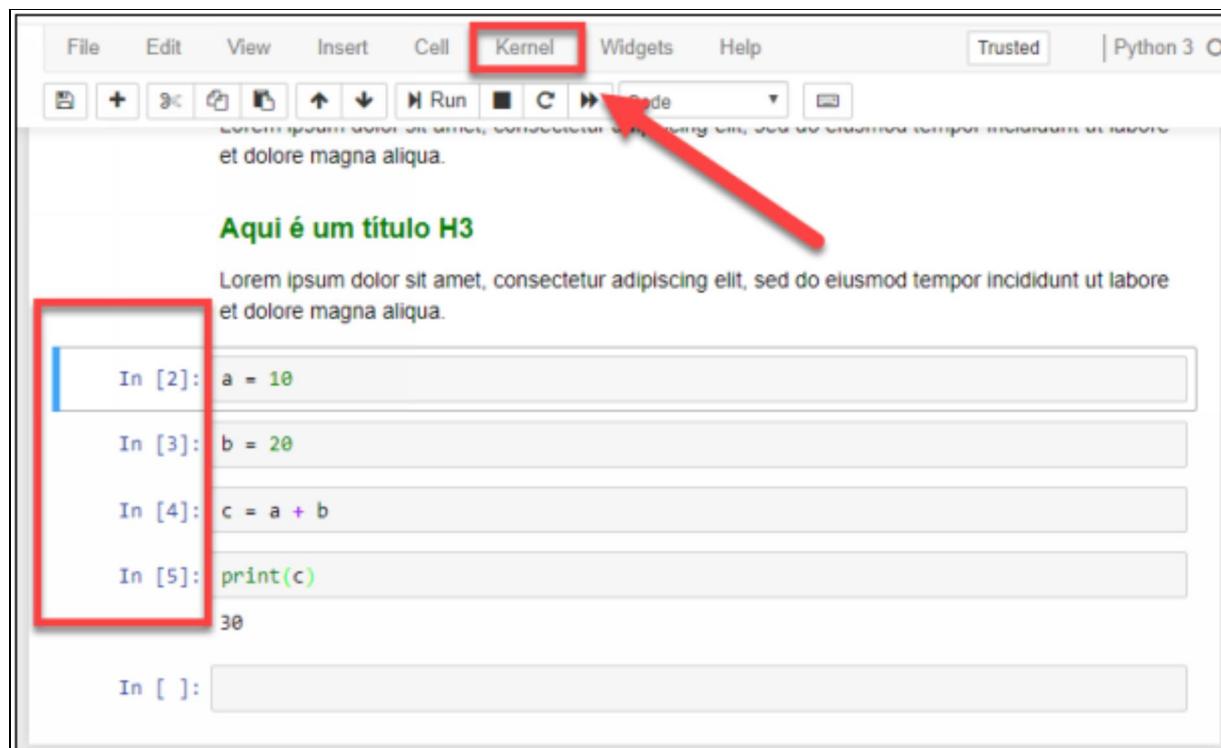
The screenshot shows a Jupyter Notebook interface. At the top, there's a toolbar with various icons: a file icon, a plus sign for new cells, a red square for running cells, a refresh icon, a cell type selector (red box), up and down arrow icons for cell navigation (also red boxed), a 'Run' button, a stop button, a 'Code' dropdown, and a help icon. A large red arrow points from the text 'Este aqui é um título' down towards the cell navigation icons. Below the toolbar, the text 'Este aqui é um título' is displayed in bold black font. Underneath it, the text 'Aqui temos um subtítulo' is also in bold black font. A code cell is shown with the input 'In [ ]:' followed by Python code: 

```
a = 77  
b = 10  
  
c = a + b  
  
print('Soma de a + b = ', c)
```

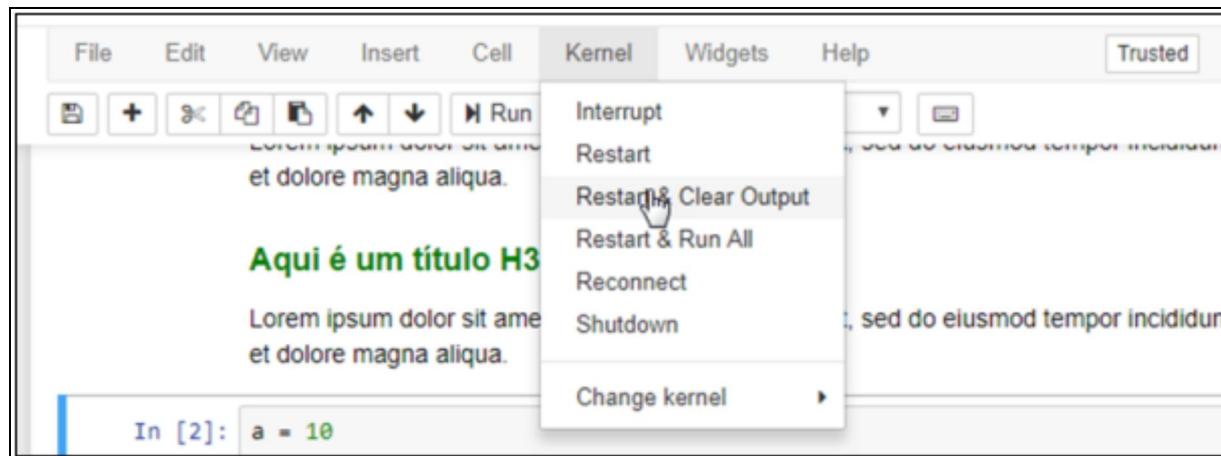
. Below the code cell, the text 'Aqui é um título H1' is displayed in bold black font. At the bottom, there is some placeholder Latin text: 'Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. .'

À medida que vamos desenvolvendo nossos códigos, o Jupyter vai salvando automaticamente as etapas para nós. Isso é muito útil.

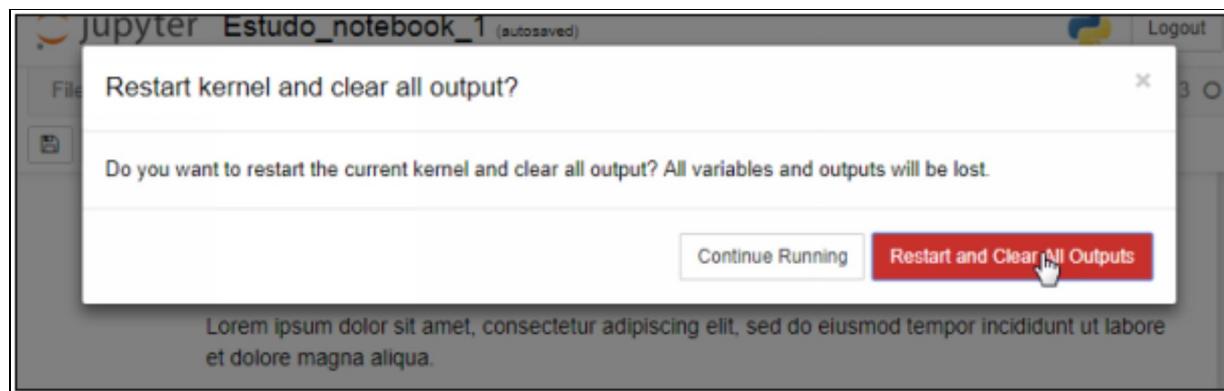
Quando vamos escrevendo nossos códigos as marcações/avisos de execução das células vão aparecendo automaticamente nos colchetes no início das células (**In [x]:**), veja abaixo:



Algo **importante** de saber é que podemos **resetar** (reiniciar) todos esses valores de ordem de execução das células a partir do **Menu -> Kernel -> Restart & Clear Output**:



Após escolher a opção faça a confirmação:

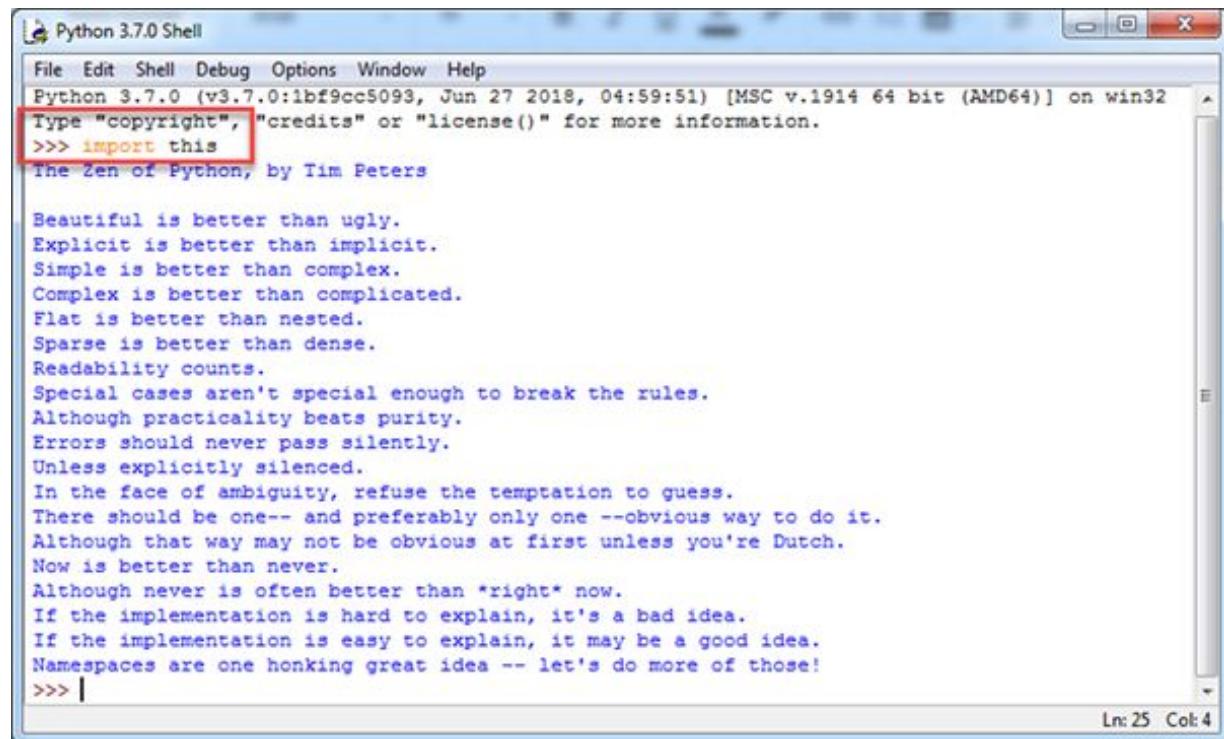


Agora você já conhece os elementos principais para começar a desenvolver e documentar seus códigos. Por isso vamos agora estudar a programação com Python.

## Programação com Python

Neste tópico vamos entender os fundamentos e filosofia por trás da programação com Python. Uma frase que resume bem a mente de um desenvolvedor Python é a seguinte: **simples é melhor que complexo e complexo é melhor que complicado.**

Abaixo segue todos os princípios por trás da filosofia de desenvolvimento com o Python (*The Zen of Python*). O comando está em destaque mostrado na figura abaixo.



The screenshot shows a Windows-style window titled "Python 3.7.0 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main window displays the "Zen of Python" code. A red box highlights the command "import this". The code itself is a series of 19 English statements starting with "Beautiful is better than ugly." and ending with "Namespaces are one honking great idea -- let's do more of those!". The window has scroll bars on the right and bottom, and status text at the bottom right indicating "Ln: 25 Col: 4".

```
File Edit Shell Debug Options Window Help
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:59:51) [MSC v.1914 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
>>> |
Ln: 25 Col: 4
```

Computadores, ao contrário do que muita gente pensa, até o presente momento são máquinas burras. Eles não têm inteligência. Portanto, precisam de um cérebro humano para orientá-lo a executar suas tarefas devidamente. E é para isso que existe o programador de computadores.

No geral, programação é uma atividade que envolve o desenvolvimento de um processo de análise de tarefas para

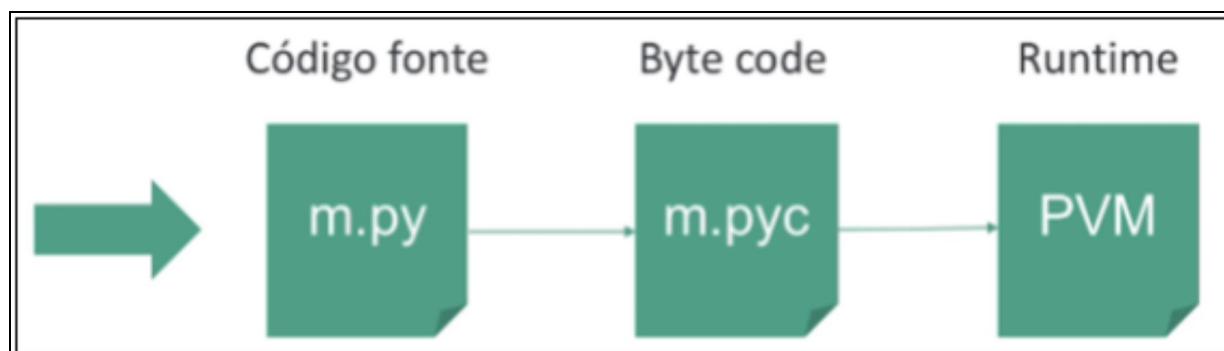
solucionar algum problema. A partir dessa análise se criam modelos divididos em etapas que precisam ser respeitadas em uma sequência de instruções compreendidas pelo computador para que o mesmo possa executá-las.

Iremos, a partir de agora, conhecer a sintaxe de programação da linguagem Python e estudar como a linguagem entende e executa nossas ordens. Para quem nunca programou computadores antes, essa é uma excelente oportunidade de começar com Python.

O mercado de trabalho para programadores demanda Python como uma das principais linguagens de programação. Isso porque Python é extensamente utilizado para trabalhar nas áreas de ciência de dados, análise de dados e inteligência artificial (IA).

Existem linguagens compiladas e interpretadas. Python é uma linguagem interpretada simples, clara e de fácil entendimento e manutenção.

Uma linguagem interpretada é aquela que precisa de um interpretador interno, o qual irá traduzir a linguagem de alto nível escrita por nós para a linguagem de máquina entendida pelo computador. Em outras palavras, na linguagem Python o código fonte é traduzido em *bytecode* que é um formato binário com instruções para o interpretador. Algo importante a respeito do bytecode é que ele é multiplataforma, por isso podemos desenvolver nossos códigos em Python e executá-los em qualquer sistema operacional como: Linux, Mac ou Windows.



Basicamente, podemos executar um programa em Python de três principais formas:

- Utilizando o *shell* (comando executados linha a linha);
- A partir de *scripts* (arquivos de texto com extensão **.py**);
- Modo interativo a partir de um navegador de internet (Jupyter Notebook).

No decorrer deste livro iremos utilizar cada uma dessas três maneiras de executar códigos em Python.

Vamos conhecer a seguir uma característica importante com relação a linguagem Python que é a indentação.

### Indentação

A estrutura de organização dos códigos com Python é feita de uma maneira peculiar a partir da indentação (ou recuo). Isso faz com que a linguagem abra mão da utilização de chaves, colchetes e parênteses desnecessários. Claro que isso nos traz benefícios e problemas.

Os benefícios são que nossos códigos ficam mais claros e simplificados. Por outro lado, devemos ficar bastante atentos com a formatação adequada da indentação. Mas, afinal, o que vem a ser indentação?

A indentação serve para definir ou destacar a estrutura de um algoritmo a partir de recuos. Para nossos códigos com Python utilizamos a tecla **tab** (ou 4 espaços) para fazer a indentação. Portanto, a **indentação faz parte da sintaxe em Python**, assim precisamos respeitá-la.

Veja um exemplo abaixo:

```
1 print('Nível 1')
2
3 if(True):
4     print('Nível 2')| indentação
```

## Comentários

É muito importante ter um código limpo e bem documentado. Para isso, precisamos utilizar caracteres especiais que fazem com que a linguagem Python não execute (interprete) no compilador linhas comentadas.

Existem duas principais maneiras de fazer comentário com a linguagem. Veja abaixo:

Parágrafos entre três aspas simples ("....") ou frases após cerquilha (# - jogo da velha) são tidos como comentários, portanto não interpretados pelo compilador.

```
*ex_fibo.py - C:/Users/Desktop/E-Book Amazon/Álgebra Linear Python/ex_fibo.py (3.7.0)*
File Edit Format Run Options Window Help
import numpy as np
from numpy.linalg import det, inv, eig

# Aqui é um comentário de uma linha

...
            podemos escrever
várias frases e até mesmos vários parágrafos

Aqui digitamos o texto

Aqui também
... 
```

É extremamente indicado que você sempre que possível utilize comentários nos seus códigos. Isso vai facilitar o entendimento para

futuras atualizações e ajudar outros desenvolvedores que, por ventura, venham a utilizar o seu código.

## Dicas de Estudos

Aprender Python é algo bastante intuitivo, mas não é por isso que seja fácil. No decorrer do nosso aprendizado da linguagem, precisamos vencer algumas barreiras. Temos que nos acostumar com a sintaxe de estruturação da linguagem, bem como, conhecer e memorizar alguns comandos básicos. Saber como buscar informações extras e ajuda (Help) também é essencial.

Assim devemos entender primeiro a estrutura sintática de formação da linguagem. Precisamos conhecer como representar variáveis e seus tipos.

Entender o funcionamento da estruturação de operações matemáticas e lógicas é fundamental.

Frequentemente, precisamos fazer rotinas (algoritmos) de repetição e toda linguagem de programação têm suas especificações. Para facilitar nossos estudos, vamos aprender algumas funções rotineiras. Essas funções nos auxiliarão em diversas atividades importantes do decorrer do desenvolvimento dos nossos projetos com Python.

Também, nos nossos estudos é indispensável conhecer a respeito de banco de dados, pois esses nos ajudam com a segurança e manipulação adequada dos nossos dados e informações.

Então fique tranquilo(a) que nos capítulos a seguir iremos passar por cada um desses assuntos citados acima.



## **Capítulo 2**

## **Variáveis, tipos, estruturas e operações**

Neste capítulo iremos aprender a estrutura de funcionamento sintática da linguagem Python. Portanto, é um dos capítulos mais importantes.

Também iremos conhecer como pode ser dividido um algoritmo e seus elementos essenciais de funcionamento operacional do corpo lógico.

Um algoritmo computacional é algo semelhante a uma receita de bolo, na qual temos os ingredientes (variáveis) e as ações (métodos e funções) que devemos realizar com eles.

Tudo isso deve seguir uma estrutura lógica, uma ordem de funcionamento com etapas finitas bem delineadas e claras.

Pois bem, vamos agora nos adentrar na programação de computadores propriamente dita utilizando Python.

## Como estruturar um programa

Podemos entender a programação de computadores como a solução de problemas humanos a partir da utilização do poder computacional das máquinas. Assim, para solucionarmos qualquer que seja o problema, devemos primeiro fazer uma definição estruturada e precisa.

Aqui está um bom plano de ação que podemos adotar na construção de algoritmos:

**1 - Definir o problema** -> **2 - Separar em partes** -> **3 - Ordenar atividades** -> **4 - Traduzir para códigos** -> **5 - Obter solução do problema.**

Todo esse fluxo de trabalho é preciso que o programador saiba estruturar bem. Entretanto, a atividade de traduzir para códigos é onde entra o programador com mais força.

Felizmente, Python nos permite resolver problemas de diversas áreas do conhecimento. Com Python estaremos munidos de uma infinidade de facilidades e **bibliotecas prontas** à nossa disposição (mais de **226 mil** <https://pypi.org/>).

No geral, todo problema precisa ser resolvido a partir de operações lógicas ou matemáticas, uma vez que estamos falando de problemas computacionais.

Assim, vamos agora estudar como podemos efetuar operações matemáticas e lógicas com Python.

## Operações matemáticas e lógicas

Quando estamos trabalhando com operações matemáticas estamos lidando com tipos numéricos. Basicamente, temos os seguintes tipos: **inteiros**, **decimais** e **lógicos**. Com eles, nossas operações podem ser realizadas a partir de valores pertencentes a um ou mais desses tipos.

Antes de começarmos nossos estudos com operações matemáticas e lógicas, vamos conhecer as chamadas funções ***built-in***. Essas funções estão naturalmente presentes no interpretador Python, assim não estão atreladas a bibliotecas ou pacotes externos. Elas são funções nativas da linguagem Python.

Portanto, mais a frente você verá que elas não precisarão ser importadas com a diretiva ***import***.

## Funções *built-in*

As funções apresentadas neste tópico nos ajudarão bastante com os nossos estudos e desenvolvimentos de algoritmos.

Precisamos, constantemente, fazer pequenas buscas, tirar dúvidas de parâmetros de entradas e saídas de funções, saber o tipo de variável que estamos trabalhando, identificar o diretório atual de trabalho, tamanho de vetores e listas de armazenamento, criar/abrir/modificar/excluir arquivos, dentre outras.

As funções *built-in* nos ajudam nessas e em várias outras de nossas necessidades básicas enquanto programadores.

Segue abaixo uma lista com as principais e mais utilizadas funções *built-in*. Até aqui você precisa apenas conhecer que existem tais funções. No decorrer dos nossos estudos iremos fazer uso de várias delas.

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

Conheça abaixo as ações de algumas das funções mais utilizadas:

`dir()` - retorna os métodos do objeto analisado.

**help()** - retorna informações e funcionamento dos objetos de interesse.

**type()** - retorna o tipo do objeto.

**len()** - retorna o tamanho do objeto.

**int()** - transforma objeto em número inteiro.

**float()** - transforma objeto em ponto flutuante (número decimal).

**list()** - transforma o objeto em uma lista.

**max()** - encontra o máximo valor de um conjunto de objetos.

**min()** - encontra o valor mínimo de um conjunto de objetos.

**range()** - cria um intervalo de valores igualmente espaçados por intervalos definidos.

## Operadores matemáticos, relacionais e lógicos

Toda linguagem de programação é antes de tudo uma calculadora sofisticada. Portanto, uma linguagem que se preze deve conter todas as operações básicas da matemática.

Abaixo estão listados os principais operadores matemáticos que são utilizados pela Python:

Operador	Descrição	Exemplo
+	Soma	$2 + 7 = 9$
-	Subtração	$3 - 1 = 2$
*	Multiplicação	$2 * 7 = 14$
/	Divisão	$50 / 5 = 10$
%	Módulo	$5 \% 2 = 1$
**	Potência	$2 ** 4 = 16$
int()	Transforma em inteiro	<code>int(3.14) = 3</code>
float()	Transforma em float	<code>float(5) = 5.0</code>

A seguir temos os operadores relacionais lógicos:

Operador	Descrição
<code>==</code>	Igualdade/equivalecia
<code>!=</code>	Desigualdade/inequivalecia
<code>&gt;</code>	Maior que
<code>&lt;</code>	Menor que
<code>&gt;=</code>	Maior que ou igual a
<code>&lt;=</code>	Menor que ou igual a

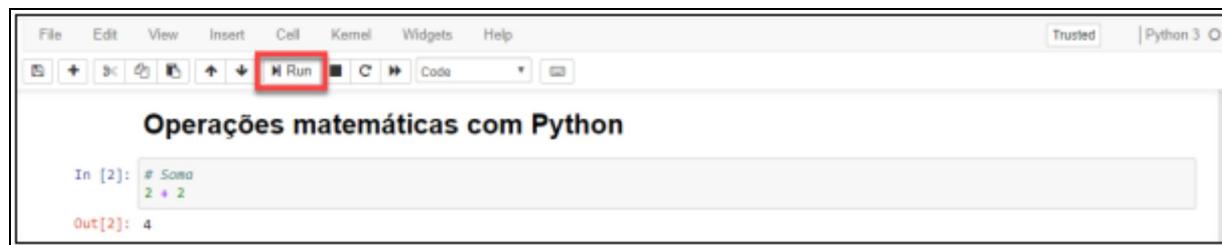
Vejamos agora os operadores de atribuição. Alguns deles são utilizados com bastante frequência como é o caso do `+=` e `-=`.

Operador	Descrição	Exemplo
<code>=</code>	Atribuição	<code>X = 10</code>
<code>+=</code>	Soma	<code>X += 10 (X = X + 10)</code>
<code>-=</code>	Subtração	<code>X -= 10 (X = X - 10)</code>
<code>*=</code>	Multiplicação	<code>X *= 10 (X = X*10)</code>
<code>/=</code>	Divisão	<code>X /= 10 (X = X/10)</code>
<code>%=</code>	Módulo	<code>X% = 10 (X = X%10)</code>
<code>**=</code>	Potência	<code>X** = 10 (X = X**10)</code>
<code>//=</code>	Divisão inteira	<code>X// = 10 (X = X//10)</code>

Por fim, e não menos importante, temos os operadores lógicos:

Operador	Descrição	Exemplo
And	Se ambos operadores forem True, retorna True	$(x \text{ and } y)$ é True
Or	Se um dos operadores for True, retorna True	$(x \text{ or } y)$ é True
Not	Usado para reverter o estado da lógica	Not $(x \text{ and } y)$ é False

Agora vamos conferir exemplos de operações na prática utilizando o Jupyter Notebook:



The screenshot shows a Jupyter Notebook interface. The toolbar at the top includes File, Edit, View, Insert, Cell, Kernel, Widgets, Help, Trusted, and Python 3. A red box highlights the 'Run' button in the toolbar. Below the toolbar, the title 'Operações matemáticas com Python' is displayed. A code cell is shown with the input 'In [2]: # Soma 2 + 2' and the output 'Out[2]: 4'.

## Operações matemáticas com Python

```
In [1]: # Soma  
2 + 2
```

```
Out[1]: 4
```

```
In [2]: # Subtração  
2 - 1
```

```
Out[2]: 1
```

```
In [3]: # Multiplicação  
7 * 7
```

```
Out[3]: 49
```

```
In [4]: # Divisão  
7 / 2
```

```
Out[4]: 3.5
```

```
In [5]: # Módulo ou resto da divisão  
9 % 2
```

```
Out[5]: 1
```

Vamos utilizar a função **type()** para identificar o tipo de algum objeto em Python. A função **type()** deve ser utilizada sempre que tivermos dúvidas a respeito do tipo de alguma variável.

Em tópicos mais a frente iremos perceber que podemos ter diversos tipos de variáveis, e precisaremos conhecer os tipos para a devida manipulação.

### Uso da Função type()

```
In [7]: type(77)
Out[7]: int

In [8]: type(1.618)
Out[8]: float

In [9]: type('Aqui temos um texto')
Out[9]: str
```

## Curiosidades

```
In [16]: # Divisão com uma barra / - retorna um float
6 / 2

Out[16]: 3.0

In [17]: # Divisão com duas barras // - retorna um int
6 // 2

Out[17]: 3
```

Podemos fazer conversão de tipos de números utilizando as funções: **float()** e **int()**:

## Conversão de números

```
In [18]: float(3)
```

```
Out[18]: 3.0
```

```
In [19]: int(1.0)
```

```
Out[19]: 1
```

```
In [20]: int(1.618) # coletando apenas a parte inteira!
```

```
Out[20]: 1
```

Algumas vezes é importante fazermos as transformações de números comuns para hexadecimal e/ou binários. Veja abaixo como fazer isso com Python:

## Transformações Hexadecimal e Binário

```
In [21]: hex(77) # para representar o inteiro 77 em hexadecimal
```

```
Out[21]: '0x4d'
```

```
In [22]: hex(231)
```

```
Out[22]: '0xe7'
```

```
In [23]: bin(77)
```

```
Out[23]: '0b1001101'
```

```
In [24]: bin(231) # para representar o inteiro 231 em binário
```

```
Out[24]: '0b11100111'
```

Iremos conhecer agora algumas funções largamente utilizadas na programação científica. Veja abaixo quais são essas funções:

## Funções para computação científica

```
In [25]: # Retorna o valor absoluto (positivo)  
abs(-8)
```

```
Out[25]: 8
```

```
In [26]: # Retorna o valor absoluto  
abs(8)
```

```
Out[26]: 8
```

```
In [27]: # Retorna a Potência qualquer de um número  
pow(2,8) # potência de 2 ^ 8 (dois elevado a oito)
```

```
Out[27]: 256
```

```
In [28]: # Retornar a potência  
pow(5,100) # potência do 5 ^ 100
```

```
Out[28]: 7888609052210118054117285652827862296732064351090230047702789306640625
```

```
In [29]: # Faz arredondamento de números  
round(3.1415)
```

```
Out[29]: 3
```

```
In [30]: round(1.618)
```

```
Out[30]: 2
```

```
In [31]: # Arredondar com duas casas decimais  
round(1.618,2)
```

```
Out[31]: 1.62
```

As variáveis que definem um modelo, que precisa ser resolvido, podem apresentar os mais variados tipos e formatos. Por isso precisamos aprender com cuidado como representar, declarar e atribuir valores aos diversos tipos de variáveis que armazenam as

estruturas de dados dos nossos problemas. Agora vamos estudar essas representações.

## Variáveis e tipos

As variáveis são espaços em memória utilizados para armazenar determinados valores de um tipo específico. Toda variável deve possuir um nome e um valor com a atribuição de um determinado tipo.

**Muita atenção** neste ponto a seguir. Vale ressaltar algumas **regras de nomeação** de variáveis em Python:

- Não devemos declarar variáveis com nomes que começam com números;
- Não pode haver espaço nos nomes das variáveis. Podemos utilizar ‘\_’ no lugar de espaço;
- Não podemos utilizar qualquer um desses símbolos a seguir: “”, <>, /, |, (), @, #, &, \$, ^, \*, +, -!

Também temos algumas palavras reservadas pelo Python e, por isso, não podemos utilizar essas palavras para declarar variáveis. Segue a lista das variáveis reservadas que você deve evitar utilizar como variáveis. Essas palavras são utilizadas exclusivamente pelo interpretador Python e diz respeito à sintaxe de funcionamento da linguagem.

False	True	as
class	def	elif
finally	from	if
is	nonlocal	Or
return	while	yield
None	and	Assert
continue	del	Else
for	global	import
lambda	not	Pass
try	with	break
except	in	raise

**Atenção:** Não utilizar as palavras deste quadro como nome de variáveis.

Toda variável possui um tipo e um valor atribuído. Antes de prosseguirmos, vejamos o seguinte caso da diferença básica entre valor e variável.

```
In [1]: a = 77 # variável 'a' com valor atribuido 77
In [2]: print(77) # imprime o valor 77
77
In [3]: print(a) # imprime o valor da variável 'a'
77
```

## Variável Inteira

Vamos criar algumas variáveis com o tipo inteiro. O Python possui uma **tipagem dinâmica**, isto é, não precisamos especificar os tipos de variáveis como nas linguagens Java, C++, C#, por exemplo.

Assim, para declararmos variáveis do tipo inteiro, basta fazermos:

```
In [1]: # Atribuindo o valor 7 à variável var_a  
var_a = 7
```

```
In [2]: # Imprimindo o valor da variável  
var_a
```

```
Out[2]: 7
```

```
In [3]: # Imprimindo o valor da variável  
print(var_a)
```

```
7
```

Supondo agora que apenas temos um nome de variável declarada sem atribuição de valor:

```
In [5]: # Não podemos utilizar uma variável sem valor atribuído  
var_b
```

---

```
NameError: name 'var_b' is not defined
```

```
-----  
Traceback (most recent call last)  
<ipython-input-5-4486aaff313f> in <module>()  
      1 # Não podemos utilizar uma variável que não foi definida. Veja a mensagem de erro.  
----> 2 var_b  
  
NameError: name 'var_b' is not defined
```

Teremos uma mensagem de erro como mostra acima, pois não podemos declarar uma variável sem definir o seu valor. Portanto, de algum modo precisamos atribuir valor à variável.

Constantemente, iremos utilizar a função **type()** para checar o tipo das nossas variáveis. Vejamos abaixo um exemplo:

```
In [7]: var_c = 20
In [8]: type(var_c)
Out[8]: int
```

Devemos estar atentos com as letras maiúsculas e minúsculas no Python, pois ele é **case-sensitive**<sup>[9]</sup>. Letras maiúsculas e minúsculas tem diferença no nome da variável.

```
In [4]: var_t = 8 # Python é case-sensitive !
In [5]: Var_t
-----
NameError: name 'Var_t' is not defined
Traceback (most recent call last)
<ipython-input-5-b6a2c2ed3874> in <module>()
----> 1 Var_t
```

## Float e double

Variáveis do tipo **float** e **double** são aquelas que apresentam a parte decimal além da inteira.

A variável do tipo *float* aloca um espaço de **4 bytes** de memória enquanto a do tipo *double* **8 bytes**.

Como a tipagem em Python é dinâmica, automaticamente é identificado que estamos utilizando uma variável **float** ou **double** quando utilizamos o ponto de decimal. Veja exemplo abaixo:

```
In [9]: var_d = 3.14
In [10]: type(var_d)
Out[10]: float

In [14]: var_e = 11.
In [15]: type(var_e)
Out[15]: float
```

## Variável String

Quando estamos precisando armazenar caracteres alfanuméricos (letras), palavras ou frases podemos utilizar a variável do tipo **string**.

Devemos utilizar as aspas simples (apóstrofo) ou duplas. Veja abaixo como devemos proceder:

## Variável do tipo string

```
In [1]: str_var_1 = 'A'
```

```
In [2]: type(str_var_1)
```

```
Out[2]: str
```

```
In [3]: str_var_2 = 'Python'
```

```
In [4]: type(str_var_2)
```

```
Out[4]: str
```

```
In [5]: str_var_3 = "Python uma excelente linguagem!"
```

```
In [6]: type(str_var_3)
```

```
Out[6]: str
```

Podemos concatenar as variáveis do tipo **string** da seguinte maneira:

```
In [12]: nome1 = "Maria"
```

```
In [13]: nome2 = " Renata"
```

```
In [14]: total = nome1 + nome2
```

```
In [15]: total
```

```
Out[15]: 'Maria Renata'
```

## Variável Booleana

Variáveis que podemos admitir os valores verdadeiro (**True**) ou falso (**False**) são chamadas do tipo booleana. Em Python a declaração desse tipo de variável é feita da seguinte maneira:

### Variável Booleana

```
In [7]: a = True # OBS True deve iniciar com letra maiúscula
```

```
In [8]: type(a)
```

```
Out[8]: bool
```

```
In [9]: b = False # OBS False deve iniciar com letra maiúscula
```

```
In [10]: type(b)
```

```
Out[10]: bool
```

## Declarações Múltiplas

Algo muito prático é que no Python podemos declarar diversas variáveis de uma só vez. No entanto, devemos ter bastante cuidado para não confundirmos os valores nas declarações. Veja os exemplos abaixo:

### Declaração Múltipla

```
In [1]: disciplina1, disciplina2, disciplina3 = "Química", "Estatística", "Matemática"  
In [2]: disciplina1  
Out[2]: 'Química'  
  
In [3]: disciplina2  
Out[3]: 'Estatística'  
  
In [4]: disciplina3  
Out[4]: 'Matemática'
```

Podemos, simultaneamente, atribuir o mesmo valor a mais de uma variável diferente, como mostrado abaixo:

```
In [5]: assunto1 = assunto2 = assunto3 = "Teoria da relatividade"  
In [6]: assunto1  
Out[6]: 'Teoria da relatividade'  
  
In [7]: assunto2  
Out[7]: 'Teoria da relatividade'  
  
In [8]: assunto3  
Out[8]: 'Teoria da relatividade'
```

Podemos, também, misturar os tipos de variáveis:

```
In [9]: idade, nome, vivo = 30, "Renata", True
```

```
In [10]: idade
```

```
Out[10]: 30
```

```
In [11]: nome
```

```
Out[11]: 'Renata'
```

```
In [12]: vivo
```

```
Out[12]: True
```

## Principais erros de declaração de variáveis

O **Capítulo 8** foi dedicado inteiro para os estudos dos erros e exceções. Mas vamos aqui antecipar algumas coisas. Veja abaixo os erros mais comuns de quando estamos iniciando com a linguagem.

```
In [1]: # Mensagem de erro, pois o Python não permite nomes de variáveis que iniciem com números
7x = 50

File "<ipython-input-1-0e9ed80ac459>", line 2
7x = 50
^

SyntaxError: invalid syntax


In [2]: class = 3 # class é uma palavra reservada em Python

File "<ipython-input-2-375b33962ec2>", line 1
class = 3 # class é uma palavra reservada em python
^

SyntaxError: invalid syntax
```

## Trabalhando com variáveis

São sempre uma atividade comum para qualquer programa computacional as operações matemáticas com variáveis. Vejamos como executar algumas delas:

### Operações com variáveis

In [1]: var1 = 10

In [2]: var2 = 3

In [3]: var1 + var2

Out[3]: 13

In [4]: var2 - var1

Out[4]: -7

In [5]: var2 \* var1

Out[5]: 30

In [6]: var2 / var1

Out[6]: 0.3

In [7]: var2 % var1 # resto da divisão de var2/var1

Out[7]: 3



## Indexação em Python

Este é um subtópico relevante. Portanto muita atenção aqui.

Conforme discutido, tudo em Python é representado por um objeto, no Capítulo 9 aprenderemos mais a respeito disso (programação orientada a objetos - POO).

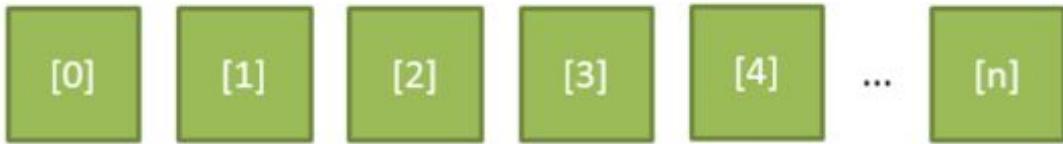
Quando estamos trabalhando com dados estamos sempre nos referindo a algo que está organizado a partir de uma estrutura mínima.

E essa estrutura mínima de armazenamento de dados pode apresentar diversos nomes. Iremos conhecer as principais delas nos próximos capítulos.

Para manipular sequências de disposição de dados em alguns tipos de estrutura em Python precisamos entender como é feita a **contagem de índices de referência**.

Assim como no C++ e Java, Python começa a contar suas estruturas de dados a partir do índice **ZERO**.

A indexação em Python começa por ZERO



No próximo tópico iremos estudar listas e entender como a indexação é primordial. Vamos, por enquanto, utilizar as *strings*, pois já estudamos esse tipo de variável.

Uma *string* é representada por um conjunto de caracteres que apresenta uma determinada ordem de indexação. Assim, podemos acessar elementos desse tipo de variável a partir de chamados

*slices* ou fatias. Em outras palavras, a partir de chamadas dos intervalos de índices dos valores das variáveis podemos fazer aquisições e/ou novas atribuições.

Para Python, por exemplo, a *string* com valor ‘Estudo’ é basicamente uma sequência de letras com uma ordem específica. Isso significa que podemos usar a indexação para obter caracteres ou partes de uma variável desse tipo.

Em Python utilizamos o **colchetes [ ]** para representar o índice de um objeto. Para melhor compreender tudo isso veja os exemplos abaixo:

```
In [1]: varstr = 'Estudo'
```

```
In [2]: varstr[0]
```

```
Out[2]: 'E'
```

```
In [3]: varstr[1]
```

```
Out[3]: 's'
```

```
In [4]: varstr[2]
```

```
Out[4]: 't'
```

```
In [5]: varstr[3]
```

```
Out[5]: 'u'
```

```
In [6]: varstr[4]
```

```
Out[6]: 'd'
```

```
In [7]: varstr[5]
```

```
Out[7]: 'o'
```

Podemos utilizar o método **find()** dos objetos *strings* para encontrar o índice da posição em que se encontra um determinado caractere ou conjunto de caracteres.

```
In [8]: # podemos buscar o índice que contém determinado caractere(es)
varstr.find('t')

Out[8]: 2

In [9]: varstr.find('ud')

Out[9]: 3
```

**OBS:** devemos ter cuidado com a indexação, pois em outras linguagens, como **Matlab** e **R**, seus índices de referência começam com '1'. Esse é um motivo de muitos erros de programação com a linguagem, pois rotineiramente nos esquecemos de que a indexação em Python **começa por ZERO**.

Muitas vezes precisamos pegar conjuntos de caracteres a partir de uma determinada posição (indexação). Para isso podemos utilizar os dois pontos ':' para fazer *slices* (fatiamentos).

Vejamos o exemplo abaixo onde queremos retornar todos os elementos de uma *string* começando a partir de uma posição especificada.

```
In [1]: frase = 'Python é uma linguagem muito poderosa!'

In [2]: frase[2:]

Out[2]: 'thon é uma linguagem muito poderosa!'
```

Podemos retornar tudo até chegar à posição 4, por exemplo:

```
In [3]: frase[:12]  
Out[3]: 'Python é uma'
```

Um recurso muito interessante e, por vezes, útil é a utilização da indexação negativa. Nesse caso estamos rastreando os caracteres de trás para frente. Vejamos abaixo como funciona:

```
In [4]: frase[-1]  
Out[4]: '!'  
  
In [5]: frase[-9:-1]  
Out[5]: 'poderosa'
```

Outro recurso interessante, porém, pouco intuitivo é a utilização dos dois pontos duas vezes (::). Com o símbolo de :: podemos fazer fatiamentos mais específicos saltando o número de posições especificadas logo após os ::. Vejamos abaixo um exemplo que esclarece melhor o entendimento:

```
In [6]: frase[::-1] # retorna a string pulando 1 caracter  
Out[6]: 'Python é uma linguagem muito poderosa!'  
  
In [7]: frase[::-2] # retorna a string pulando 2 caracteres  
Out[7]: 'Pto m igae ut oeoa'  
  
In [8]: frase[::-3] # retorna a string pulando 3 caracteres  
Out[8]: 'Ph u namuoora'
```

Agora vamos a uma maneira interessante e rápida de inverter os caracteres de uma frase:

```
In [9]: frase[::-1] # retorna a string pulando 1 caractere
Out[9]: '!asoredop otium megaugnil amu é nohtyP'
```

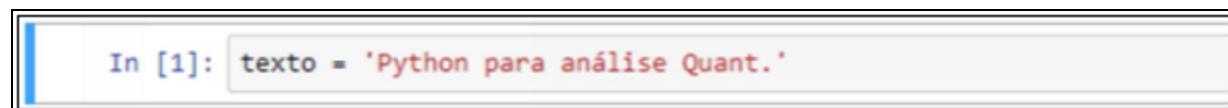
Mais à frente iremos utilizar essas mesmas funcionalidades para obter valores de listas, vetores, matrizes, tuplas e outras. Todas essas são estruturas importantes de armazenamento de dados.

Contudo, já que para entender como funciona a indexação em Python utilizamos as *strings* como exemplo vamos aproveitar para conhecer alguns métodos ou funções ***built-in*** para as *strings*.

## Funções *Built-in* das Strings

Atualmente, os estudos voltados para as variáveis do tipo *strings* estão em alta. Com o avanço da inteligência artificial a compreensão de linguagem natural (NLP - *Natural Language Processing*) vem se tornando etapa cada vez mais importante para os avanços nesse campo.

Pensando nisso, o Python já apresenta um conjunto bastante expressivo de funções e bibliotecas para nos ajudar a manipular *strings*.

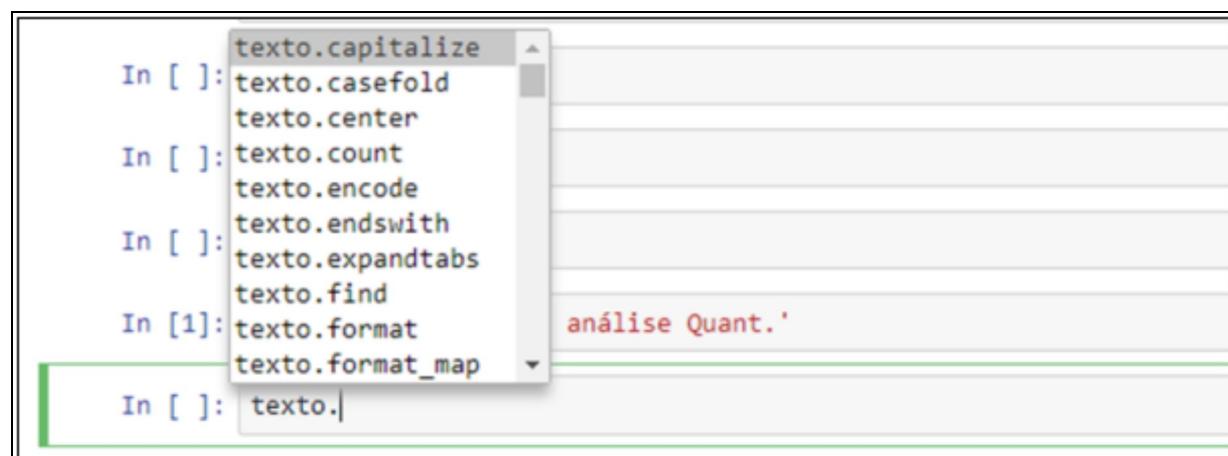


```
In [1]: texto = 'Python para análise Quant.'
```

Para que possamos visualizar essas funções (ou métodos, pois, como já dito antes, tudo em Python é um objeto) basta digitar um ponto.

Portanto, cada tipo de objeto em Python possui um conjunto de atributos e métodos (funções) para sua manipulação. Dessa forma, para visualizar as funções existentes basta colocar um **ponto** após o objeto **string** e apertar **Tab**.

Veja exemplo abaixo:



```
In [ ]: texto.capitaliz
In [ ]: texto.casefold
In [ ]: texto.center
In [ ]: texto.count
In [ ]: texto.ecode
In [ ]: texto.endswith
In [ ]: texto.expandtabs
In [ ]: texto.found
In [1]: texto.format
In [1]: texto.format_map
In [ ]: texto.
```

The screenshot shows a Jupyter Notebook interface. In the code editor, there is a list of string methods appearing as suggestions after the dot operator. The word 'capitaliz' is partially typed, and the suggestion dropdown lists 'capitaliz', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format', and 'format\_map'. To the right of the dropdown, the string 'análise Quant.' is visible. The cell number 'In [1]' is present at the top left of the code area.

O recurso de utilizar `.` (ponto) depois a tecla **tab** para visualizar as funções é chamado de **intellisense**.

Note que já fizemos o uso da função `.find()`. Vamos conhecer mais algumas:

```
In [2]: # Transforma texto para caracteres maiúsculos  
        texto.upper()  
  
Out[2]: 'PYTHON PARA ANÁLISE QUANT.'  
  
In [3]: # Transforma texto para caracteres minúsculos  
        texto.lower()  
  
Out[3]: 'python para análise quant.'
```

Uma função muito utilizada é a `.split()`. Com ela podemos separar conjuntos de *strings* a partir de caracteres especificados na função. O padrão da função `split()` é dividir a *string* nos espaços, lembrando que os espaços também são caracteres. Veja um exemplo abaixo:

```
In [4]: # Divide string por espaços em branco (padrão da função)  
        texto.split()  
  
Out[4]: ['Python', 'para', 'análise', 'Quant.']}
```

```
In [5]: # Divide string no elemento especificado 't'  
        texto.split('t')  
  
Out[5]: ['Py', 'hon para análise Quan', '.']
```

Agora que você já conhece algumas das principais funções, pode continuar explorando o restante delas.

Lembrando que trabalhar com *string* é algo que está em ascensão, pois dentro do campo de inteligência artificial são, frequentemente, utilizadas para análises de sentimento em redes sociais, rastreios de informações na web, traduções automáticas, dentre outras.

## Variáveis para armazenamento de dados

Neste tópico iremos estudar variáveis úteis para trabalhar com armazenamento de **dados**. Sabemos que o nosso elemento de estudo são os dados que de alguma maneira foram coletados e precisam de um devido tipo estrutural e armazenamento.

Como programadores, cientistas e analistas precisaremos em etapas subsequentes transformar nossos dados em informações. Por isso a importância de melhor estrutura-los dentro da forma mais adequada de armazenamento.

Então para cada tipo de variável, visando o devido armazenamento, precisamos aprender:

- O que são as variáveis;
- Como funciona a indexação e fatiamentos (*slices*);
- Os métodos básicos presentes nas variáveis;
- Como fazer aninhamentos.

Vamos assim conhecer as principais e mais importantes formas de armazenar dados com a linguagem Python.

## Listas

As listas são uma das mais importantes e utilizadas variáveis em Python. As variáveis do tipo listas são tidas como uma sequência generalizada, por isso os elementos dentro de uma lista podem ser facilmente mutáveis. Em outras palavras, podemos modificar facilmente, de diversos modos, os elementos dentro de uma lista.

Cada tipo de variável de armazenamento de dados na Python é representado de maneira específica, por exemplo, para representarmos **listas** utilizamos os **colchetes []**.

Logo, todas as vezes que declaramos uma variável cujos elementos estão dentro de colchetes estamos criando uma lista em Python.

Os elementos das listas devem estar separados por vírgula, como mostra o caso geral abaixo:

**Lista = [ ele<sub>1</sub>, ele<sub>2</sub>, ..., ele<sub>N</sub> ]**

Existem alguns tipos de variáveis que precisamos previamente especificar o seu tamanho, no entanto, isso não acontece com as listas.

As listas são um tipo de variável bastante flexível. Outra característica importante é que não precisamos restringir os tipos dos elementos de uma lista. Assim podemos ter elementos de **vários tipos** dentro de uma mesma lista.

Portanto, antes de começar a ver os exemplos práticos vamos às principais características das variáveis do tipo **lista**:

- Listas são ordenadas;
- Listas podem ser **iteradas**<sup>[10]</sup> (podemos percorrer de modo ordenado seus elementos);

- Listas são mutáveis (podemos adicionar, mover, mudar ou remover os elementos);
- Listas aceitam múltiplos tipos de elementos.

Vamos aos exemplos:

```
In [1]: # Cria uma lista vazia
lista_vazia = []

In [2]: # Também cria uma lista vazia
lista_vazia2 = list()

In [3]: type(lista_vazia)

Out[3]: list

In [4]: type(lista_vazia2)

Out[4]: list
```

```
In [1]: # para criar listas utilizamos []
lista_nomes1 = ['Ramon', 'Ricardo', 'Rafael', 'Renata']

In [2]: lista_nomes2 = ['Ramon', Ricardo, Rafael, Renata']
```

```
In [11]: type(lista_nomes1)

Out[11]: list

In [12]: type(lista_nomes2)

Out[12]: list
```

Preste **atenção** na diferença das duas listas acima (lista\_nomes1 e lista\_nomes2). Na imagem seguinte você perceberá essa diferença.

```
In [5]: # função Len() diz o tamanho da Lista  
len(lista_nomes1)
```

```
Out[5]: 4
```

```
In [6]: len(lista_nomes2)
```

```
Out[6]: 1
```

Podemos notar que a 'lista\_nomes1' possui 4 elementos, enquanto que a 'lista\_nomes2' apenas 1 elemento. Isso porque os elementos das listas são *strings* e na 'lista\_nomes2' temos apenas um elemento, uma *string* ('Ramon , Ricardo , Rafael , Renata').

Preste atenção na posição das aspas simples (apóstrofo) (vale lembrar que também podemos colocar aspas duplas). Veja abaixo:

```
In [7]: # Retorna o elemento index 0  
lista_nomes1[0]
```

```
Out[7]: 'Ramon'
```

```
In [8]: # Retorna o elemento index 0  
lista_nomes2[0]
```

```
Out[8]: 'Ramon , Ricardo , Rafael , Renata'
```

Podemos misturar os tipos de elementos presentes nas listas veja um exemplo a seguir:

```
In [9]: lista_mista = ['elevador', 77, 3.14]
In [10]: print(lista_mista)
['elevador', 77, 3.14]
```

Vejamos agora como fica o acesso a cada um dos elementos da 'lista\_mista'.

```
In [3]: print('Tamanho da lista = ', len(lista_mista))
Tamanho da lista = 3
In [4]: lista_mista[0]
Out[4]: 'elevador'
In [5]: lista_mista[1]
Out[5]: 77
In [6]: lista_mista[2]
Out[6]: 3.14
```

Sempre é importante lembrar que a **indexação** em Python começa pelo **ZERO**. Portanto, como temos três elementos na 'lista\_mista', o terceiro elemento possui índice igual a 2.

Veja abaixo o que acontece se tentarmos acessar o elemento de **índice** igual a 3:

```
In [7]: lista_mista[3]

-----
IndexError                                                 Traceback (most recent call last)
<ipython-input-7-95741ac6169b> in <module>()
----> 1 lista_mista[3]

IndexError: list index out of range
```

Como esperado, temos uma tentativa de acesso de elemento fora do *range* (tamanho) da lista. Isso só poderia nos retornar um erro (*index out of range*) de indexação.

### Deletando elemento de uma lista

Podemos **deletar** um **elemento ou conjunto** de elementos de uma lista a partir da diretiva '**del**'. Veja abaixo como podemos fazer isso:

```
In [8]: del lista_mista[1]

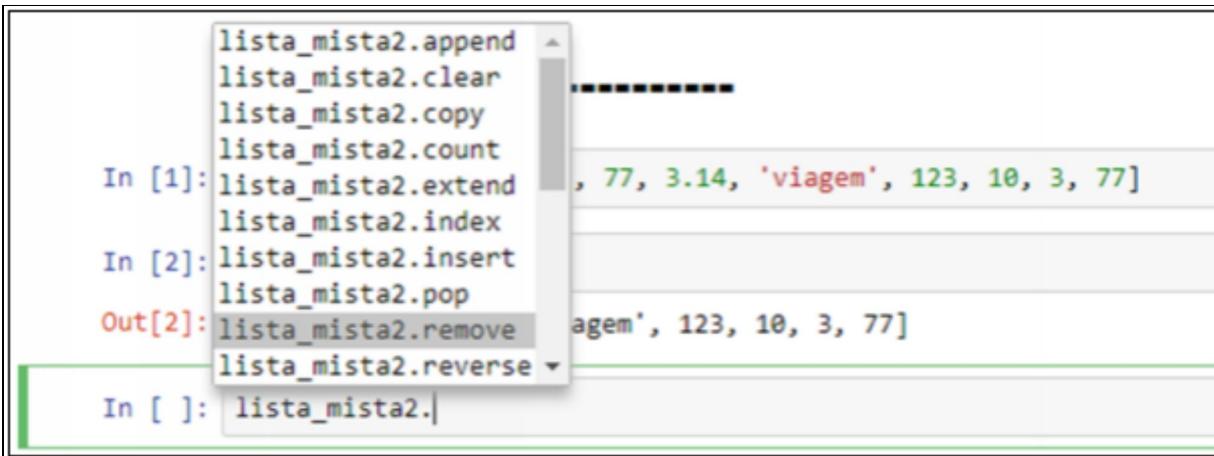
In [9]: print(lista_mista)
['elevador', 3.14]
```

Uma maneira prática de **deletar elementos** específicos de uma lista é a partir do método **remove()**.

```
In [1]: lista_mista2 = ['casa', 77, 3.14, 'viagem', 123, 10, 3, 77]

In [2]: lista_mista2
Out[2]: ['casa', 77, 3.14, 'viagem', 123, 10, 3, 77]
```

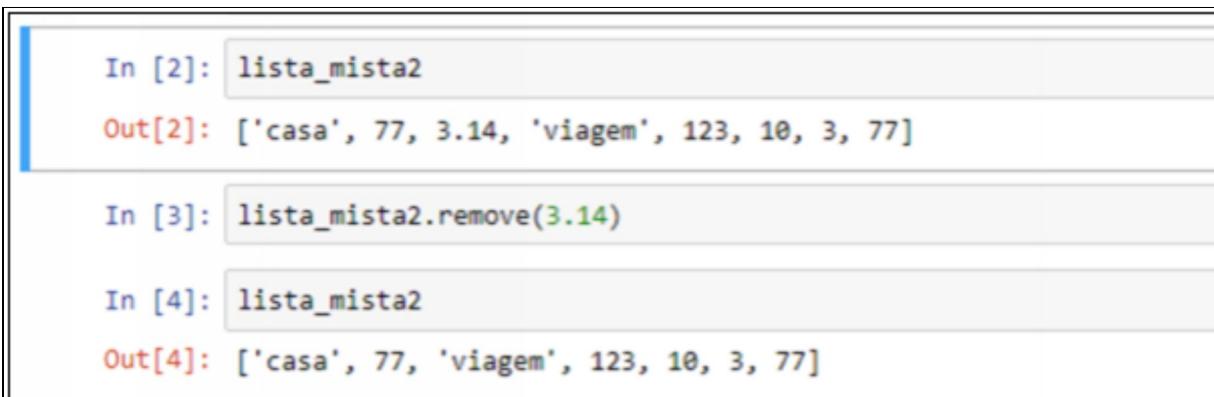
Para isso, basta digitar o nome da lista, adicionar um ponto e a tecla **tab** e em seguida escolher a função **remove()**:



In [1]: lista\_mista2.append  
lista\_mista2.clear  
lista\_mista2.copy  
lista\_mista2.count  
In [2]: lista\_mista2.extend  
lista\_mista2.index  
In [2]: lista\_mista2.insert  
lista\_mista2.pop  
Out[2]: lista\_mista2.remove  
lista\_mista2.reverse  
In [ ]: lista\_mista2.

The screenshot shows a Jupyter Notebook interface. In the code editor, the user has typed 'lista\_mista2.' followed by a tab key. A dropdown menu appears, listing several methods available for the 'lista\_mista2' list object, including 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', and 'reverse'. The 'remove' method is highlighted with a gray background, indicating it is the selected option for completion.

A seguir vamos remover o elemento **float** 3.14:



In [2]: lista\_mista2  
Out[2]: ['casa', 77, 3.14, 'viagem', 123, 10, 3, 77]  
In [3]: lista\_mista2.remove(3.14)  
In [4]: lista\_mista2  
Out[4]: ['casa', 77, 'viagem', 123, 10, 3, 77]

This screenshot shows a Jupyter Notebook session. The user has defined a list named 'lista\_mista2' containing various elements. In step 2, they print the list to see its current state. In step 3, they use the 'remove' method to remove the element '3.14'. Finally, in step 4, they print the list again to verify that '3.14' has been removed, and it is indeed gone, replaced by the next element in the sequence.

Preste bastante **atenção** neste próximo exemplo. Note que temos dois elementos inteiros com valor igual a 77. Vamos pedir para remover o elemento 77 com o método **.remove()**. Veja o que acontece:

```
In [5]: lista_mista2.remove(77)

In [6]: lista_mista2
Out[6]: ['casa', 'viagem', 123, 10, 3, 77]
```

Como podemos observar apenas o primeiro elemento da lista com valor igual a 77 foi removido. Pedindo para remover o elemento 77 pela segunda vez, teremos:

```
In [7]: lista_mista2.remove(77)

In [8]: lista_mista2
Out[8]: ['casa', 'viagem', 123, 10, 3]
```

O segundo elemento foi removido. E agora se pedirmos para remover o elemento 77 novamente, veja o que acontece:

```
In [9]: lista_mista2.remove(77)
-----
ValueError                                                 Traceback (most recent call last)
<ipython-input-9-be71bdec19c9> in <module>()
----> 1 lista_mista2.remove(77)

ValueError: list.remove(x): x not in list
```

Teremos uma mensagem de erro, pois não temos mais o elemento 77 em nossa 'lista\_mista2'.

Para não termos esse tipo de erro podemos verificar, antecipadamente, se existe o valor daquele elemento antes de tentar removê-lo. Mais à frente iremos estudar uma maneira eficiente de fazer essa verificação a partir da diretiva 'in'.

## Aninhamento de Listas

Podemos fazer **aninhamentos** com as listas, ou seja, dentro de estruturas de listas podemos criar outras listas. Vamos ao exemplo abaixo:

```
In [1]: # Vamos criar uma Lista de Listas
lista = [[11,22,33], [1,2,3,4], [100,200,300,400,500]]

In [2]: lista
Out[2]: [[11, 22, 33], [1, 2, 3, 4], [100, 200, 300, 400, 500]]

In [3]: type(lista) # tipo do objeto lista
Out[3]: list

In [4]: len(lista) # tamanho da lista
Out[4]: 3
```

Note que cada elemento da lista também é outra lista:

```
In [5]: lista[0]
Out[5]: [11, 22, 33]

In [6]: type(lista[0])
Out[6]: list
```

Para acessarmos elementos de listas devemos fazer o **slice (fatiamento)** da seguinte maneira:

```
In [7]: lista
Out[7]: [[11, 22, 33], [1, 2, 3, 4], [100, 200, 300, 400, 500]]

In [8]: lista[2][2] # da Lista 3 pegue o elemento 3
Out[8]: 300

In [9]: lista[2][4] # da Lista 3 pegue o elemento 5
Out[9]: 500
```

## Junção de Listas e avaliações de elementos

Devemos aprender o máximo possível a respeito da manipulação de listas, pois elas são largamente utilizadas nas atividades de um cientista ou analista de dados, por exemplo. Lembrando que essas áreas de trabalho estão bastante aquecidas e com uma escassez muito grande de profissionais.

Mais cedo ou mais tarde precisaremos conectar listas nos nossos processos de análises. Veja como fazer isso com o Python:

```
In [1]: lista1 = [1,2,3,4,5]
In [2]: lista2 = [22,33,44,55,66,77]
In [3]: sum_lista = lista1 + lista2
In [4]: sum_lista
Out[4]: [1, 2, 3, 4, 5, 22, 33, 44, 55, 66, 77]
```

Os elementos de uma lista são iteráveis e podem ser facilmente manipulados a partir de fatiamentos, veja alguns exemplos abaixo:

```
In [1]: minha_lista = ['tio', 'mãe', 'pai', 'irmão', 'sobrinha']

In [2]: # Fatiando os 4 primeiros elementos da lista
        minha_lista[0:4]

Out[2]: ['tio', 'mãe', 'pai', 'irmão']

In [3]: # Fatiando os elementos de 2 a 7 da lista
        minha_lista[1:3]

Out[3]: ['mãe', 'pai']

In [4]: # Último elemento da lista
        minha_lista[-1]

Out[4]: 'sobrinha'

In [5]: # Elementos de trás pra frente da Lista
        minha_lista[::-1]

Out[5]: ['sobrinha', 'irmão', 'pai', 'mãe', 'tio']
```

Vamos estudar um operador muito importante utilizado para avaliar se um determinado elemento existe dentro de uma lista.

Lembre-se que surgirá uma mensagem de erro no caso de tentarmos remover de uma lista um elemento que não existe. Assim, nada melhor do que verificar se esse elemento está presente na lista antes da remoção com o operador ‘in’:

```
In [4]: sum_lista
Out[4]: [1, 2, 3, 4, 5, 22, 33, 44, 55, 66, 77]

In [5]: 20 in sum_lista # avalia se 20 estar na sum_lista
Out[5]: False

In [6]: 77 in sum_lista # avalia se 77 estar na sum_lista
Out[6]: True
```

Após digitarmos o nome da lista em seguida um ponto e mais um **tab**, assim como para qualquer outro objeto, teremos acesso a vários métodos para manipular as listas.

Lembre-se que já estudamos o método **.remove()**. Agora vamos estudar outros amplamente utilizados:

```
Out[4]: sum_lista.append
         sum_lista.clear
         sum_lista.copy
         sum_lista.count
         sum_lista.extend
         sum_lista.index
         sum_lista.insert
         sum_lista.pop
         sum_lista.remove
         sum_lista.reverse
In [5]: sum_lista.
In [6]: sum_lista.
```

Se quisermos adicionar novos elementos a uma lista, uma maneira prática de fazermos isso é a partir do método **append()**.

```
In [7]: sum_lista.append(121) # adiciona o elemento 121 no fim da lista  
In [8]: sum_lista  
Out[8]: [1, 2, 3, 4, 5, 22, 33, 44, 55, 66, 77, 121]
```

Geralmente é preciso sabermos quantas vezes determinado elemento aparece em uma lista, para isso podemos utilizar o método **.count()**:

```
In [9]: sum_lista.count(33) # contar quantas vezes o elemento 33 aparece  
Out[9]: 1
```

Às vezes precisamos saber qual é o índice de algum elemento de uma lista, para isso podemos utilizar o **.index()**. Confira o exemplo abaixo:

```
In [10]: sum_lista.index(121) # saber o índice do elemento 121  
Out[10]: 11
```

Podemos inserir um elemento em determinada posição previamente especificada com o método **.insert()**. Vejamos abaixo como isso funciona:

```
In [11]: sum_lista.insert(6,101) # adiciona o elemento 101 na posição 7  
In [12]: sum_lista  
Out[12]: [1, 2, 3, 4, 5, 22, 101, 33, 44, 55, 66, 77, 121]
```

Com frequência precisamos ordenar os elementos de uma lista e para isso utilizamos o método **sort()**.

Também podemos inverter a ordem da lista a partir do **.reverse()**. Analisemos os exemplos abaixo:

```
In [12]: sum_lista  
Out[12]: [1, 2, 3, 4, 5, 22, 101, 33, 44, 55, 66, 77, 121]  
  
In [13]: sum_lista.sort() # ordena os elementos da lista  
In [14]: sum_lista  
Out[14]: [1, 2, 3, 4, 5, 22, 33, 44, 55, 66, 77, 101, 121]  
  
In [15]: sum_lista.reverse() # inverte a ordem dos elementos  
In [16]: sum_lista  
Out[16]: [121, 101, 77, 66, 55, 44, 33, 22, 5, 4, 3, 2, 1]
```

## Cópia e criação de Listas

A sintaxe do Python apresenta uma estrutura muito inteligente e peculiar. Vamos estudar nesta parte um exemplo que mostra uma peculiaridade interessante e muito importante de ser conhecida.

Algumas vezes precisamos fazer cópias de variáveis do tipo lista. Para isso precisamos ter cuidado, pois ao **invés de uma cópia**

poderemos estar **fazendo apenas uma referênciação**, um apontamento, mas não uma cópia.

Veja com **cuidado** o exemplo abaixo:

```
In [1]: # Lista de números
numeros = [11, 22, 33, 44, 55]

In [2]: # Faz referênciação de numeros para novos
novos = numeros

In [3]: # Adicionar valores a lista 'novos'
novos.append(1)
novos.append(77)

In [4]: # Veja o que acontece com numeros:
numeros

Out[4]: [11, 22, 33, 44, 55, 1, 77]
```

Podemos notar, no exemplo acima, que **não criamos** uma nova variável (**novos**) com os elementos de (**numeros**). Para fazermos isso, vejamos abaixo como prosseguir:

```
In [5]: # Cria nova variável com elementos de 'numeros'
numeros2 = numeros[:]

In [6]: numeros2.append(57)

In [7]: numeros

Out[7]: [11, 22, 33, 44, 55, 1, 77]

In [8]: numeros2

Out[8]: [11, 22, 33, 44, 55, 1, 77, 57]
```

Veja duas outras maneiras de fazer cópias de listas:

```
In [9]: # Faz cópia do elementos de 'numeros' para 'numeros3'  
numeros3 = numeros.copy()  
  
In [10]: # Faz cópia do elementos de 'numeros' para 'numeros3'  
numeros4 = list(numeros)
```

Para terminar esse importante tópico sobre listas, vamos criar a lista mais simples e básica de todas que é a lista vazia. Para isso, basta abrirmos e fecharmos colchetes da seguinte maneira:

```
In [1]: lis = []    # cria uma lista vazia  
  
In [2]: type(lis)  
Out[2]: list  
  
In [3]: len(lis)  
Out[3]: 0
```

Como já estudado, a partir de uma lista podemos adicionar elementos com o método `.append()`. Esse tipo de ação é muito frequente nos estudos de um cientista de dados.

## List Comprehension

As **List comprehension** são amplamente utilizadas em *scripts* para agilizar a execução de vários tipos de operações.

Atividades de busca e varredura entre elementos de listas são muito frequentes. A linguagem Python nos disponibiliza uma maneira

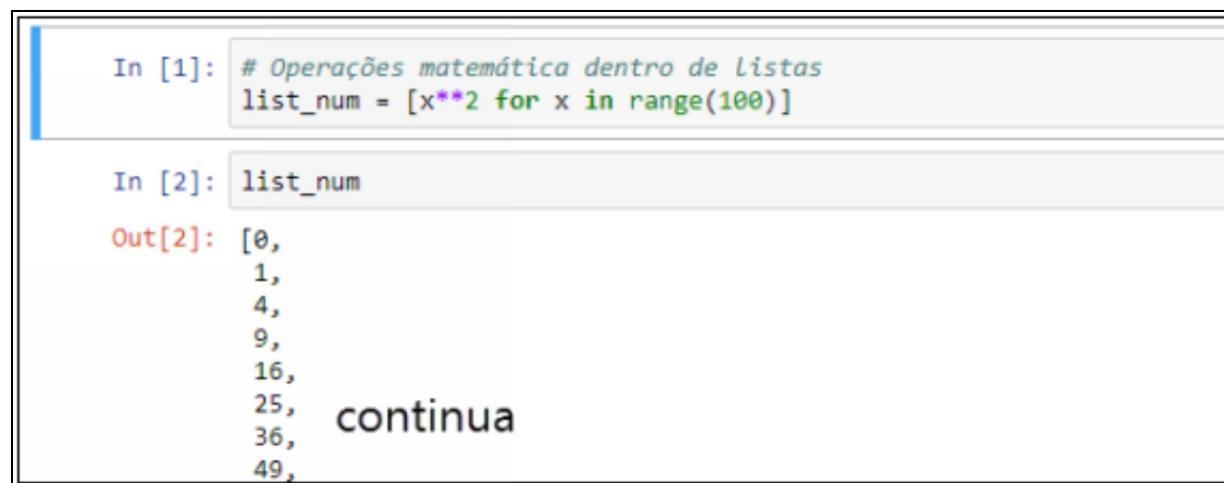
condensada e inteligente de fazer isso.

De modo simplificado, com list comprehension podemos desenvolver **listas** usando uma notação condensada que dispensa a utilização de **loops for** externos. A utilização desses tipos de listas nos traz vantagens com melhores desempenhos de tempo computacional.

A maneira de fazer declarações com a **list comprehension** lembra muito como fazermos descrições de conjuntos numéricos em notação matemática. Veja um exemplo geral:

**Lc = [x for x in ‘sequência’]**

O exemplo abaixo mostra como podemos pegar uma lista com 100 valores de 0 a 99 e elevar cada elemento ao quadrado. Veja como fazer isso com apenas uma linha de código:



```
In [1]: # Operações matemática dentro de Listas
list_num = [x**2 for x in range(100)]  
  
In [2]: list_num  
Out[2]: [0,  
        1,  
        4,  
        9,  
        16,  
        25,  
        36,  
        49,  
        continua
```

Vejamos como obter uma lista com apenas os elementos pares da ‘list\_num’:

```
In [3]: # Obter lista somente com elementos de valores par da 'list_num'  
list_par = [x for x in list_num if x % 2 == 0]  
  
In [4]: list_par  
  
Out[4]: [0,  
        4,  
       16,  
      36,  
      64,  
     100,  
    144,  
   196,  
  256]
```

Podemos utilizar a list comprehension em *strings*. Veja abaixo:

```
In [5]: # De uma sequência de caracteres retornar cada caracter  
letras = [letra for letra in 'Python']  
  
In [6]: letras  
  
Out[6]: ['P', 'y', 't', 'h', 'o', 'n']
```

Algo que deve ser ressaltado é que o interpretador Python sempre entenderá que estamos criando uma lista quando estamos abrindo e fechando colchetes. Isso deve ficar bem claro para todos nós.

Portanto, toda vez que atribuímos valores dentro de **colchetes** para uma determinada variável, imediatamente o Python interpreta essa variável como sendo uma lista.

A seguir, iremos conhecer como o Python interpreta outras estruturas de dados com o uso das **chaves {}** e **parênteses ()**.

## Dicionários

Uma maneira muito prática de organizar os dados e informações no Python é a partir de dicionários.

Podemos entender dicionário como uma tabela de dados do tipo chave e valor.

Como vimos anteriormente, as listas são definidas a partir de colchetes `[]`. Já os elementos dos dicionários são representados dentro de **chaves {}**.

Veja abaixo como é a estrutura do dicionário em Python:

```
dict = { chave1:valor1, chave2:valor2, ... , chaveN:valorN }
```

Os dicionários são ótimos para fazer armazenamento de dados, quando o mapeamento de dados for um requisito técnico importante de ser considerado.

Trabalhar com listas, como já foi estudado, não nos permite fazer associação de pares de valores do tipo chave e valor como é o caso dos dicionários.

Portanto, com os dicionários, temos um mapeamento que relaciona uma coleção de objetos que são armazenados por uma chave, ao invés de uma sequência de objetos armazenados por sua posição relativa como é o caso das listas.

Quando estamos trabalhando com **Big Data** utilizando algum banco de dados não relacional (Mongodb, por exemplo) a estrutura de um dicionário no mapeamento de chave e valor se faz presente e é bastante útil.

Portanto, os dicionários são estruturas de chave e valor onde seus elementos são:

- Não ordenados;
- Iteráveis;

- Mutáveis.

Entenda bem a estrutura de um dicionário porque com certeza você irá precisar utilizá-lo com frequência em seus projetos.

Veja abaixo como fica a diferença entre uma lista e um dicionário:

```
In [1]: # Exemplo de Lista
salario_list = ['Pedro', 1000, 'Carlos', 750, 'Ricardo', 2000, 'Letícia', 2500]

In [2]: salario_list
Out[2]: ['Pedro', 1000, 'Carlos', 750, 'Ricardo', 2000, 'Letícia', 2500]

In [3]: type(salario_list)
Out[3]: list

In [4]: # Exemplo de dicionário
salario_dict = {'Pedro':1000, 'Carlos':750, 'Ricardo':2000, 'Letícia':2500}

In [5]: salario_dict
Out[5]: {'Carlos': 750, 'Letícia': 2500, 'Pedro': 1000, 'Ricardo': 2000}

In [6]: type(salario_dict)
Out[6]: dict
```

Podemos notar que a diferença visual é pequena. No entanto, estruturalmente temos diferenças significativas.

Nos dicionários podemos utilizar as chaves como índices para encontrarmos os valores correspondentes. Veja como podemos fazer isso:

```
In [7]: # Qual é o salário de Pedro?  
salario_dict['Pedro']  
  
Out[7]: 1000  
  
In [8]: # Qual é o salário de Letícia?  
salario_dict['Letícia']  
  
Out[8]: 2500
```

Podemos adicionar valores aos dicionários da seguinte maneira:

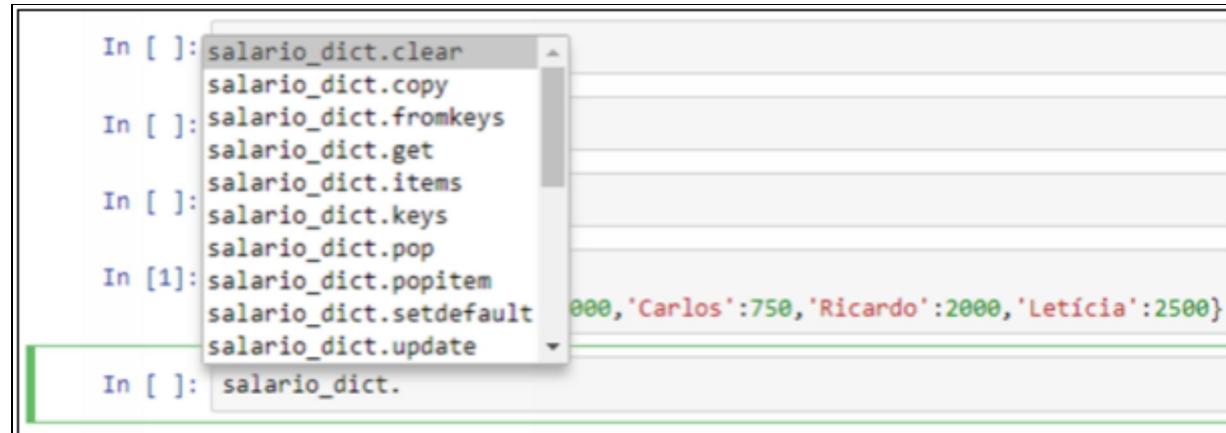
```
In [9]: len(salario_dict)  
Out[9]: 4  
  
In [10]: # Adiciona o salário de Rafael  
salario_dict['Rafael'] = 1200  
  
In [11]: salario_dict  
Out[11]: {'Carlos': 750,  
          'Letícia': 2500,  
          'Pedro': 1000,  
          'Rafael': 1200,  
          'Ricardo': 2000}  
  
In [12]: len(salario_dict)  
Out[12]: 5
```

Vamos agora conhecer os processamentos que podemos fazer com os dados de um dicionário.

## Principais métodos dos dicionários

Assim como os objetos, listas possuem um conjunto de métodos específicos para manipulação de elementos. Os dicionários também possuem os seus próprios métodos.

Como de costume, podemos visualizar todos os métodos a partir do **ponto** mais a tecla **tab**.



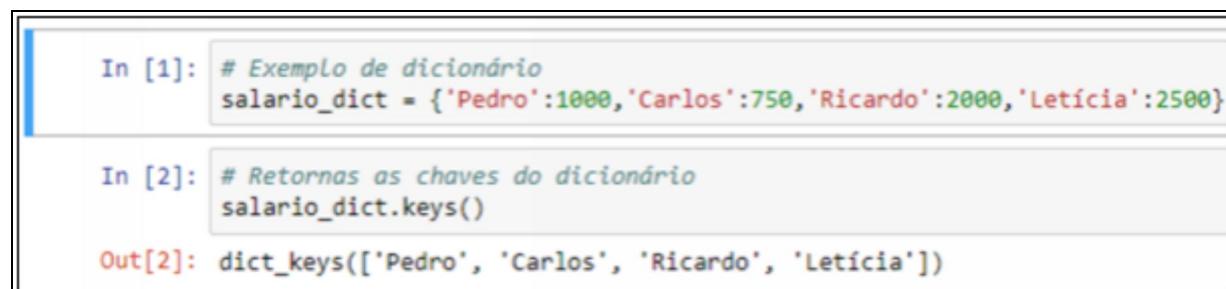
A screenshot of a Jupyter Notebook cell. The code input is:

```
In [ ]: salario_dict.
```

The completion menu is open, listing methods for the `salario_dict` object. The methods listed are: `clear`, `copy`, `fromkeys`, `get`, `items`, `keys`, `pop`, `popitem`, `setdefault`, and `update`. To the right of the menu, the dictionary's content is visible: `{'Pedro':1000, 'Carlos':750, 'Ricardo':2000, 'Letícia':2500}`.

Vamos estudar os principais métodos presentes nos dicionários. Como mencionado os dicionários são formados por estruturas do tipo **chave:valor**.

Muitas vezes é útil conhecer as chaves presentes em um dicionário. Para isso, o objeto dicionário nos fornece o método `.keys()`.



A screenshot of a Jupyter Notebook cell. The code inputs are:

```
In [1]: # Exemplo de dicionário  
salario_dict = {'Pedro':1000, 'Carlos':750, 'Ricardo':2000, 'Letícia':2500}
```

```
In [2]: # Retorna as chaves do dicionário  
salario_dict.keys()
```

The output is:

```
Out[2]: dict_keys(['Pedro', 'Carlos', 'Ricardo', 'Letícia'])
```

Podemos também ter acesso aos valores das chaves a partir do método `.values()`:

```
In [3]: salario_dict.values()  
Out[3]: dict_values([1000, 750, 2000, 2500])
```

Caso tenhamos dois dicionários e o interesse for juntá-los, temos o método **.update()**. Veja abaixo como funciona:

```
In [1]: # Dicionário 1 contendo os nomes e respectivos pesos (Kg)  
pesos1 = {'Paulo':57, 'Priscila':65}  
  
In [2]: # Dicionário 2 contendo os nomes e respectivos pesos (Kg)  
pesos2 = {'Débora':68, 'Jefferson':70}  
  
In [3]: # Agrupa dicionário peso2 em pesos1  
pesos1.update(pesos2)  
  
In [4]: pesos1  
Out[4]: {'Débora': 68, 'Jefferson': 70, 'Paulo': 57, 'Priscila': 65}
```

Podemos **deletar** chaves e valores de um dicionário a partir da diretiva **del**:

```
In [4]: pesos1  
Out[4]: {'Débora': 68, 'Jefferson': 70, 'Paulo': 57, 'Priscila': 65}  
  
In [5]: # Deleta a chave Paulo com seu respectivo valor  
del pesos1['Paulo']  
  
In [6]: pesos1  
Out[6]: {'Débora': 68, 'Jefferson': 70, 'Priscila': 65}
```

Se o interesse for **deletar todo o conteúdo** do dicionário podemos utilizar o método **.clear()**.

```
In [6]: pesos1
Out[6]: {'Débora': 68, 'Jefferson': 70, 'Priscila': 65}

In [7]: pesos1.clear()

In [8]: pesos1
Out[8]: {}
```

No entanto, se quisermos apagar/remover **todo o dicionário** com seu conteúdo, claro, podemos utilizar o **del**:

```
In [9]: pesos2
Out[9]: {'Débora': 68, 'Jefferson': 70}

In [10]: # Deleta o dicionário pesos2 - não apenas o conteúdo
          # como é feito com o método .clear()
          del pesos2

In [11]: pesos2 # teremos um erro pois não existe mais a variável na memória
-----
NameError: name 'pesos2' is not defined
```

Vale lembrar que os dicionários em sua estrutura aceitam diferentes tipos de dados. Em outras palavras, podemos misturar os tipos de dados das chaves e valores. Veja abaixo um exemplo:

```

In [1]: # Dicionário Misto
dicionario = {'Maria':53, 77:'João', 'Nome':'Silva'}
```

```

In [2]: dicionario
```

```

Out[2]: {77: 'João', 'Maria': 53, 'Nome': 'Silva'}
```

```

In [3]: # Acessando o valor a partir da chave Maria
dicionario['Maria']
```

```

Out[3]: 53
```

```

In [4]: # Acessando o valor a partir da chave 77
dicionario[77]
```

```

Out[4]: 'João'
```

```

In [5]: # Acessando o valor a partir da chave Nome
dicionario['Nome']
```

```

Out[5]: 'Silva'
```

Podemos criar dicionários de outras estruturas, por exemplo, dicionários de listas.

Veja que interessante:

```

In [1]: dici = {'chave1':231, 'chave2':[3,4,5,3], 'chave3':['seg','ter','qua','qui','sex','sab','dom']}
```

```

In [2]: dici
```

```

Out[2]: {'chave1': 231,
         'chave2': [3, 4, 5, 3],
         'chave3': ['seg', 'ter', 'qua', 'qui', 'sex', 'sab', 'dom']}
```

Igual como fizemos para a lista, podemos estruturar dicionários aninhados, ou seja, dicionários dentro de outros dicionários.

Abaixo temos um exemplo:

```
In [1]: # Dicionários aninhados
dic2 = {'c1':1,'c2':77,'c3':{'d1':10,'d2':5,'d3':17}}
```

```
In [2]: dic2
```

```
Out[2]: {'c1': 1, 'c2': 77, 'c3': {'d1': 10, 'd2': 5, 'd3': 17}}
```

```
In [3]: dic2['c3']
```

```
Out[3]: {'d1': 10, 'd2': 5, 'd3': 17}
```

## Tuplas

As **tuplas** são estruturas bastante semelhantes às listas. A principal diferença entre tuplas e listas é que tuplas são estruturas que armazenam dados e são imutáveis. Por outro lado, como sabemos, às listas são estruturas mutáveis.

Veja as principais características das **tuplas**:

- Seu tamanho **não** pode ser **alterado**, diferentemente dos dicionários e listas;
- Seus elementos são **imutáveis**, iteráveis e ordenados;
- **Não** pode conter **múltiplos tipos** de dados.

A estrutura geral de uma tupla é basicamente a mesma de uma lista onde seus valores são separados por vírgulas, porém entre parênteses;

**tupla = ('val1', 'val2', ..., 'valN')**

Podemos notar que uma tupla é **caracterizada** pelo uso de **parênteses** (). Lembrando que em uma lista utilizamos colchetes [] e dicionários utilizamos chaves {}.

Vale ressaltar que na estrutura de uma tupla em Python não é requisito obrigatório a utilização de parênteses, podemos omiti-los. Mas para manter um padrão de conversão iremos sempre utilizar os parênteses.

Vejamos um exemplo disso:

```
In [1]: # Definição de uma tupla sem parênteses
tupla1 = 'a', 'b', 'c', 'd', 'e'

In [2]: tupla1
Out[2]: ('a', 'b', 'c', 'd', 'e')

In [3]: type(tupla1)
Out[3]: tuple

In [4]: # Definição de uma tupla sem parênteses
tupla2 = ('a', 'b', 'c', 'd', 'e')

In [5]: tupla2
Out[5]: ('a', 'b', 'c', 'd', 'e')

In [6]: type(tupla2)
Out[6]: tuple
```

Atenção a este ponto a seguir.

Devemos utilizar as tuplas quando na construção dos nossos algoritmos a imutabilidade dos dados é um requisito importante. Isso fornece uma maneira segura de garantir a integridade de dados.

Portanto, quando uma tupla for criada não podemos fazer mais nada que venha a modificar seus valores.

Note que as opções de métodos para esse tipo de estrutura são bem limitadas comparadas com as outras estruturas de dados estudadas até então:

```
In [ ]: tupla1.|  
        tupla1.count  
        tupla1.index  
In [ ]: tupla1.count.  
        tupla1.index.
```

Logo, se tivermos uma **tupla** e tentarmos adicionar um item ou **deletar** um elemento, por exemplo, teremos uma mensagem de erro pois estamos tentando modificar algo imutável.

```
In [1]: tupla = (11,1.5,154,77,'Maria')

In [2]: tupla
Out[2]: (11, 1.5, 154, 77, 'Maria')

In [3]: tupla[2]
Out[3]: 154
```

Vejamos os principais métodos disponíveis para cada um dos elementos de uma tupla:

```
In [2]: tupla
Out[2]: (11, 1.5, 154, 77, 'Maria')

In [3]: tupla[2]
Out[3]: 154

In [ ]: tupla[1].  
as_integer_ratio  
.conjugate  
.fromhex  
.hex  
.imag  
.is_integer  
.real  
.ipython_checkpoints/
```

Agora se tentarmos deletar algum elemento de uma tupla:

```
In [4]: del tupla[1]

-----
TypeError                                     Traceback (most recent call last)
<ipython-input-4-093aa513d8c6> in <module>()
      1 del tupla[1]

TypeError: 'tuple' object doesn't support item deletion
```

Por outro lado, podemos deletar uma tupla inteira assim:

```
In [6]: del tupla

In [7]: tupla

-----
NameError                                     Traceback (most recent call last)
<ipython-input-7-5dbcc13828bc> in <module>()
      1 tupla

NameError: name 'tupla' is not defined
```

Vamos agora tentar adicionar outro item a tupla:

```
In [5]: tupla[5] = 33

-----
TypeError                                     Traceback (most recent call last)
<ipython-input-5-050920957b6c> in <module>()
      1 tupla[5] = 33

TypeError: 'tuple' object does not support item assignment
```

Podemos usar alguns artifícios para fazer modificações em estruturas de tuplas. Para isso, podemos transformar uma tupla em uma lista, fazer as alterações necessárias e depois retornar para uma tupla. Veja um exemplo abaixo:

```
In [1]: # Definir uma tupla
tu1 = (1,2,3,4,5)

In [2]: # Transformar tupla em lista
lis1 = list(tu1)

In [3]: # Adicionar um valor a lista
lis1.append(6)

In [4]: # Transformar a lista em tupla
tu1 = tuple(lis1)

In [5]: tu1
Out[5]: (1, 2, 3, 4, 5, 6)
```

## Vetores e matrizes

Podemos representar conjuntos de dados com vetores. Devemos imaginar que os vetores são úteis para armazenar coleções de valores de um único tipo.

Listas, dicionários e tuplas, como já estudado, podem armazenar valores de diversos tipos. No entanto, os vetores e matrizes, em termos matemáticos, precisam conter elementos de um único tipo, que no nosso caso será o tipo numérico.

### Vetores

Precisamos de uma biblioteca Python chamada de **NumPy** [\[11\]](#) (*Numerical Python*) para trabalharmos com *arrays* (vetores de um modo geral) e matrizes multidimensionais. Assim, precisamos carregar previamente essa biblioteca com o comando *import*.

```
In [1]: import numpy as np

In [2]: # Criar um vetor
vet1 = np.array([1,2,3,4])

In [3]: vet1

Out[3]: array([1, 2, 3, 4])

In [4]: type(vet1)

Out[4]: numpy.ndarray
```

Perceba que o tipo da variável **vet1** é um ‘**numpy.ndarray**’ que representa um vetor de maneira geral.

Os elementos do vetor são todos apenas do tipo numérico (**int**, **float** ou **double**). Veja o que acontece se tentarmos misturar os tipos dos elementos:

```
In [5]: vet2 = np.array([1,2,3,'4']) # tentando adicionar str '4'

In [6]: vet2
Out[6]: array(['1', '2', '3', '4'], dtype='|U11')

In [7]: vet2[1]
Out[7]: '2'

In [8]: type(vet2[1])
Out[8]: numpy.str_
```

O que aconteceu no exemplo acima é algo bastante interessante. Podemos notar que iniciamos adicionando valores numéricos ao nosso **vet2** (1,2,3), porém o último elemento do vetor é uma **string** ('4').

Sabemos que vetores e matrizes só podem conter um tipo específico de elemento e por isso o interpretador Python decidiu transformar todos os elementos do vetor em uma **string** para manter a **consistência** de um vetor.

A pergunta que não quer calar é: **por que o Python não transformou a string '4' em um número?** Veja o exemplo abaixo que vai esclarecer esse questionamento:

```
In [9]: vet3 = np.array([7,8,9,'a'])

In [10]: vet3

Out[10]: array(['7', '8', '9', 'a'], dtype='|<U11')

In [11]: type(vet3[0])

Out[11]: numpy.str_
```

Perceba que colocamos a string ‘a’ dentro do vetor de números, portanto o Python, como esperado para manter a consistência, transformou tudo em string.

Note que não seria possível transformar a string ‘a’ em um número, por isso o Python faz a transformação de todos os elementos para o caso mais geral que nessa situação é uma string.

Atenção que esse artifício computacional é feito em outras linguagens de programação como, por exemplo, a linguagem estatística R.

## Matrizes

Quando estamos interessados em representar coleções de vetores, as matrizes são uma maneira inteligente e simplificada que nos auxiliam nessa atividade.

Os elementos das matrizes podem ser números reais, números complexos, expressões algébricas e até mesmo funções.

Matrizes são compostas por linhas e colunas. Assim como acontece para os vetores, as matrizes precisam possuir elementos de mesmo tipo. Veja o exemplo abaixo:

```
In [1]: import numpy as np

In [2]: # Define uma matriz com 2 linhas e 3 colunas
         a = np.array([[1,2,3],[3,4,5]])

In [3]: # Define uma matriz com 3 linhas e 3 colunas
         a2 = np.matrix([[1,2,3],[3,4,5]])

In [4]: type(a)

Out[4]: numpy.ndarray

In [5]: type(a2)

Out[5]: numpy.matrixlib.defmatrix.matrix
```

Note que podemos definir uma matriz de duas maneiras. De ambas as maneiras podemos executar as operações matriciais. Veja abaixo como obter elementos e fatias a partir das matrizes:

```
In [6]: # Acessar elemento da linha 2 e coluna 3
         a[1,2]

Out[6]: 5

In [7]: # Obter todos os elementos da linha 1
         a[0,:]

Out[7]: array([1, 2, 3])

In [8]: # Obter todos os elementos da coluna 2
         a2[:,1]

Out[8]: matrix([[2],
               [4]])
```

O mesmo comportamento de que precisamos manter o mesmo tipo de elementos em um vetor acontece com as matrizes:

```
In [1]: import numpy as np  
  
In [2]: b = np.array([[1,2,3],[3,4,'5']])  
  
In [3]: b  
Out[3]: array([[1, 2, 3],  
                 [3, 4, 5]], dtype='<U11')  
  
In [7]: type(b[0][0])  
Out[7]: numpy.str_
```

## Operações básicas com Vetores e Matrizes

Fazer operações com vetores e matrizes com o Python a partir da biblioteca **NumPy** é bastante simples e intuitivo.

```
In [1]: import numpy as np  
  
In [2]: # Define o vetor  
v = np.array([11,22,33])  
  
In [3]: # Define a matriz  
a = np.array([[1,2,3],[4,5,6]])  
  
In [4]: # Define a matriz  
b = np.array([[1,1,1],[2,3,3]])  
  
In [5]: # Soma o vetor com a matriz  
sva = v + a  
  
In [6]: sva  
Out[6]: array([[12, 24, 36],  
                 [15, 27, 39]])
```

Algumas operações são feitas elemento a elemento. O Python se responsabiliza de fazer a concordância operacional entre os elementos dos vetores e matrizes.

```
In [7]: # Multiplica o vetor 'v' pela matriz 'a'  
va = v*a
```

```
In [8]: va
```

```
Out[8]: array([[ 11,  44,  99],  
               [ 44, 110, 198]])
```

```
In [9]: # Soma dos elementos das matrizes  
c = a + b
```

```
In [10]: c
```

```
Out[10]: array([[2, 3, 4],  
                  [6, 8, 9]])
```

```
In [11]: # Operações com matrizes  
d = 2*a+b
```

```
In [12]: d
```

```
Out[12]: array([[ 3,  5,  7],  
                  [10, 13, 15]])
```

Devemos ter cuidado nas operações de multiplicação porque temos dois tipos de produtos entre matrizes: produto entre os escalares da matriz e o produto matricial. Veja no exemplo abaixo essa diferença.

```
In [13]: # Multiplica elementos das matriz
mab = a * b

In [14]: mab
Out[14]: array([[ 1,  2,  3],
   [ 8, 15, 18]])

In [15]: # Multiplicação matricial
dab = np.dot(a,b) # teremos uma mensagem de erro!

-----
ValueError                                     Traceback (most recent call last)
<ipython-input-15-4a874b5f90f6> in <module>()
      1 # Multiplicação matricial
----> 2 dab = np.dot(a,b)

ValueError: shapes (2,3) and (2,3) not aligned: 3 (dim 1) != 2 (dim 0)
```

Na multiplicação matricial precisamos de uma concordância dimensional entre o número de colunas de uma matriz com o número de linhas da outra. Veja o exemplo:

```
In [16]: # Para multiplicação matricial devemos ter
# nº de colunas de 'a' igual nº de Linhas de 'b'
dab = np.dot(a,b.T) # b.T -> transposta de 'b'

In [17]: dab
Out[17]: array([[ 6, 17],
   [15, 41]])
```

Como a matriz ‘a’ apresenta três colunas e a matriz ‘b’ possui apenas duas linhas, foi preciso obter a transposta de ‘b’ transformando as linhas em colunas e colunas em linhas. Dessa forma foi possível obter a consistência necessária para o produto matricial.

Se você tem interesse em conhecer como trabalhar e manipular vetores e matrizes de maneira mais detalhada poderá adquirir o e-

book: [Álgebra Linear com Python: Aprenda na prática os principais conceitos.](#)

Este curso online com preço muito especial pode ajudar você nos estudos sobre matrizes – [AQUI](#).



# **Capítulo 3**

## Condicionais, laços de repetição e listas

Antes de prosseguirmos com este capítulo, vamos a um breve resumo do que já foi estudado até o presente momento:

- Estruturas básicas do Python;
- Os tipos de variáveis;
- Os principais operadores lógicos e matemáticos;
- Estruturas de dados: listas, dicionários, tuplas e vetores e matrizes.

É isso aí programador(a). Já estamos ficando *experts* na linguagem Python.

Iremos agora conectar tudo o que foi aprendido dentro de estruturas lógicas mais elaboradas.

O principal poder fornecido por uma linguagem de programação será apresentado neste capítulo.

Nós como programadores precisamos rotineiramente trabalhar com estruturas de condições lógicas para modificar a direção de andamento de um programa.

Também é frequente a necessidade de efetuar operações repetitivas e para isso temos os laços de repetição.

Contudo, vamos começar estudando os condicionais if/else/elif

## Condicionais if/else/elif

Programar é escolher direcionamentos e condicionantes para um determinado objetivo. Nesse sentido, praticamente toda linguagem de programação apresenta um conjunto de estruturas condicionais que possibilita direcionar nosso código para caminhos mais específicos baseados nas estruturas lógicas do nosso problema.

Constantemente, precisaremos avaliar determinada condição lógica e tomar uma ação a partir disso. Assim, iremos estudar como executar expressões lógicas para permitir que o computador tome decisões baseadas em critérios específicos.

### Condicional if

Muitas vezes precisamos determinar um conjunto de ações para que as máquinas solucionem algum problema. Nisso, a estrutura ‘if’ (**se**) nos ajuda de forma simples e rápida. Essa diretiva faz a seguinte ordem: se (if) determinada condição lógica for verdadeira, execute uma ação.

Temos a seguinte estrutura de declaração do **if** no Python:

**if (condição a ser avaliada):**

Bloco (**indentado**) de execução de comandos (executar algo)

Veja o exemplo abaixo:

```
In [1]: # Entrada de valores  
A = int(input('Digite o valor de A: '))  
B = int(input('Digite o valor de B: '))
```

```
Digite o valor de A: 10  
Digite o valor de B: 7
```

```
In [2]: if(A > B):  
    print('A é maior do que B!')  
  
if(A < B):  
    print('B é maior do que A!')
```

```
A é maior do que B!
```

Note, no exemplo acima, que a função ‘**input**’ lança para o usuário a opção de escolher um valor para as variáveis A e B.

O bloco de condicionais vai avaliar e mostrar na tela se o valor digitado para **A** é maior ou menor que o valor **B**.

Uma importante observação é que Python utiliza o **recuo (indentação)** à direita para prosseguir com a estrutura de comando pertencentes ao condicional ‘**if**’.

Em outras linguagens de programação são utilizadas palavras ou caracteres especiais como **BEGIN** e **END** (no caso da linguagem Pascal) ou as chaves como em Java, C#, ou C++.

### Condisional else

Podemos melhorar nosso código do último exemplo acima, deixando-o mais legível e menos verboso a partir do uso do ‘**else**’ (que pode ser traduzido como: caso contrário). Ou seja, ‘**if**’ (se determinada condição for aceita execute a estrutura) ‘**else**’ (caso contrário) execute esta outra estrutura de comandos.

Veja um exemplo prático abaixo:

```
In [3]: if(A > B):
    print('A é maior do que B!')
else:
    print('B é maior do que A!')
```

```
A é maior do que B!
```

Podemos fazer aninhamentos (estruturas dentro de outras estruturas) com **if** e **else**:

```
In [1]: idade = 80
sex = 'F'
```

```
In [2]: if idade < 9:
    print('É uma criança')
    if sex == 'M':
        print('Sexo masculino')
    else:
        print('Sexo feminino')
else:
    print('Não é uma criança!')
    if sex == 'M':
        print('Sexo masculino')
    else:
        print('Sexo feminino')
```

```
Não é uma criança!
Sexo feminino
```

Perceba que trabalhar com programação é algo parecido como escrever uma receita de bolo. Se você gosta de cozinhar então deve está se divertindo à beça.

No fundo, no cerne de tudo estamos orientando as máquinas. Estamos dando ordens para que elas executem atividades padronizadas e repetitivas.

Vejamos outro caso.

```
In [3]: idade = 7
sex = 'M'

In [4]: if idade < 9:
    print('É uma criança')
    if sex == 'M':
        print('Sexo masculino')
    else:
        print('Sexo feminino')
else:
    print('Não é uma criança!')
    if sex == 'M':
        print('Sexo masculino')
    else:
        print('Sexo feminino')
```

```
É uma criança
Sexo masculino
```

Vamos a outro exemplo, preste muita **atenção** neste:

```
In [1]: nota = 5

In [2]: if nota <= 1 and nota > 0:
    print('Nota muito baixa!')
else:
    if nota > 1 and nota <=4:
        print('Nota baixa.')
    else:
        if nota > 4 and nota <= 7:
            print('Nota na média.')
    else:
        if nota > 7 and nota <= 10:
            print('Nota alta!')
    else:
        print('Nota inválida!')
```

```
Nota na média.
```

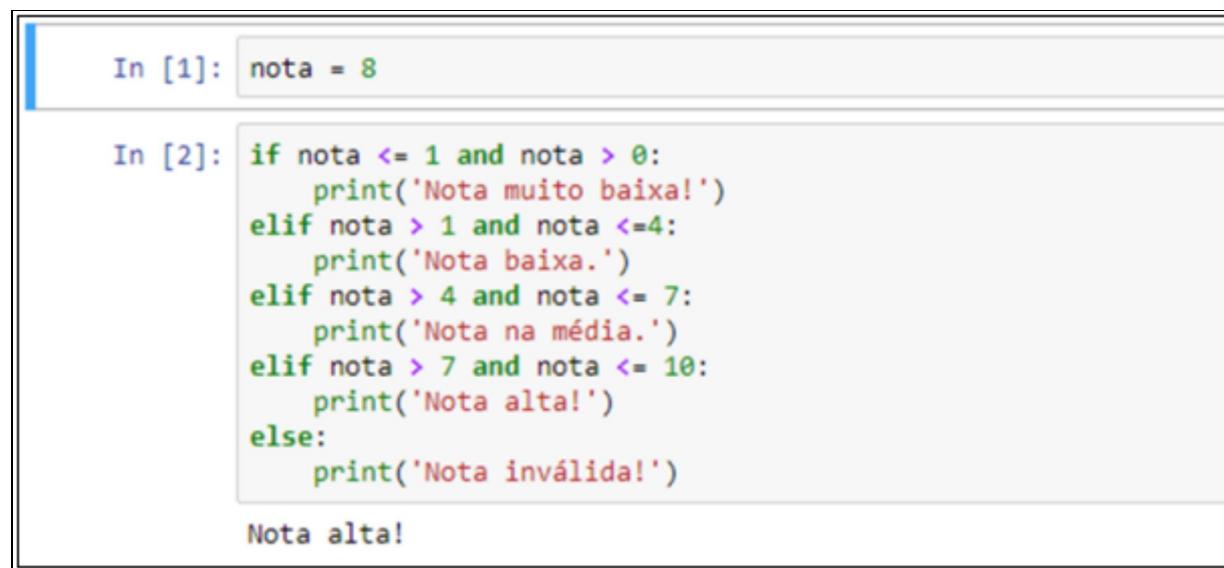
Perceba, com o exemplo acima, que estruturamos critérios lógicos para que a nota fique bem determinada dentro de 0 a 10.

Qualquer valor de nota abaixo de ZERO ou acima de DEZ resultará em uma ‘Nota inválida’. Desse modo, presenciamos um exemplo de aninhamento de estruturas **if** e **else**.

### Condicional **Elif**

Se precisarmos fazer múltiplas avaliações de blocos lógicos sem ter que ficar criando outros níveis de **if** e **else** podemos utilizar a **elif**.

Vejamos como funciona esse elemento condicional a partir de modificações do último exemplo apresentado:



In [1]: nota = 8

```
In [2]: if nota <= 1 and nota > 0:  
    print('Nota muito baixa!')  
elif nota > 1 and nota <=4:  
    print('Nota baixa.')  
elif nota > 4 and nota <= 7:  
    print('Nota na média.')  
elif nota > 7 and nota <= 10:  
    print('Nota alta!')  
else:  
    print('Nota inválida!')
```

Nota alta!

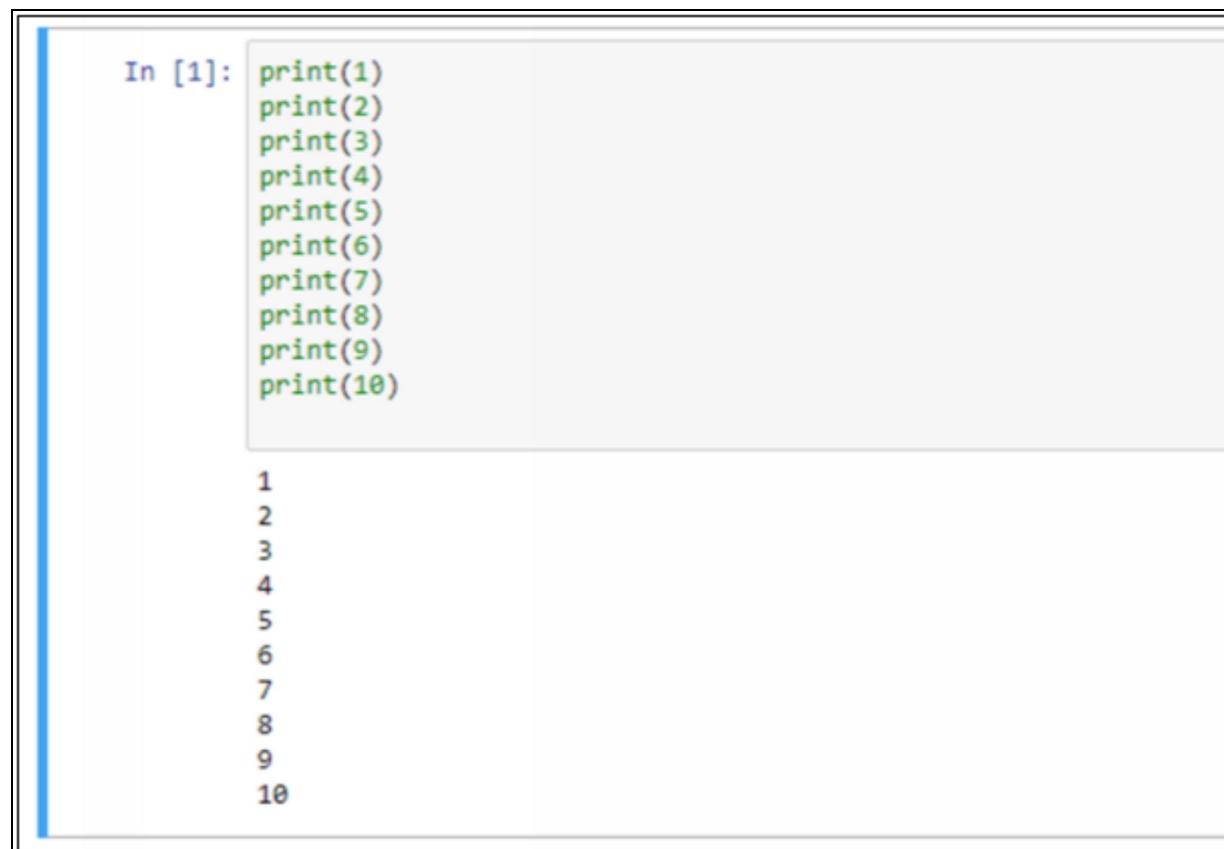
Desse modo, com o uso do ‘**elif**’ nosso código ficou muito mais enxuto e legível.

Temos um ganho significativo com a estrutura de indentação onde não precisamos fazer aninhamentos que possam atrapalhar no entendimento do código.

## Laços de Repetição **for** e **while**

Atividades repetitivas são muito frequentes nos algoritmos. Por isso, as estruturas de repetição representam a base de vários programas. Utilizamos essas estruturas quando precisamos que alguma atividade seja executada um número específico de vezes (**FOR**) ou até que uma condição lógica seja atingida (**WHILE**).

Por exemplo, vamos supor que precisamos imprimir na tela os números de **UM** a **DEZ**. Poderíamos fazer o seguinte:



```
In [1]: print(1)
print(2)
print(3)
print(4)
print(5)
print(6)
print(7)
print(8)
print(9)
print(10)

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

Isso não pareceu ser algo prático de ser feito, né verdade? Podemos fazer isso de maneira mais inteligente com o laço de repetição **FOR**.

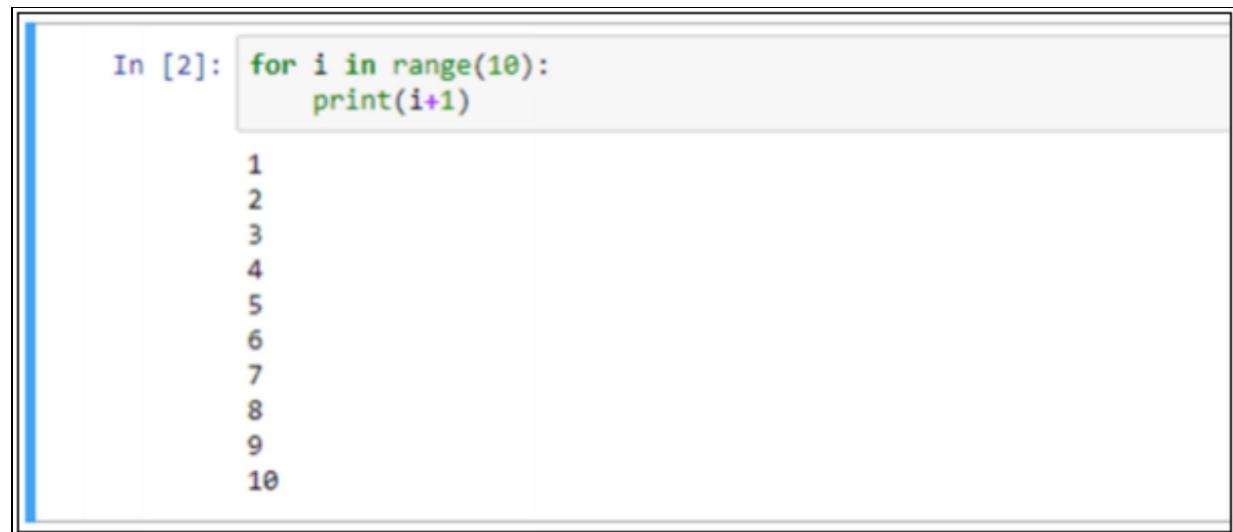
## Loop for

Uma maneira muito mais prática de fazer a mesma coisa é a partir do uso do ‘**for**’. A forma geral do uso do loop for é a seguinte:

```
for item in conjunto-de-ítems:  
    execute comandos
```

Lembre-se sempre da indentação. Para ajudar fique esperto(a) que toda vez que utilizamos os dois pontos ‘:’ depois de alguma diretiva os códigos que virão na linha seguinte deverão possuir indentação.

Veja o exemplo abaixo e perceba que programar é basicamente conversar com o computador:



In [2]: `for i in range(10):  
 print(i+1)`

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

O laço ‘**for**’ em Python é algo que empodera em muito a vida de um programador. Utilizando ‘**for**’ a linguagem Python permite iterações dentro de listas de uma maneira bastante intuitiva e simples.

Perceba que utilizamos o **range(10)** para gerar um intervalo de valores de **0** a **9** (lembrando que a indexação sempre começa com

zero). Por isso foi necessário somar uma unidade no `print(i+1)`. Poderíamos ter utilizado o `range(1,11)` e encontrar o mesmo resultado:

```
In [3]: for i in range(1,11):
    print(i)

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

Observe que para o `range(1,11)` o elemento 11 não está incluso no nosso range de valores. Vale lembrar também que `range()` é uma função ***built-in***. Tá vendo como essas funções são essenciais e importantes?

Muitas vezes precisamos de um **range** de valores dentro de um determinado intervalo com incremento especificado. Veja abaixo um exemplo de valores entre 0 a 20 com incremento de 3 unidades.

```
In [4]: for i in range(0,20,3): # zero a 20 com incremento de 3
    print(i)

0
3
6
9
12
15
18
```

Podemos fazer com que o Python faça um passeio por dentro dos valores de uma lista/dicionários/tuplas quaisquer. Veja como funciona:

```
In [1]: lista = [10,77,88,32,100]

In [2]: for i in lista:
    print(i)

10
77
88
32
100
```

Também podemos colocar estruturas condicionais dentro dos nossos *loops*.

Vamos fazer um exemplo no qual apenas os números pares de uma lista de valores irão se impressos na tela:

```
In [1]: valores = [11, 55, 22, 16, 10, 7, 9, 68]

In [2]: for v in valores:
    if v%2 == 0: # se resto da divisão por 2 for nula
        print(v)

22
16
10
68
```

O interessante é que podemos executar diversas operações matemáticas com cada um dos elementos das listas:

```
In [3]: lista = [10,77,88,32,100]

for i in lista:
    # fazer alguma operação matemática com os elementos
    # da lista:
    resp = (i+2)/10
    print(resp)

1.2
7.9
9.0
3.4
10.2
```

Veja abaixo um exemplo de *loop ‘for’* aplicado a dicionários:

```
In [4]: # dicionário:
dic = {'var1':10,'var2':20,'var3':30,'var4':40}

In [5]: for i in dic:
         print(i)

var1
var2
var3
var4
```

Note que o iterando ‘i’ para uma estrutura de dicionário retorna apenas a chave. Por vezes, podemos precisar das chaves e dos valores dentro de uma estrutura de dicionário. Veja abaixo como fazer isso:

```
In [7]: for c,v in dic.items():
    print(c,v) # mostra c-chave, v-valor

var1 10
var2 20
var3 30
var4 40
```

Agora um exemplo utilizando tuplas:

```
In [6]: # tupla:
tupla = (11,22,33,44,55)

In [7]: for i in tupla:
    print(i)

11
22
33
44
55
```

Vamos estudar *loops* aninhados, ou seja, *loop* dentro de outro *loop*. Esse tipo de estrutura de repetição é muito utilizado quando estamos trabalhando com vetores e matrizes, por exemplo.

```
In [1]: lista = [[1,2,3],[4,5,6,7,7,8],[11,12,13,14,15,16,17,18]]

In [2]: for i in lista:
    print(i)

[1, 2, 3]
[4, 5, 6, 7, 7, 8]
[11, 12, 13, 14, 15, 16, 17, 18]
```

Podemos observar que o valor do iterando ‘i’ agora é uma lista e não mais um valor apenas. Assim podemos fazer outro *loop* dentro de cada um desses elementos das listas ‘i’. Vejamos como fica:

```
In [3]: for i in lista:  
    for j in i:  
        print(j)  
    print('-----')  
  
1  
2  
3  
-----  
4  
5  
6  
7  
7  
8  
-----  
11  
12  
13  
14  
15  
16  
17  
18  
-----
```

Estamos fazendo uma nova iteração ‘j’ onde cada valor deste iterando está contido dentro de cada uma das listas ‘i’.

Vale ressaltar que podemos chamar as variáveis iteradoras dentro dos *loops* ‘for’ da maneira que melhor nos agrade. Vejamos o exemplo abaixo com o mesmo resultado:

```
In [4]: for li in lista:  
    for var in li:  
        print(var)  
    print('-----')  
  
1  
2  
3  
-----  
4  
5  
6  
7  
7  
8  
-----  
11  
12  
13  
14  
15  
16  
17  
18  
-----
```

## Loop while

Quando for necessária uma condição de repetição baseada em algum critério lógico, a melhor estratégia para isso pode ser através do uso do *loop ‘while’*.

Veja abaixo a estrutura geral de funcionamento desse **loop**:

**while condicao-logica:**  
    **<bloco de instrucoes>**

Nesse tipo de estrutura temos que ter bastante cuidado para não gerarmos um loop **infinito**.

Nosso código precisa estar estruturado para terminar a execução do *loop* uma vez que determinada condição lógica for atingida.

Contudo, a instrução **while** será executada de modo **repetitivo**, uma única vez ou inúmeras vezes, até que uma condição lógica **esteja verdadeira**.

Veja como imprimir valores de **1** a **10** como fizemos com o *loop* ‘**for**’:

```
In [1]: cont = 1

while cont <= 10:
    print(cont)
    cont = cont + 1 # temos que add essa linha para
                    # não termos um Loop infinito

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

Vamos estudar agora algo muito útil. Dentro dos nossos comandos *loops* ‘**for**’ ou ‘**while**’ podemos mudar o seu comportamento a partir das diretivas: **pass**, **break** ou **continue**.

**pass**: *não executa nada na estrutura, dando continuidade ao laço de repetição*.

**break**: *termina com o laço de repetição*.

**continue**: *pula uma iteração passando para a próxima seguinte*.

A diferença entre o **pass** e o **continue** é bastante sutil, todavia importante. Fique atento(a) no exemplo a seguir para compreender melhor essa diferença:

```
In [7]: for letra in "programador":  
    if letra == 'a':  
        pass  
    print(letra)
```

```
p  
r  
o  
g  
r  
a  
m  
a  
d  
o  
r
```

Note que quando o **if** é executado, ou seja, quando encontramos a primeira letra igual a ‘a’ o **pass** faz com que prossigamos com o *loop* sem entrarmos mais na estrutura **if** no próximo iterando.

Por outro lado, se utilizarmos no lugar do **pass** a diretiva **continue** sempre iremos avaliar a estrutura ‘if’ e toda vez que a letra for encontrada iremos passar para a iteração seguinte antes de imprimir a letra na tela.

Veja o exemplo abaixo para maiores esclarecimentos:

```
In [8]: for letra in "programador":  
    if letra == 'a':  
        continue  
    print(letra)
```

```
p  
r  
o  
g  
r  
a  
m  
d  
o  
r
```

Caso coloquemos um **break** no lugar do **continue** teremos uma interrupção no *loop* quando o **if** for verdadeiro, ou seja, **letra == 'a'**.

```
In [9]: for letra in "programador":  
    if letra == 'a':  
        break  
    print(letra)
```

```
p  
r  
o  
g
```

Vamos agora nos divertir com um joguinho bastante interessante de ser programado. Nele iremos utilizar várias das estruturas estudadas até aqui.

### Jogo de adivinhação de número.

As regras são bastante simples. Vamos a elas:

- O jogador deverá escolher um **número** qualquer **entre 0 e 50**;

- O jogador receberá dicas se seu **chute** está acima ou abaixo do valor **oculto** gerado pelo computador de maneira **aleatória**, caso não acerte de primeira o número;
- O jogador terá **10 tentativas** para **adivinar** qual foi o número oculto gerado pelo computador entre o intervalo (0 a 50).

```
In [1]: import numpy as np

In [2]: n_tentativas = 10

In [3]: print('Adivinhe o número entre 1 a 50! Você tem ',n_tentativas, ' chances!')
Adivinhe o número entre 1 a 50! Você tem 10 chances!

In [*]: # Gera um número aleatório inteiro entre 0 e 100:
num_oculto = np.random.randint(0,50)

while n_tentativas > 0:
    n_tentativas = n_tentativas - 1
    chute = int(input('Escolha um número: '))
    if chute > num_oculto:
        print('O número oculto é menor!')
        print('Nº restante de tentativas = %d' % n_tentativas)
    elif chute < num_oculto:
        print('O número oculto é maior!')
        print('Nº restante de tentativas = %d' % n_tentativas)
    else:
        print('Parabéns! Você acertou o nº que é: %d' % num_oculto)
        break

if n_tentativas == 0:
    print('-----')
    print('Infelizmente suas tentativas acabaram!')
    print('O nº oculto era: %d' % num_oculto)

Escolha um número: 
```

Veja um exemplo onde se perde no jogo:

```
Escolha um número: 25
O número oculto é maior!
Nº restante de tentativas = 9
Escolha um número: 30
O número oculto é maior!
Nº restante de tentativas = 8
Escolha um número: 25
O número oculto é maior!
Nº restante de tentativas = 7
Escolha um número: 40
O número oculto é menor!
Nº restante de tentativas = 6
Escolha um número: 39
O número oculto é menor!
Nº restante de tentativas = 5
Escolha um número: 37
O número oculto é menor!
Nº restante de tentativas = 4
Escolha um número: 36
O número oculto é menor!
Nº restante de tentativas = 3
Escolha um número: 35
O número oculto é menor!
Nº restante de tentativas = 2
Escolha um número: 32
O número oculto é maior!
Nº restante de tentativas = 1
Escolha um número: 33
O número oculto é maior!
Nº restante de tentativas = 0
-----
Infelizmente suas tentativas acabaram!
O nº oculto era: 34
```

E agora um exemplo onde se vence:

```
Escolha um número: 30
O número oculto é maior!
Nº restante de tentativas = 9
Escolha um número: 40
O número oculto é menor!
Nº restante de tentativas = 8
Escolha um número: 35
O número oculto é maior!
Nº restante de tentativas = 7
Escolha um número: 37
O número oculto é maior!
Nº restante de tentativas = 6
Escolha um número: 38
Parabéns! Você acertou o nº que é: 38
```

Muito divertido esse exemplo né verdade? Está vendo como você está evoluindo na linguagem Python. Até joguinhos já está conseguindo fazer.

Agora o universo é o limite para sua imaginação e criatividade. Comece agora mesmo a criar seus próprios algoritmos transferindo suas ideias para o computador.

O que vai tornar você um profissional de excelência não são boas notas em provas ou estudar em universidades e faculdades reconhecidas e renomadas, mas sim a capacidade que você tem de transformar ideias em algo concreto (em algoritmos funcionais) útil para as pessoas.

Vamos em frente com muita vontade, garra e esforço. A jornada é longa, mas todos nós temos o potencial de chegar lá.

Pare um pouco reflita sobre tudo o que foi aprendido até então. Tome um bom gole de água ou café estique o corpo e vamos para o próximo capítulo.



# **Capítulo 4**

## Funções

Muitas etapas de um algoritmo podem ser aglomeradas/aglutinadas e representadas por funções.

Quando uma mesma ação aparece repetidas vezes no decorrer do nosso algoritmo, uma maneira interessante de simplificar as estruturas dos códigos é a partir da criação de funções.

E o que são funções para a linguagem Python? Funções são agrupamentos de instruções que são executadas a partir de um conjunto de informações de entradas gerando saídas processadas.

Com as funções podemos executar diversas vezes um conjunto de instruções em códigos mais complexos sem a necessidade de reescrever essas instruções rotineiras.

Até o presente momento fizemos uso de diversas funções úteis tais como: `len()`, `range()`, `list()`, `int()`, dentre outras.

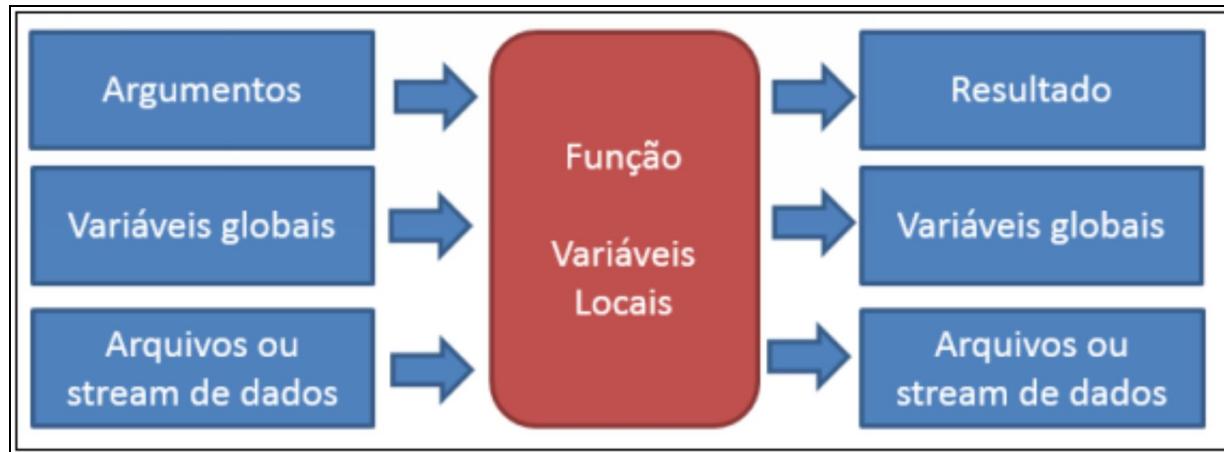
Vamos agora aprender a criar nossas próprias funções em Python.

De modo geral, em Python, as funções são representadas da seguinte maneira:

```
def nom_fun(arg1, arg2, ..., argn):
    <corpo_da_funcao>
    <retorno_da_funcao>
```

As funções são muito importantes, pois podemos condensar códigos que frequentemente precisamos utilizar em diversas etapas da solução dos nossos problemas.

Veja abaixo um esquema que mostra o princípio de funcionamento das funções:



As funções também nos ajudam a tornar nosso código mais legível e intuitivo. Vamos aos exemplos práticos.

Suponha que estamos interessados em criar uma calculadora e para isso precisamos desenvolver as funções para as quatro operações básicas: **soma**, **subtração**, **multiplicação** e **divisão**.

```

In [1]: def soma(a,b):
          return a+b

def subtrai(a,b):
          return a-b

def multiplica(a,b):
          return a*b

def divide(a,b):
          if b != 0:
              return a/b
          else:
              return 'Erro: divisão por zero!'

```

Agora que temos nossas funções criadas, ou seja, elas estão armazenadas na memória do computador, vamos utilizá-las:

```
In [2]: x = 10
y = 20

print('Soma de x com y :',soma(x,y))
print('Subtração de x com y :',subtrai(x,y))
print('Multiplicação de x com y :',multiplica(x,y))
print('Divisão de x com y :',divide(x,y))

Soma de x com y : 30
Subtração de x com y : -10
Multiplicação de x com y : 200
Divisão de x com y : 0.5
```

Podemos utilizar tantos parâmetros de entrada quantos forem necessários para o processamento de nossa função.

A função pode também **não** ter nenhum parâmetro de entrada e retorno. Veja um exemplo abaixo:

```
In [6]: def ola_mundo():
    print('Olá Mundo!')
```

```
In [7]: ola_mundo()
```

```
Olá Mundo!
```

Notou a utilidade e flexibilidade das funções? Pois bem a partir de agora passe sempre que possível a utilizar essa facilidade nos seus códigos.

## Variáveis locais e globais - funções

Quando começamos a trabalhar com funções, o conceito de **variável local e global** se torna útil.

Variável local, como o próprio nome diz, é aquela variável criada dentro de um contexto local, uma função, por exemplo. Por outro lado, variáveis globais podem ser utilizadas de maneira geral dentro do nosso algoritmo, seja dentro de funções ou fora.

Assim, devemos ter cuidado, pois variáveis declaradas **dentro** das **funções** apresentam **escopo local** enquanto variáveis no corpo do nosso código, **fora** das **funções**, possuem **escopo global**.

Isso tudo deu um nó na sua cabeça? Não se preocupe que no início realmente isso parece um pouco confuso. Aos poucos você irá entendendo melhor.

Tenha certeza que a compreensão da diferença entre variáveis locais e globais é **extremamente** importante. Releia essa parte quantas vezes forem necessárias. Não saia daqui sem entender bem esse ponto.

Veja o exemplo abaixo para elucidar melhor tudo isso:

```
In [1]: # Variável global  
var_resp = 7  
  
def soma_dois_numeros(a,b):  
    # var_resp = variável local  
    # serve apenas para o contexto da função  
    var_resp = a + b  
    return(var_resp)
```

```
In [2]: soma_dois_numeros(3,20)
```

```
Out[2]: 23
```

```
In [3]: # Mostrar a var_resp  
print(var_resp)
```

```
7
```

Para fixar melhor o conceito, devemos perceber que a ‘**var\_resp**’ dentro da função ‘**soma\_dois\_numeros()**’ é local e, portanto, é compreendida pela Python apenas dentro do escopo da função. No entanto, a ‘**var\_resp**’ fora da função tem a abrangência global e quando pedimos para imprimir com o ‘**print**’ o valor dela é retornado.

Bastante atenção agora. Não é uma boa prática de programação utilizar o mesmo nome para variáveis locais e globais, pois podemos nos confundir na utilização. Esse tipo de cuidado certamente evitará diversos erros de compilação, inclusive erros que muitas vezes são difíceis de serem encontrados.

Vamos a outro exemplo:

```
In [4]: # Variável global
var_resp = 7

def soma_dois_numeros(a,b):
    # var_resp = variável Local
    # serve apenas para o contexto da função
    var_resp_loc = a + b
    return(var_resp_loc)

In [5]: # Mostrar a variável Local: var_resp_loc
print(var_resp_loc)
```

---

```
NameError Traceback (most recent call last)
<ipython-input-5-ab1757614e0d> in <module>()
      1 # Mostrar a var_resp_loc
----> 2 print(var_resp_loc)

NameError: name 'var_resp_loc' is not defined
```

Ao tentarmos imprimir a variável local ‘`var_resp_loc`’ obtivemos uma mensagem de variável não definida, uma vez que estamos no escopo global pedindo para que seja impressa uma variável local.

## Funções úteis no Python

No decorrer do desenvolvimento de um programa computacional, iremos utilizar continuamente funções. Muitas dessas funções já estão previamente desenvolvidas e testadas e estão à nossa disposição para uso imediato.

É importante conhecer as principais funções para não corremos o risco de começar a reinventar a roda.

Por isso, devemos dar uma pesquisada na documentação do Python a fim de verificar se já existe função criada para uma determinada atividade que estamos precisando.

Muitas vezes temos pacotes ou bibliotecas de funções especializadas na solução de problemas específicos. Assim, uma pesquisa no repositório de pacotes Python (<https://pypi.org/>) é crucial antes de iniciarmos o desenvolvimento de um novo projeto.

Tome nota das principais funções ***built-in*** que certamente irão fazer parte de muitos dos programas que iremos desenvolver no futuro.

***str()*** - transforma valores numéricos em strings.

***int()*** - transforma variáveis (string/numéricas) em inteiros.

***float()*** - transforma variáveis (string/numéricas) em decimais (ponto flutuante).

***max()*** - encontra o máximo valor de um conjunto de dados.

***min()*** - encontra o valor mínimo de um conjunto de dados.

***len()*** - retorna o comprimento de uma lista, tupla, dicionário, vetor, matriz etc.

***sum()*** - soma todos os elementos de uma lista.

Os exemplos abaixo mostram o cuidado que devemos ter com a estruturação dos tipos de variáveis na linguagem Python. Iremos

utilizar as funções *built-in* para nos ajudar com os problemas.

```
In [1]: # variável
nota = input('Digite a sua nota: ')

if nota > 7:
    print('Aluno aprovado com nota = ', nota)
else:
    print('Aluno reprovado com nota = ', nota)

-----
Digite a sua nota: 8

-----
TypeError                                     Traceback (most recent call last)
<ipython-input-1-1ef252194256> in <module>()
      2 nota = input('Digite a sua nota: ')
      3
----> 4 if nota > 7:
      5     print('Aluno aprovado com nota = ', nota)
      6 else:

TypeError: '>' not supported between instances of 'str' and 'int'
```

O tipo de erro acusado acima mostra que, dentro da condicional ‘**if**’, estamos avaliando se uma *string* (valor da nota digitada pelo usuário) é maior do que 7. Portanto, precisamos transformar essa *string* (*nota*) em um tipo ‘**int**’ ou ‘**float**’. Veja como fica o código:

```
In [2]: # variável
nota = input('Digite a sua nota: ')

if float(nota) > 7:
    print('Aluno aprovado com nota = ', nota)
else:
    print('Aluno reprovado com nota = ', nota)

-----
Digite a sua nota: 9
Aluno aprovado com nota =  9
```

Veja abaixo o uso das outras funções *built-in* comentadas anteriormente:

```
In [1]: lista_num = [1, 3.14, '7', 1.618, '99', 8, 33, 77]
In [2]: lista_num
Out[2]: [1, 3.14, '7', 1.618, '99', 8, 33, 77]
In [3]: # Tamanho da lista:
        len(lista_num)
Out[3]: 8
```

Dentro da lista ‘`lista_num`’ temos valores inteiros, *floats* e *strings*. Se tentarmos encontrar o valor máximo (`max()`) ou mínimo (`min()`) dentro dessa lista iremos ter a seguinte mensagem de erro:

```
In [4]: # Máximo valor da lista
        max(lista_num)
-----
TypeError                                         Traceback (most recent call last)
<ipython-input-4-f672c8b24f56> in <module>()
      1 # Máximo valor da lista
----> 2 max(lista_num)

TypeError: '>' not supported between instances of 'str' and 'float'
```

Por isso devemos primeiro garantir que todos os elementos da lista possuem valores numéricos, sem strings. Veja um exemplo abaixo de como converter todos os elementos da lista (`lista_num`) em números inteiros.

```
In [5]: # variável para adicionar lista de inteiros
lista_int = []

for i in lista_num:
    lista_int.append(int(i))

lista_int

Out[5]: [1, 3, 7, 1, 99, 8, 33, 77]
```

Agora vamos estudar como fazer a conversão dos valores da lista (lista\_num) em números do tipo **float**:

```
In [6]: # variável para adicionar lista de floats
lista_float = []

for i in lista_num:
    lista_float.append(float(i))

lista_float

Out[6]: [1.0, 3.14, 7.0, 1.618, 99.0, 8.0, 33.0, 77.0]
```

Veja como se dar a conversão para *strings*:

```
In [7]: # variável para adicionar lista de strings
lista_string = []

for i in lista_num:
    lista_string.append(str(i))

lista_string

Out[7]: ['1', '3.14', '7', '1.618', '99', '8', '33', '77']
```

Agora, podemos encontrar os valores máximos e mínimos dentro das listas: **lista\_int** ou **lista\_float**, pois todos os elementos são do mesmo tipo. Perceba que programação exige consistência e muita lógica.

```
In [9]: # valor máximo  
max(lista_float)  
  
Out[9]: 99.0  
  
In [10]: # valor mínimo  
min(lista_float)  
  
Out[10]: 1.0  
  
In [11]: # Soma de todos os valores da lista  
sum(lista_float)  
  
Out[11]: 229.75799999999998
```

### Funções com parâmetros indeterminados

Caso seja preciso criar uma função na qual não sabemos bem ao certo a quantidade de parâmetros de entrada, podemos utilizar o número de argumentos indeterminado com o uso de um **\*** (**asterisco**). Veja um exemplo abaixo desse tipo de uso.

```
In [1]: # Função com argumentos indeterminados
def soma_valores(*valores):
    soma = 0
    for i in valores:
        soma = soma + i
    return(soma)
```

```
In [2]: soma_valores(1,2)
```

```
Out[2]: 3
```

```
In [3]: soma_valores(1,2,3,4)
```

```
Out[3]: 10
```

```
In [4]: soma_valores(10,20,30,40,50,60,70,80,90,100)
```

```
Out[4]: 550
```

Muito interessante essa possibilidade de deixar os argumentos de entrada em aberto para valores quaisquer né verdade?

### Tópico especial: Funções *Lambda*

Um bom programador, principalmente aqueles voltados para ciência e análise de dados, precisa conhecer bem e aplicar as **funções lambdas**.

As funções lambdas são também conhecidas como funções *inline* ou **funções anônimas**.

Para quem está iniciando, as funções anônimas podem parecer algo confuso e difícil de serem aplicadas.

Porém, à medida que vamos evoluindo com o uso, aos poucos percebemos que esses tipos de funções são simples e facilitam bastante nossas vidas como programadores/analistascientistas.

O principal benefício das funções lambda é que, a partir delas, podemos fazer rapidamente funções para finalidades específicas sem a necessidade de definir uma função usando a palavra reservada ‘**def**’, como aprendemos nos tópicos anteriores. Assim, as funções são criadas em tempo de execução do código. Veja que interessante!

O uso de **funções lambdas** para alguns tipos de tarefas especializadas ganham vantagem comparadas com as funções criadas a partir da palavra **def**.

O corpo da função lambda é uma expressão única, não um bloco de instruções. Analistas e cientistas de dados fazem uso rotineiro das funções lambdas em conjunto com, por exemplo, as funções **map()**, **reduce()** e **filter()**.

Veja o formato geral de uma função lambda:

**lambda** **x**: **x\*2**

Nesse exemplo, o parâmetro de entrada da função é o **x** e o retorno da função é a multiplicação deste parâmetro **x** por **2**.

Como podemos notar, temos apenas uma única instrução matemática e não um bloco de instruções como no uso da palavra **def**.

Não se preocupe se tudo isso está parecendo confuso. Iremos fazer numerosos exemplos para elucidar o uso das funções lambdas. Vejamos agora as principais diferenças entre as funções lambda e função com a palavra reservada **def**:

**def** - *internamente cria um objeto colocando o nome da função apontando para ele.*

**lambda** - *cria um objeto, entretanto o retorno é efetuado em tempo de execução tornando o código muito mais rápido.*

Acompanhe nos exemplos abaixo os principais benefícios da utilização das funções lambda. Primeiro vamos criar funções utilizando a palavra reservada **def**.

```
In [1]: # Definir função com três linhas
def duplica(x):
    resposta = x * 2
    return(resposta)

In [2]: duplica(3)

Out[2]: 6
```

Vamos agora definir a mesma função com apenas duas linhas:

```
In [3]: # Definir função com duas linhas
def duplica(x):
    return(x * 2)

In [4]: duplica(6)

Out[4]: 12
```

Podemos ainda com apenas uma linha definir a função:

```
In [5]: # Definir função com apenas uma Linhas
def duplica(x): return(x * 2)

In [6]: duplica(5)

Out[6]: 10
```

Agora vamos criar a mesma função utilizando a expressão lambda:

```
In [1]: # Utilizando a expressão Lambda
duplicar = lambda x:x*2

In [2]: duplicar(8)

Out[2]: 16
```

Com a função lambda temos mais flexibilidade, pois não precisamos definir diretamente um nome para ela. Temos com isso vários benefícios, dentre eles o ganho de desempenho computacional.

```
In [3]: # Função avalia se aluno aprovado:
aprovado = lambda nota:nota>7

In [4]: aprovado(8)

Out[4]: True

In [5]: aprovado(4)

Out[5]: False
```

Veja como criar rapidamente uma função para inverter a ordem das letras dentro de qualquer palavra:

```
In [6]: # Função para inverter ordem das Letras
inverte_letras = lambda palavra:palavra[::-1]

In [7]: inverte_letras('Python')

Out[7]: 'nohtyP'
```

Podemos adicionar vários parâmetros de entrada nas nossas funções lambdas. Vamos criar uma função para multiplicar três valores entre si.

```
In [8]: # Multiplica três valores
mult_tres_valores = lambda var1,var2,var3:var1*var2*var3

In [9]: mult_tres_valores(2,5,10)

Out[9]: 100
```

Esses foram apenas alguns exemplos da poderosa funcionalidade que temos em nossas mãos a partir das expressões ou funções lambda.

Faça agora mesmo uma pesquisa na internet para mais exemplos interessantes e úteis utilizando as funções lambda. Tem muita coisa interessante, você vai ver.



# **Capítulo 5**

## I/O de arquivos

Neste capítulo, iremos aprender uma etapa essencial para todo programador Python: abrir, editar e salvar arquivos (I/O).

I/O significa I = ***input*** (entrada), O = ***output*** (saída).

Os arquivos são uma maneira eficiente de fornecer dados de entrada e saída para os nossos programas. A partir dos arquivos podemos ler dados de outros programas e gerar saídas para alimentar banco de dados relacionais, por exemplo.

Numa linguagem mais próxima a de máquina, um arquivo é um espaço na memória rígida (disco HD) onde podemos ler, editar e gravar dados e informações. Esse espaço é gerenciado pelo sistema operacional que estamos trabalhando.

Python permite a abertura, leitura, modificação e gravação de inúmeras maneiras. Para isso temos alguns caracteres reservados que podemos utilizar nas funções de manipulação de arquivos.

Existem diversos modos de manipulação de arquivos, vamos aqui estudar os principais, que são: **r** – leitura (*read*), **w** – escrita (*write*) e modificação de conteúdo já existente, **a** – adição (*append*) de novos conteúdos em um preexistente, **b** - modo binário de leitura/gravação, **+** atualização para leitura e escrita.

Veja um exemplo de como podemos criar um arquivo e escrever valores numéricos utilizando a função **open()**.

In [ ]: # Ponteiro para um arquivo  
arq = open('dados\_fun\_quadratica.txt','w')  
  
arq.  
  buffer  
  close  
  closed  
  detach  
  encoding  
  errors  
  fileno  
  flush  
  isatty  
  line\_buffering

A screenshot of a Jupyter Notebook cell. The code has been partially typed, showing the creation of a file pointer 'arq' using the 'open' function. A tooltip is displayed over the variable 'arq.', listing its methods: buffer, close, closed, detach, encoding, errors, fileno, flush, isatty, and line\_buffering. The method 'closed' is highlighted with a blue selection bar.

O objeto ‘**arq**’ criado acima nos fornece um conjunto de métodos para manipulá-lo. Iremos utilizar o método *write* para escrever um conjunto de valores numéricos dentro de um arquivo .txt.

Note que fizemos uma especificação ‘w’ (*write* - escrita) na função **open()**. Caso o arquivo não exista, com o modo ‘w’ ele será imediatamente criado. Caso o arquivo já exista, ele será apagado e substituído. Por isso, bastante cuidado para não sobrescrever arquivos importantes.

In [3]: # Ponteiro para um arquivo  
arq = open('dados\_fun\_quadratica.txt','w')  
  
for linha in range(0,101):  
 arq.write('%d %d \n' % (linha,linha\*\*2))  
  
arq.close()

A screenshot of a Jupyter Notebook cell labeled 'In [3]'. The code has been completed from the previous snippet. It creates a file pointer 'arq' using 'open', then uses a 'for' loop to write 101 lines to the file. Each line contains two integers separated by a space, followed by a newline character. Finally, 'arq.close()' is called to close the file.

Após executar a célula, se tudo ocorrer bem, será gerado dentro do diretório onde se encontra o **Jupyter** notebook um arquivo com o nome ‘**dados\_fun\_quadratica.txt**’ contendo duas colunas de dados.

Podemos utilizar a função **open()** para abrir esse arquivo que foi criado, utilizando ‘r’ (*read* - leitura).

```
In [1]: # Abrir arquivo para leitura apenas
arq = open('dados_fun_quadratica.txt','r')

In [2]: # O método .readlines() ler todas as linhas do arq
linhas = arq.readlines()
type(linhas)

Out[2]: list

In [3]: # Tamanho da Lista
len(linhas)

Out[3]: 101
```

Como o método **.readlines()** nos retorna uma lista, podemos fazer um *print* de seus valores com um *loop for*. Veja abaixo:

```
In [4]: for linha in linhas:  
    print(linha)
```

```
arq.close()
```

```
0 0
```

```
1 1
```

```
2 4
```

```
3 9
```

```
4 16
```

```
5 25
```

```
6 36
```

```
7 49
```

```
8 64
```

```
9 81
```

Após a manipulação do arquivo é sempre uma boa prática fechá-lo com o método `.close()`.

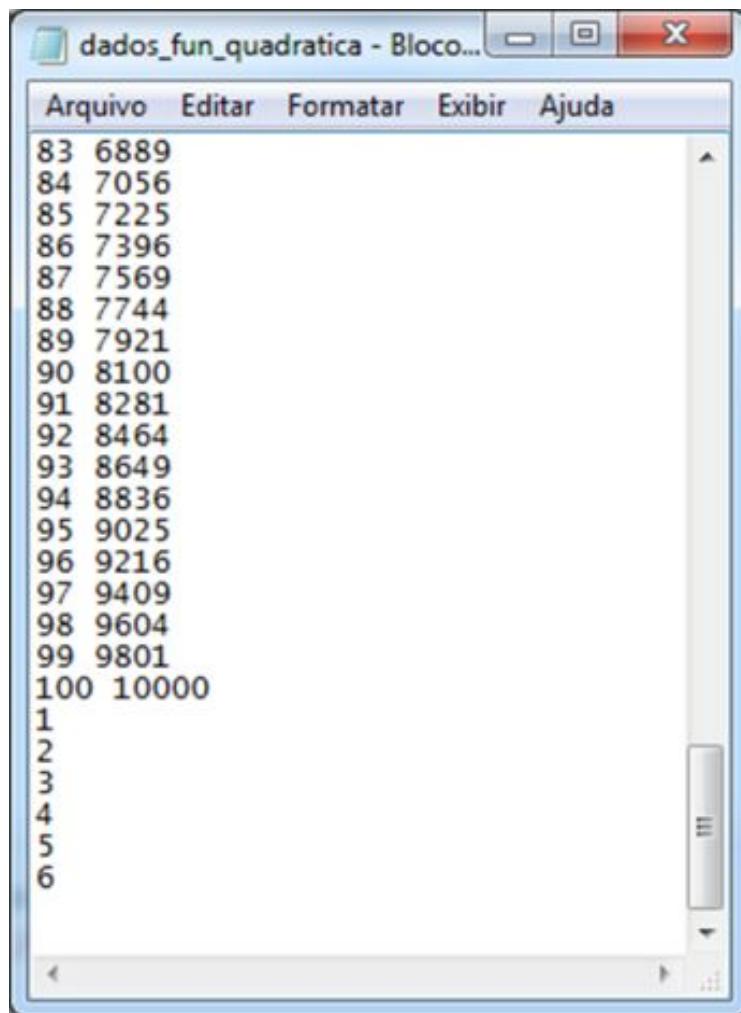
Vamos estudar agora o modo ‘a’ (**append** - adiciona informações extras em um arquivo já criado). Vamos supor que estamos interessados em colocar mais dados imediatamente abaixo dos dados do arquivo ‘dados\_fun\_quadratica.txt’.

```
In [6]: # Abrir arquivo no modo append  
arq = open('dados_fun_quadratica.txt', 'a')
```

```
In [7]: for i in range(1,7):  
    arq.write('%d \n' % i)  
arq.close()
```

Abra o arquivo **dados\_fun\_quadratica.txt** e perceba que no final temos seis linhas a mais com os números de **um a seis** ordenados.

Veja abaixo esse arquivo aberto:



A screenshot of a Windows Notepad window titled "dados\_fun\_quadratica - Bloco de notas". The window has a menu bar with "Arquivo", "Editar", "Formatar", "Exibir", and "Ajuda". The main content area contains two sets of data. The first set consists of 19 numerical values from 83 to 100, each followed by a space and a four-digit value. The second set consists of six numerical values from 1 to 6, each on a new line. The window has scroll bars on the right side.

83	6889
84	7056
85	7225
86	7396
87	7569
88	7744
89	7921
90	8100
91	8281
92	8464
93	8649
94	8836
95	9025
96	9216
97	9409
98	9604
99	9801
100	10000
1	
2	
3	
4	
5	
6	

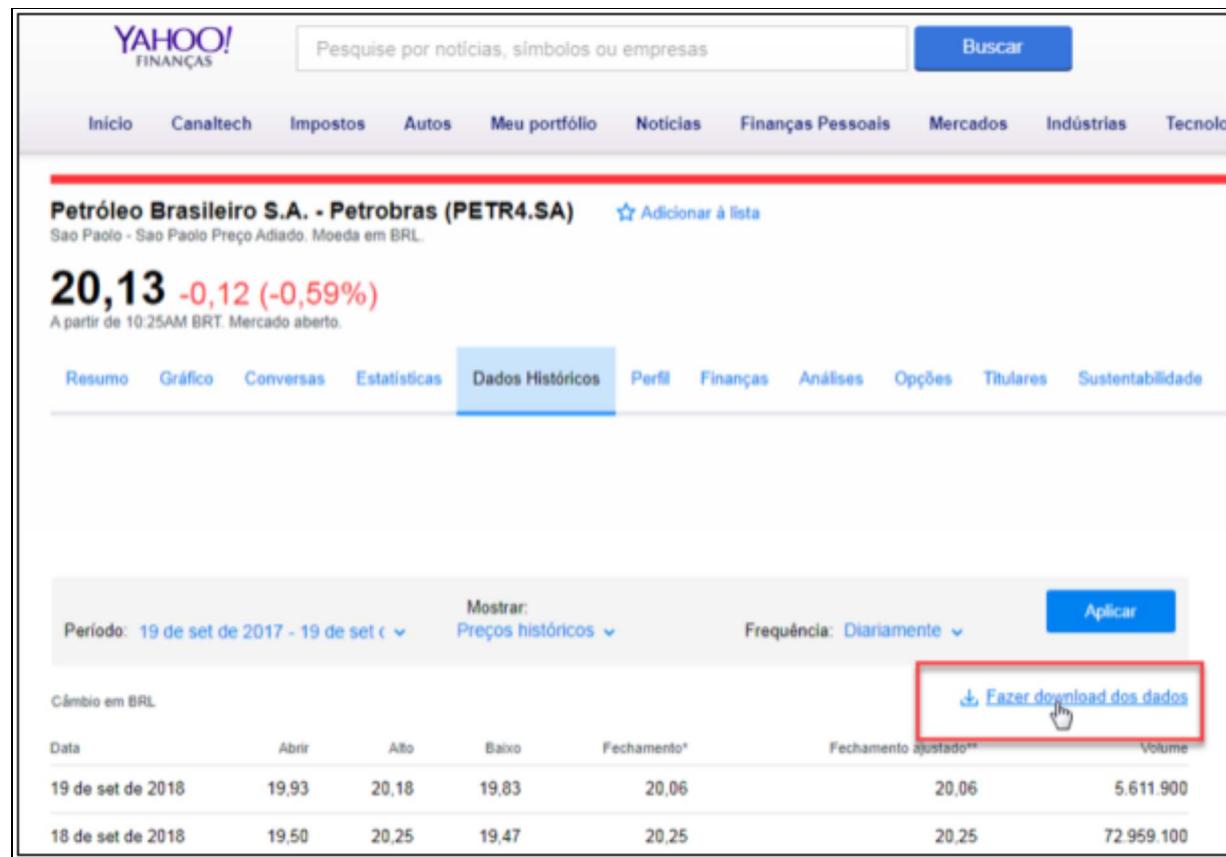
## Arquivos CSV

Um tipo de estrutura de arquivo bastante utilizado é o **CSV** (*comma-separated values*) ou arquivos de valores separados por vírgula. Nas áreas de análises de dados, mercado financeiro e estudo de séries temporais, o uso de arquivos CSV são praticamente um padrão.

Podemos encontrar outros caracteres separadores dos dados como o ; (**ponto e vírgula**), mas não necessariamente apenas vírgulas.

O Python nos permite trabalhar com esses tipos de arquivos a partir de uma biblioteca própria chamada **csv**. Portanto, devemos importá-la com o comando **import**.

Vamos primeiro baixar um exemplo de arquivo CSV, do site Yahoo finanças, contendo dados de cotações de preços das ações da Petrobras (<https://goo.gl/trw5cz>).



The screenshot shows the Yahoo Finance interface for the Petrobras stock (PETR4.SA). The main header includes the Yahoo Finance logo, a search bar, and a 'Buscar' button. Below the header, there's a navigation menu with links like Início, Canaltech, Impostos, Autos, Meu portfólio, Notícias, Finanças Pessoais, Mercados, Indústrias, and Tecnologia. The main content area displays the stock information for 'Petróleo Brasileiro S.A. - Petrobras (PETR4.SA)'. It shows the current price as 20,13, a change of -0,12 (-0,59%), and a note that it's based on BRT (Brasil) time. Below this, there's a summary table with columns for Data, Abrir (Open), Alto (High), Baixo (Low), Fechamento\* (Closing), Fechamento ajustado\*\* (Adjusted Closing), and Volume. At the bottom of the table, there's a link labeled 'Fazer download dos dados' (Download data) with a red box around it. Above this link, there are dropdown menus for 'Período' (Period) set to '19 de set de 2017 - 19 de set de 2018', 'Mostrar:' (Show) set to 'Preços históricos' (Historical prices), and 'Frequência:' (Frequency) set to 'Diariamente' (Daily). There's also an 'Aplicar' (Apply) button.

Data	Abrir	Alto	Baixo	Fechamento*	Fechamento ajustado**	Volume
19 de set de 2018	19,93	20,18	19,83	20,06	20,06	5.611.900
18 de set de 2018	19,50	20,25	19,47	20,25	20,25	72.959.100

Precisamos agora mover o arquivo ‘**PETR4.SA.csv**’ baixado para o diretório que estamos trabalhando com o Jupyter.

Iremos utilizar o pacote **csv** para trabalhar com manipulação dos arquivos. Existem diversas maneiras de abrir um arquivo csv para leitura. Podemos utilizar outros pacotes como, por exemplo, o **Pandas**<sup>[12]</sup>() para abrir arquivos csv.

Iremos utilizar o **open()** com a diretiva ‘r’ (read). Uma delas bastante utilizada e mostrada a seguir.

```
In [1]: # importar bibliotecas para trabalhar com CSV
import csv

In [2]: # Leitura de arquivo CSV
with open('PETR4.SA.csv', 'r') as arquivo:
    linhas = csv.reader(arquivo)

    for linha in linhas:
        print(linha)

['Date', 'Open', 'High', 'Low', 'Close', 'Adj Close', 'Volume']
['2018-08-20', '18.370001', '18.510000', '18.090000', '18.360001', '18.360001', '56315600']
['2018-08-21', '18.270000', '18.530001', '17.650000', '17.719999', '17.719999', '68735300']
['2018-08-22', '17.700001', '18.379999', '17.650000', '18.350000', '18.350000', '56623700']
['2018-08-23', '18.350000', '18.520000', '17.879999', '17.950001', '17.950001', '56284100']
['2018-08-24', '18.350000', '18.440001', '18.049999', '18.299999', '18.299999', '37893000']
['2018-08-27', '18.410000', '18.760000', '18.299999', '18.709999', '18.709999', '35428000']
['2018-08-28', '18.760000', '18.850000', '18.340000', '18.350000', '18.350000', '38079200']
['2018-08-29', '18.500000', '19.309999', '18.450001', '19.299999', '19.299999', '71434300']
['2018-08-30', '19.150000', '19.540001', '18.680000', '18.799999', '18.799999', '63981900']
['2018-08-31', '18.950001', '19.430000', '18.910000', '19.260000', '19.260000', '75767400']
['2018-09-03', '18.990000', '19.129999', '18.760000', '19.000000', '19.000000', '25401600]
```

Pronto, estamos com nossos dados carregados e prontos para serem analisados.

Vamos estudar agora como criar um arquivo .csv e adicionar dados.

```
In [1]: # importar bibliotecas para trabalhar com CSV
import csv

In [2]: # Criar e salvar dados em arquivo csv
with open('DADOS.csv','w') as arquivo:
    leitor = csv.writer(arquivo)
    leitor.writerow(['Col 1', 'Col 2', 'Col 3'])
    for i in range(100):
        # salvar colunas de dados
        leitor.writerow((i,2*i,3*i))
```

Tivemos o objetivo de criar um arquivo com o nome **DADOS.csv** e para isso foi preciso utilizar o parâmetro '**w**' (*write*).

Foram criadas três colunas (Col 1, Col 2, Col 3) nas quais foram adicionados números a partir do método **writerow()** (escrever na linha).

Veja abaixo como ficou a estrutura do nosso arquivo **csv** criado.

jupyter DADOS.csv 6 minutos atrás	
1	Col 1,Col 2,Col 3
2	
3	0,0,0
4	
5	1,2,3
6	
7	2,4,6
8	
9	3,6,9
10	
11	4,8,12
12	
13	5,10,15

Muitas vezes precisamos abrir os dados dos arquivos e salvá-los em uma **lista** (*list*) para facilitar nossas análises. Veja abaixo como fazer

isso:

```
In [1]: # importar bibliotecas para trabalhar com CSV
import csv

In [2]: with open('PETR4.SA.csv','r') as arquivo:
        leitor = csv.reader(arquivo)
        # transforma em lista
        dados = list(leitor)

In [3]: print(dados)

[['Date', 'Open', 'High', 'Low', 'Close', 'Adj Close', 'Volume'],
[001', '18.360001', '56315600'], ['2018-08-21', '18.270000', '18.53',
['2018-08-22', '17.700001', '18.379999', '17.650000', '18.350000',
0000', '17.879999', '17.950001', '17.950001', '56284100'], ['2018-
'18.299999', '37893000'], ['2018-08-27', '18.410000', '18.760000',
```

Fizemos essa conversão dos dados para um tipo lista, pois assim podemos utilizar os métodos prontos para trabalhar com essa estrutura de dados.

Vejamos ver como abrir arquivos .csv de outra maneira:

```
In [1]: aqruivo = open('PETR4.SA.csv','r')
dados = aqruivo.readlines()

In [2]: dados

Out[2]: ['Date,Open,High,Low,Close,Adj Close,Volume\n',
'2018-08-20,18.370001,18.510000,18.090000,18.360001,18.360001,56315600\n',
'2018-08-21,18.270000,18.530001,17.650000,17.719999,17.719999,68735300\n',
'2018-08-22,17.700001,18.379999,17.650000,18.350000,18.350000,56623700\n',
'2018-08-23,18.350000,18.520000,17.879999,17.950001,17.950001,56284100\n',
'2018-08-24,18.350000,18.440001,18.049999,18.299999,18.299999,37893000\n',
'2018-08-27,18.410000,18.760000,18.299999,18.709999,18.709999,35428000\n',
'2018-08-28,18.760000,18.850000,18.340000,18.350000,18.350000,38079200\n',
```

No capítulo a seguir, iremos estudar como obter gráficos com o **Matplotlib** a partir dos dados contidos em arquivos .csv.



## Trabalhando com diretórios

Como já sabemos criar, ler e alterar arquivos, vamos aprender agora como listar e manipular os mesmos dentro de diretórios. Por vezes, precisamos também verificar o tamanho de arquivos e data de criação.

```
In [1]: import os  
  
In [2]: # Mostra o diretório atual de trabalho  
os.getcwd()  
  
Out[2]: 'C:\\\\Users\\\\Computer\\\\Desktop\\\\E-Book Amazon\\\\Livro_Python\\\\CódigosPython'
```

Podemos listar os diretórios ou pastas de arquivos presentes dentro do diretório de trabalho da seguinte maneira:

```
In [3]: # Listar diretório  
os.listdir()  
  
Out[3]: ['inumh_checkpoints',  
         'ipynb',  
         'pynb',  
         ]
```

Lista de diretórios

Também podemos criar diretórios (pastas) facilmente a partir do comando ‘**mkdir**’. Esse comando é muito útil, veja como utilizá-lo abaixo:

```
In [4]: # Criar novo diretório  
os.mkdir('ESTUDO PYTHON')  
  
In [5]: os.listdir()  
  
Out[5]: ['.ipynb_checkpoints',  
          '... if else elif.ipynb',  
          'dados_fun_quadratica.txt',  
          'Dicionarios.ipynb',  
          'estudo diretórios.ipynb',  
          'Estudo notebook_1.ipynb',  
          'ESTUDO PYTHON',  
          'funcoes_lamda.ipynb',  
          'Funcoes.ipynb',
```

Algumas vezes precisamos trocar de diretórios para manipular arquivos. Com Python podemos fazer isso da seguinte maneira:

```
In [6]: # Diretório atual  
os.getcwd()  
  
Out[6]: 'C:\\Users\\Computer\\Desktop\\E-Book Amazon\\Livro_Python\\CódigosPython'  
  
In [7]: # Mudar para o diretório ESTUDO PYTHON  
os.chdir('ESTUDO PYTHON')  
  
In [8]: # Diretório atual  
os.getcwd()  
  
Out[8]: 'C:\\Users\\Computer\\Desktop\\E-Book Amazon\\Livro_Python\\CódigosPython\\ESTUDO PYTHON'
```

A diretiva ‘..’ dentro do método **chdir()** (*change directory*) nos permite retornar para o diretório imediatamente anterior ao atual. Essa troca de diretórios se dá a partir de hierarquias de diretórios mais externos para os mais internos e vice e versa.

```
In [8]: # Diretório atual  
os.getcwd()  
  
Out[8]: 'C:\\Users\\Computer\\Desktop\\E-Book Amazon\\Livro_Python\\CódigosPython\\ESTUDO PYTHON'  
  
In [9]: # Retornar para diretório anterior  
os.chdir('..')  
  
In [10]: # Diretório atual  
os.getcwd()  
  
Out[10]: 'C:\\Users\\Computer\\Desktop\\E-Book Amazon\\Livro_Python\\CódigosPython'
```

Podemos especificar diretamente o caminho do diretório no qual queremos trabalhar. Algo importante é ficar atento ao sistema operacional, pois a estrutura de referênciação de diretórios pode mudar.

Neste caso em específico, os exemplos deste livro estão sendo mostrados em um Sistema Operacional Windows, consequentemente, temos a contrabarra '\` como caractere usado para separar diretórios. No sistema operacional LINUX, por exemplo, temos a barra '/'.

Vejamos agora como podemos renomear diretórios preexistentes. Iremos renomear o diretório **PYHTON\_ESTUDO** para **ESTUDO\_PYTHON**.

```
In [11]: # Renomear diretório  
os.rename('ESTUDO PYTHON', 'PYTHON ESTUDO')
```

```
In [12]: os.listdir()
```

```
Out[12]: ['.ipynb_checkpoints',  
          '... . . . . : elif.ipynb',  
          'dados_fun_quadratica.txt',  
          'Dicionarios.ipynb',  
          'estudo diretórios.ipynb',  
          'Estudo_notebook_1.ipynb',  
          'funcoes lambda.ipynb',  
          'Funcoes.ipynb',  
          'Listas.ipynb',  
          'Manipulação de Arquivos.ipynb',  
          'meuprimeironotebook.ipynb',  
          'PYTHON ESTUDO',  
          'string_estudo.ipynb',
```

A remoção/exclusão de um diretório e arquivos é feita a partir do método `.remove('nome_arquivo')`. Devemos ter a permissão de administrador do computador para fazer esse tipo de ação.

```
In [13]: os.remove('PYTHON ESTUDO')
```

```
-----  
PermissionError                                                 Traceback (most recent call last)  
<ipython-input-13-cd5f63a39b61> in <module>()  
----> 1 os.remove('PYTHON ESTUDO')  
  
PermissionError: [WinError 5] Acesso negado: 'PYTHON ESTUDO'
```

Veja um exemplo de remoção de arquivo:

```
In [2]: # Verificar se arquivo existe  
os.path.exists('dados_fun_quadratica.txt')
```

```
Out[2]: True
```

```
In [3]: # Remover arquivo txt  
os.remove('dados_fun_quadratica.txt')
```

```
In [4]: # Verificar se arquivo existe  
os.path.exists('dados_fun_quadratica.txt')
```

```
Out[4]: False
```

## Data e hora

Saber a data e hora da criação e/ou modificação de arquivos e diretórios é de extrema importância para o devido gerenciamento de nossas análises. Por isso, Python conta com uma biblioteca especializada para trabalhar com **tempo**.

O módulo **time** nos permite manipular informações do tempo em nossas atividades.

```
In [1]: import time

In [2]: # Mostra o tempo agora em segundos
atual = time.time()

In [3]: print(atual)
1537284444.8542824

In [4]: # Traduz a notação temporal de máquina
time.ctime(atual)

Out[4]: 'Tue Sep 18 12:27:24 2018'
```

A função **time.time()** retorna a hora atual em segundos usando como referência o horário de *Greenwich* ou UTC (Tempo Universal Coordenado). A função **ctime()** traduz para nós o tempo em segundos para uma notação mais usual.

Se precisamos trabalhar com o tempo especificado no fuso horário devemos utilizar o **time.localtime()** que retorna uma tupla, diferentemente do **time.time()** que retorna um número apenas.

A função **gmtime** pode ser utilizada para converter o retorno da tupla em nove elementos que são:

ano (**tm\_yar**),

mês (`tm_mon`),  
dia (`tm_day`),  
hora (`tm_hour`),  
minutos (`tm_min`),  
segundos (`tm_sec`),  
dia da semana entre 0 a 6 (segunda é 0) (`tm_wday`),  
dia do ano (1 a 366) (`tm_yday`),  
horário de verão indicado pelo número 1 (`tm_isdst`).

```
In [5]: # Tempo Local  
local = time.localtime()  
  
In [6]: print(local)  
time.struct_time(tm_year=2018, tm_mon=9, tm_mday=18, tm_hour=12, tm_min=28, tm_sec=34, tm_wday=1, tm_yday=261, tm_isdst=0)
```

Podemos obter apenas as informações necessárias a partir de chamadas da variável ‘local’. Veja um exemplo abaixo:

```
In [7]: # Retornar o ano atual  
print('Ano: %d' % local.tm_year)  
Ano: 2018  
  
In [8]: # Retornar hora  
print('Hora: %d' % local.tm_hour)  
Hora: 12  
  
In [9]: # Retornar min  
print('Minutos: %d' % local.tm_min)  
Minutos: 45  
  
In [10]: # Retornar segundos  
print('Segundos: %d' % local.tm_sec)  
Segundos: 29
```

São vários detalhes né verdade? Não se preocupe que você não precisa decorar todos eles. Entretanto, é importante que você conheça a existência desses comandos e métodos para um dia quando precisar saber que é fácil a sua utilização.

Muitos novatos no mundo da programação possuem a ideia errônea de que é preciso decorar as funções, métodos, variáveis etc. Todavia, nós como programadores de computadores precisamos conhecer que existem facilidades e precisamos saber aplicá-las apenas.

Não precisamos ter nada decorado, pois o Google está aí a nossa disposição a todo o momento.

Se você acha que já esqueceu tudo o que foi visto no início do livro, não se preocupe porque isso é normal.

Nossa mente é fantástica pensamos que esquecemos, mas tudo está lá bem guardadinho em um lugar escondido da memória. Quando a gente se deparar com algum problema um dia, as lembranças surgem para nós de maneira mágica. E mais uma vez o Google é o maior aliado, amigo e companheiro da memória.

Vamos ao próximo capítulo!



# **Capítulo 6**

# Geração de gráficos e diagramas com Python

Como sabemos, nós humanos possuímos diversas limitações. Por isso, utilizamos o computador para suplantar nossas fraquezas e potencializar nossas capacidades cognitivas e de memória.

Vivemos atualmente na era do **big data** que basicamente pode ser definida como uma geração de dados em grande **volume, velocidade e variedade** sem precedentes.

Toda essa massa de dados precisa ganhar forma e ser interpretada. Dados precisam virar informações. Dentro dessa conjuntura, vem surgindo novos profissionais dispostos a solucionar as principais questões dentro dessa problemática.

Analistas, engenheiros e cientistas de dados são atualmente as profissões mais requisitadas e bem remuneradas. Basicamente, todos eles estão em busca de transformar dados em soluções para problemas de caráter competitivo dentro de algum campo de negócio.

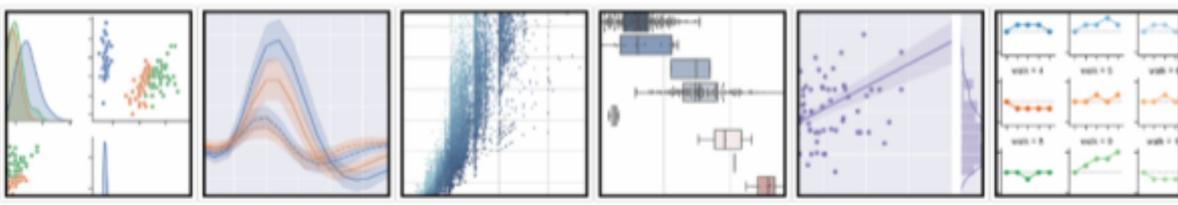
Praticamente, toda e qualquer boa análise passa pela criação e apresentação dos dados em diagramas gráficos. Como sabemos bem, uma figura fala mais do que mil palavras. Assim, um gráfico pode expressar bilhões de dados de uma maneira simplificada e intuitiva.

Quando o assunto é gerar e manipular gráficos, o Python apresenta uma enorme quantidade de pacotes e bibliotecas para esse fim.

As principais bibliotecas para gerar e manipular gráficos e figuras são:

- **Seaborn** (<https://seaborn.pydata.org/>) - bastante utilizada para análises de visualização estatística.

## seaborn: statistical data visualization



- **Matplotlib** (<https://matplotlib.org/index.html>) - uma das mais utilizadas para gerar gráficos 2D e 3D. Várias bibliotecas utilizam a Matplotlib como base para criação de novas funcionalidades.



[home](#) | [examples](#) | [tutorials](#) | [API](#) | [docs](#) »

Matplotlib is a Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms. Matplotlib can be used in Python scripts, the Python and IPython shells, the Jupyter notebook, web application servers, and four graphical user interface toolkits.



- **Pandas** (<https://pandas.pydata.org/>) - biblioteca com diversas funções preparadas para trabalhar com análise e visualização de dados.

**pandas**

$$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$$

[home](#) // [about](#) // [get pandas](#) // [documentation](#) // [community](#) // [talks](#) // [donate](#)

## Python Data Analysis Library

*pandas* is an open source, BSD-licensed library providing high-performance, easy-to-use data structures and data analysis tools for the [Python](#) programming language.

*pandas* is a [NumFOCUS](#) sponsored project. This will help ensure the success of development of *pandas* as a world-class open-source project, and makes it possible to [donate](#) to the project.

**VERSIONS**

<b>Release</b> 0.23.4 - August 2018 <a href="#">download</a> // <a href="#">docs</a> // <a href="#">pdf</a>
<b>Development</b> 0.24.0 - 2018 <a href="#">github</a> // <a href="#">docs</a>

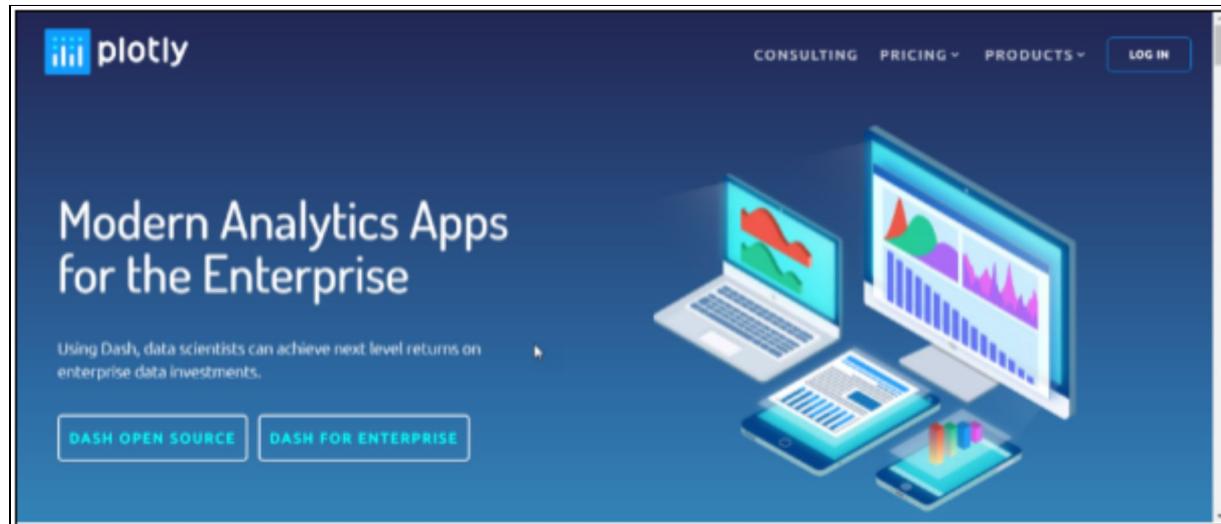
- **Altair** (<https://altair-viz.github.io/>) - possui um conjunto de bibliotecas para trabalhar de modo declarativo os dados estatísticos. Muito boa para trabalhar com mapas e diagramas.

## Altair: Declarative Visualization in Python

Altair is a declarative statistical visualization library for Python, based on [Vega](#) and [Vega-Lite](#), and the source is available on [GitHub](#).

- **Plotly** – (<https://plot.ly/>) - gera gráficos interativos de maneira fácil, amigável e intuitiva. É muito utilizada no mundo

corporativo.



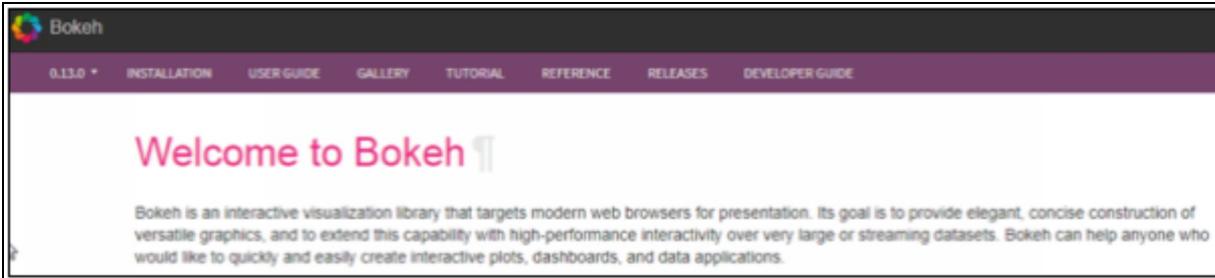
- **ggplot** - (<http://ggplot.yhatq.com/>) - este é um gerador de gráficos baseado no gerador ggplot2 da linguagem R. Com pouquíssimas linhas de códigos podemos gerar gráficos complexos e muito bonitos.

A screenshot of the ggplot from yhat website. The header includes links for About, Installation, How It Works, Docs, and Gallery. The main content area starts with the heading "ggplot from yhat". Below it is a brief description: "ggplot is a plotting system for Python based on R's ggplot2 and the Grammar of Graphics. It is built for making professional looking, plots quickly with minimal code." A large red box contains the heading "ggplot is easy to learn" and a code snippet:

```
from ggplot import *

ggplot(aes(x='date', y='beef'), data=meat) +\
    geom_line() +\
    stat_smooth(colour='blue', span=0.2)
```

- Bokeh –(<https://bokeh.pydata.org/en/latest/>) esta é uma excelente biblioteca para construir gráficos com muita qualidade e interatividade.



Após essa breve apresentação das principais bibliotecas gráficas nós devemos escolher uma para começar a trabalhar. Portanto, neste capítulo, iremos nos deter ao uso do **Matplotlib**.

## Gerando Gráficos com o Matplotlib

A biblioteca **Matplotlib** é bastante completa (ver documentação para mais detalhes - <https://matplotlib.org/>). Dentro dela iremos utilizar o pacote **pyplot**.

Para podermos utilizar os recursos da biblioteca Matplotlib precisamos fazer o **import** como descrito abaixo. Também iremos criar duas funções **y1** (linear) e **y2** (quadrática). Criaremos um conjunto de 100 valores para **x**, que variam de zero a dois, confira:

```
In [1]: import matplotlib.pyplot as plt  
import numpy as np  
  
In [2]: # criar 100 dados igualmente intervalados  
# entre zero e dois:  
x = np.linspace(0,2,100)  
  
# Função Linear  
y1 = x**2  
  
# Função quadrática  
y2 = x**2
```

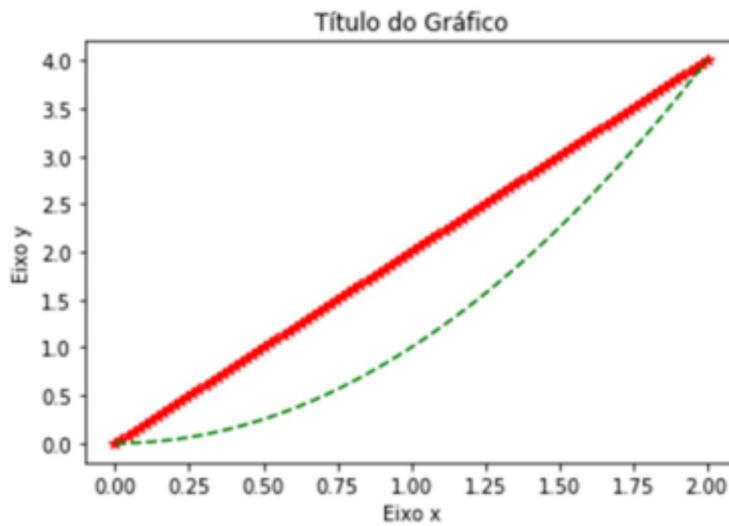
Com isso temos a variável **x** com 100 elementos e as variáveis **y1** e **y2** também com 100 elementos transformados segundo sua lei funcional: linear e quadrática, respectivamente.

Vamos agora plotar as duas funções **y1** e **y2**:

```
In [3]: # Plotar dados  
plt.plot(x,y1,'*r')  
plt.plot(x,y2,'g--')  
  
# Legenda dos eixos  
plt.xlabel('Eixo x')  
plt.ylabel('Eixo y')  
  
# Título do gráfico  
plt.title('Título do Gráfico')
```

O resultado será o seguinte:

```
Out[3]: Text(0.5,1,'Título do Gráfico')
```



Podemos customizar nossos gráficos de várias maneiras: modificando a cor (**r** - red, **b**- blue, **y** - yellow, **g** - green, dentre outros), o tipo dos pontos (\*, --, \*-, o , dentre vários outros).

As possibilidades de formatações gráficas são praticamente infinitas, depende de nossa criatividade. Consulte a documentação para mais formatos.

Vejamos agora uma funcionalidade bem útil: como obter vários gráficos um ao lado do outro. Para isso, vamos gerar mais uma

função chamada de **y3** (função cúbica).

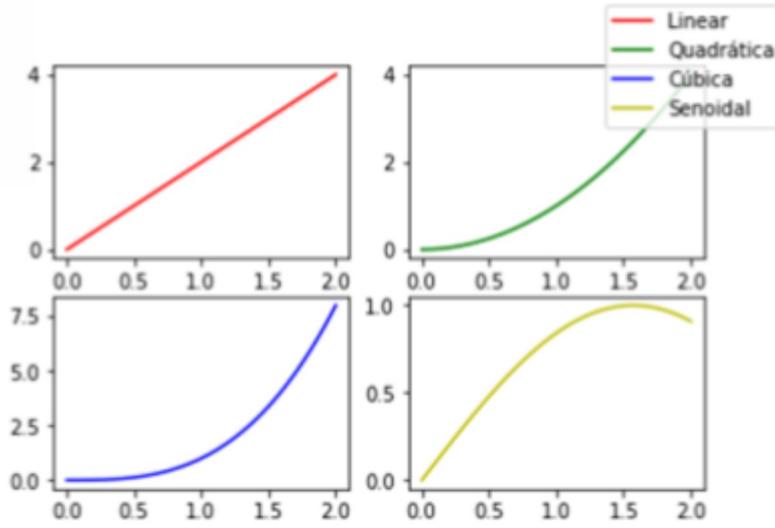
```
In [1]: import matplotlib.pyplot as plt  
  
import numpy as np  
  
In [2]: # criar 100 dados igualmente intervalados  
# entre zero e dois:  
x = np.linspace(0,2,100)  
  
# Função Linear  
y1 = x**2  
# Função quadrática  
y2 = x**2  
# Função cúbica  
y3 = x**3  
# Função seno  
y4 = np.sin(x)
```

Iremos utilizar a função **subplot()**. Veja abaixo:

```
In [3]: # plots Lado a Lado:  
# Gerar janela 2 linhas com 2 colunas  
fig, janela = plt.subplots(2, 2)  
  
# Adicionar a cada janela um gráfico  
janela[0, 0].plot(x, y1,'r')  
janela[0, 1].plot(x, y2,'g')  
janela[1, 0].plot(x, y3,'b')  
janela[1, 1].plot(x, y4,'y')  
  
fig.legend(['Linear','Quadrática','Cúbica','Senoidal'])
```

Confira o resultado da saída gráfica:

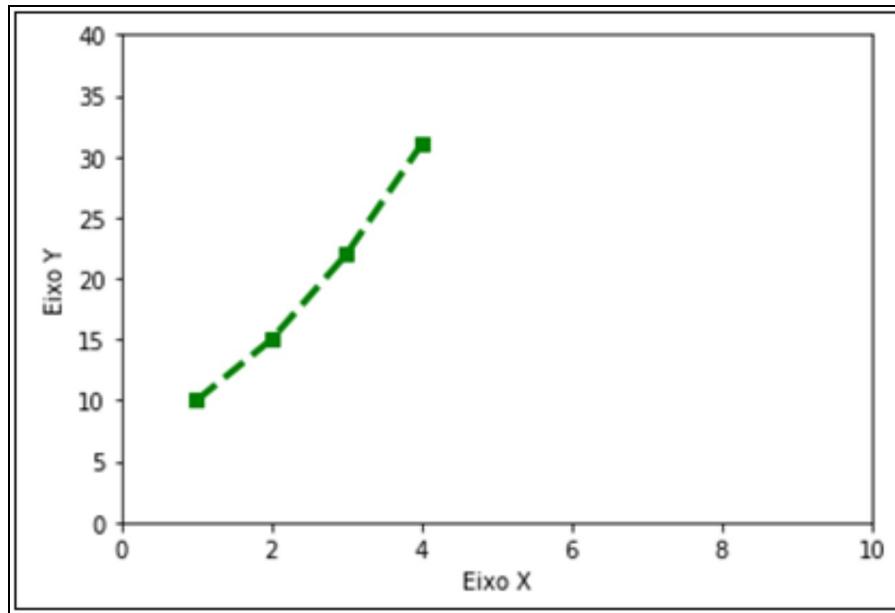
```
Out[3]: <matplotlib.legend.Legend at 0x5963748>
```



Veja um exemplo de um gráfico mais customizado:

```
In [1]: import matplotlib.pyplot as plt  
import numpy as np
```

```
In [2]: # Desenhar gráfico  
plt.plot([1,2,3,4], [10, 15, 22, 31],  
         linestyle='--', color='green', marker='s', linewidth=3.0)  
  
# Ajustar eixos do gráfico  
plt.axis([0,10, 0,40]) # [x1,x2, y1,y2]  
plt.xlabel('Eixo X')  
plt.ylabel('Eixo Y')  
plt.show()
```



### Gráfico de Barras

O gráfico de barras é comumente utilizado para fazer demonstrativos estatísticos no meio corporativo.

Podemos obter um gráfico de barras com a função **bar()**, na qual definimos a posição das barras no **eixo X** (localização das variáveis) e sua altura no **eixo Y** (valores das variáveis). De modo adicional, podemos configurar outras características, como a espessura das barras, cor etc.

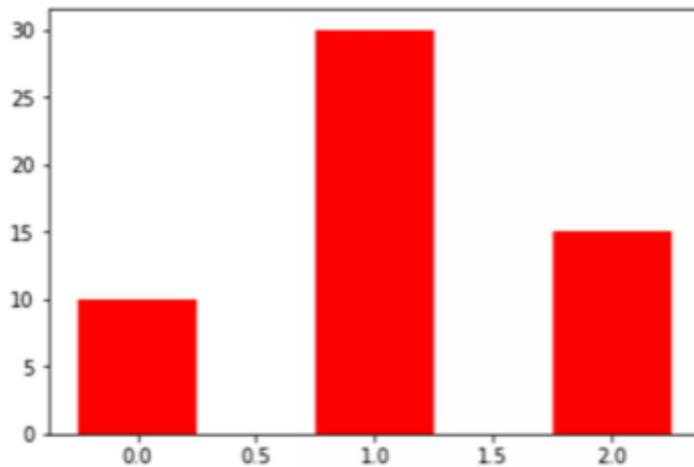
O eixo X representa um **range** com os nomes das variáveis. Por outro lado, o eixo Y apresenta os valores das propriedades do eixo X.

Vamos estudar um exemplo bem simples, onde estamos guardando os valores das variáveis em listas.

```
In [2]: # Variáveis para o Bar Chart
eixo_y = [10,30,15]
eixo_x = range(len(eixo_y))

largura = 0.5
cor_barra = 'red'

plt.bar(eixo_x, eixo_y, width=largura, color=cor_barra)
plt.show()
```



Este é apenas um exemplo básico de gráfico de barras. Podemos obter gráficos muito complexos, com superposições de barras, barras laterais, dentre outras. Ver documentação para mais detalhes.

## Histogramas

Os histogramas são gráficos descritivos largamente utilizados. Eles geralmente são utilizados para apresentar a distribuição da frequência de ocorrência de uma série de variáveis.

A função Python nesse caso é a **hist()**. Nessa função passaremos como parâmetros propriamente ditos a série de valores e os **bins** (são os tamanhos dos intervalos em que serão avaliadas as frequências de observação).

Vale muito a pena para o leitor que não está familiarizado com histogramas fazer uma pesquisa rápida no Google para entender melhor tudo o que foi dito até aqui. Esse tipo de gráfico é essencial para todo e qualquer analista de dados.

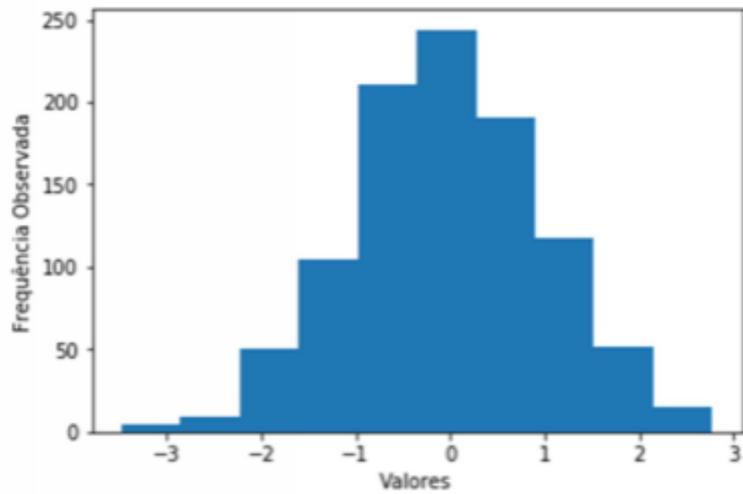
Vamos para um exemplo prático. Siga o seguinte raciocínio: primeiro precisamos gerar nossos dados com o **NumPy**, depois plotar o histograma com os dados.

```
In [1]: import matplotlib.pyplot as plt
import numpy as np

In [2]: # Gerar números aleatórios com distribuição Normal
num_norm = np.random.randn(1000,1) # 1000 Linhas e 1 Coluna
```

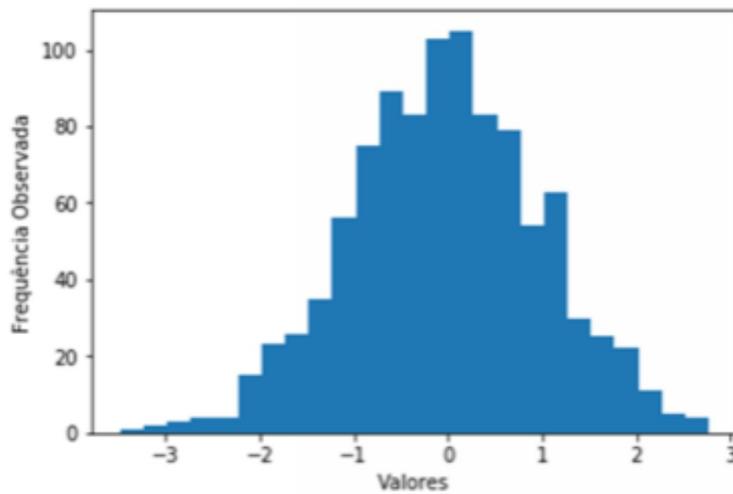
Para plotar um histograma, que é basicamente um gráfico de barras onde observamos a frequência de ocorrência de um determinado valor em intervalos igualmente especificados (**bins**), basta utilizar a função **hist()**.

```
In [3]: # bins = representa a quantidade de caixas com intervalos de  
# observação de frequência  
plt.hist(num_norm, bins=10)  
plt.xlabel('Valores')  
plt.ylabel('Frequência Observada')  
plt.show()
```



Vamos aumentar o número de **bins** de **10** para **25** e ver o que acontece com nosso histograma.

```
In [4]: plt.hist(num_norm, bins=25)
plt.xlabel('Valores')
plt.ylabel('Frequência Observada')
plt.show()
```



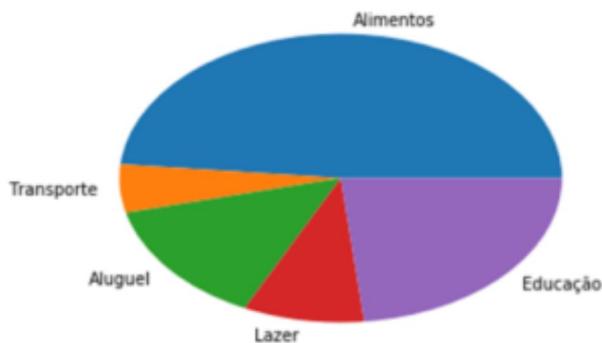
### Gráficos de Pizza

O gráfico de pizza, também chamado de gráfico de setores, apresenta um diagrama circular com subdivisões setoriais de sobremodo intuitivas. Essas subdivisões representam as categorias de análises que são proporcionais em tamanhos relativos a alguma característica de estudo.

Com a função **pie()** podemos obter gráficos de setores de uma maneira muito simplificada:

```
In [2]: # Gastos financeiros  
dados = [520, 58, 150, 95, 250]
```

```
In [3]: # Cria gráfico de pizza  
plt.pie(dados, labels=['Alimentos', 'Transporte', 'Aluguel', 'Lazer', 'Educação'])  
plt.show()
```



Este foi um capítulo curto, mas que envolve muito assunto a ser estudado. É sempre bom lembrar que depois de análises e estudos iremos tentar sempre representar nossos resultados em gráficos. É aí que a boa aparência e a estética entram em jogo.

Existem também questões de formatação linguísticas (*storytelling*) e de cores (psicologia das cores). Portanto, existe um verdadeiro mundo de oportunidade para aqueles que gostam e pretendem se especializar nessa área.

Faça novos estudos e tente avançar mais nesse campo tão importante e facilitador das apresentações e interpretações de nossas análises que são os gráficos.



# **Capítulo 7**

## Funções especiais

O mundo do cientista e analista de dados é repleto de atividades de transformações de dados. Muitas dessas atividades estão relacionadas à limpeza e organização.

Com frequência precisamos executar ações repetitivas, como remover ou adicionar dados, que necessitam da aplicação de uma função específica. Pensando nisso, foram desenvolvidas algumas funções visando agilizar *loops* de processamentos no Python.

Essas funções estão otimizadas e permitem que os analistas executem suas atividades com grande desempenho de velocidade e praticidade.

Iremos estudar com vários exemplos cada uma das seguintes funções, a saber: **Map**, **Reduce**, **Filter**, **Zip** e **Enumerate**.

Algumas dessas funções são *built-in*, mais especificamente, as seguintes:

**map**(função, sequência)

**reduce**(função, sequência)

**filter**(função, sequência)

**lambda**

## Função Map

A programação funcional é orientada a expressões que precisam ser interpretadas diretamente com base em uma lógica. A função **map()** é um exemplo de função orientada a expressão. Veja abaixo a estrutura principal da função Map:

### **map(Fun, Seq)**

O uso de loop for pode tornar o código lento para grandes volumes de dados. Sempre que possível vale a pena evitar o uso do *loop for*. Para isso, temos a função **map()** que aplica uma ação (função) a todos os elementos de uma lista sem a necessidade de utilizar *loop for*. Assim, a função **map()** retorna uma nova lista com os elementos alterados pela função escolhida (**Fun**).

Vamos a um exemplo. Primeiro precisamos criar funções para serem aplicadas as listas a partir da função **Map**.

Iremos criar uma função preço de uma corrida de táxi, na qual temos um valor fixo, que representa o valor cobrado logo quando entramos no táxi (R\$ 2,50), mais um valor variável referente à quilometragem percorrida, multiplicado por uma taxa constante igual a R\$ 1,50.

A função é a seguinte:  $P(x) = 2,5 + 1.5*x$ , onde  $x$  representa a quilometragem percorrida.

```
In [1]: # Função para cálculo do preço da corrida
def precoCorridaTaxi(x):
    return(2.5+1.5*x)

In [2]: # Listas de distâncias percorridas
distancias = [120, 100, 50, 70, 88]

In [5]: # Preço das corridas
preco = list(map(precoCorridaTaxi,distancias))

In [6]: preco
Out[6]: [182.5, 152.5, 77.5, 107.5, 134.5]
```

Desse modo, a função `built-in map` aplicou a função ‘`precoCorridaTaxi`’ iterativamente a cada um dos elementos da lista ‘`distancias`’, retornando um iterator (`<map>`). Contudo, como queremos uma lista como retorno, precisamos utilizar a função ‘`list`’. Como já dito antes, esse tipo de ação agiliza tanto o processo de desenvolvimento dos algoritmos como o processamento computacional.

Podemos tornar ainda mais eficiente nosso código a partir das **expressões lambda**. Veja o exemplo abaixo.

```
In [5]: # Uso de expressões
preco = list(map(lambda x: 2.5+1.5*x,distancias))

In [6]: preco
Out[6]: [182.5, 152.5, 77.5, 107.5, 134.5]
```

Quando o conjunto de dados é pequeno, claro que a diferença de tempo computacional entre utilizar funções lambda e função com a diretiva ‘`def`’ é imperceptível. Por outro lado, quando estamos

trabalhando com grandes volumes de dados (terabytes, por exemplo) a percepção de ganho temporal se torna evidente.

Vamos estudar um exemplo que apresenta o enorme poder que temos nas mãos a partir da função **map()**. Para isso, iremos aplicar operações a três listas diferentes ao mesmo tempo, veja abaixo:

```
In [7]: # Listas
L1 = [10,20,30,40,50]
L2 = [1,2,3,4,5]
L3 = [11,22,33,44,55]

In [8]: soma_listas = list(map(lambda a,b,c: a+b+c, L1,L2,L3))

In [9]: soma_listas

Out[9]: [22, 44, 66, 88, 110]
```

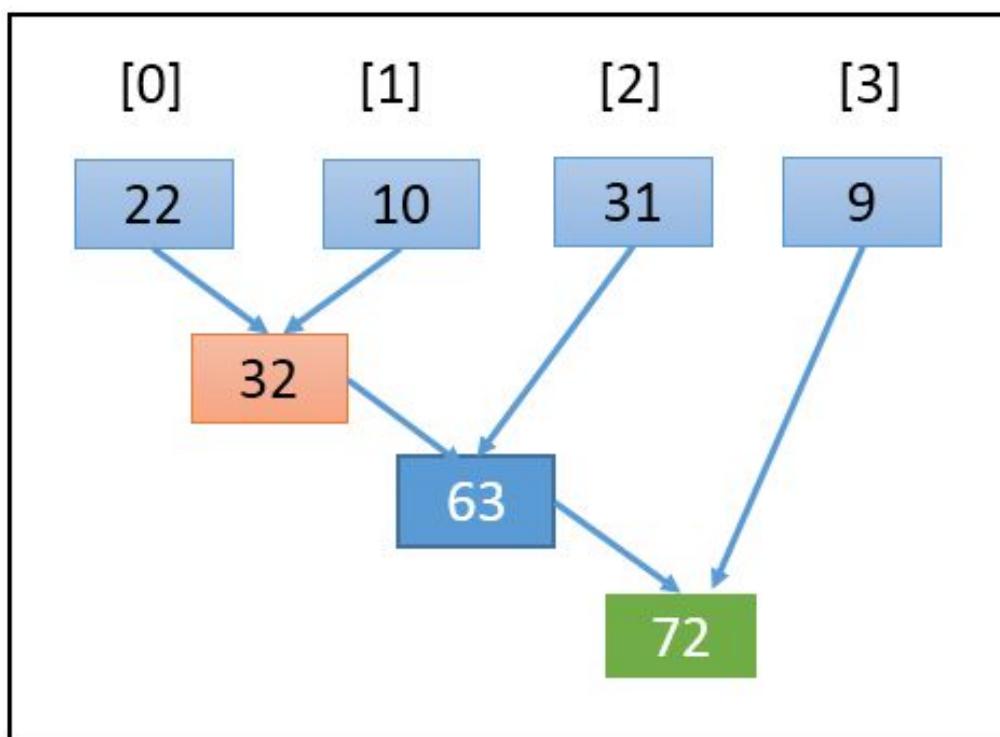
No caso acima fomos capazes de aplicar ao mesmo tempo uma função de três variáveis, em que cada variável pode ser representada como um elemento de três listas.

## Função Reduce

Assim como a função **map()** a função **reduce()** recebe dois argumentos apenas. Veja abaixo o caso geral de funcionamento:

`reduce(fun,seq)`

O primeiro argumento é uma função e o segundo é uma sequência de elementos (uma lista, por exemplo). A função **reduce**, diferentemente da função **map**, aplica a cada elemento da sequência a função passada como parâmetro até que reste apenas um elemento.



A função **reduce()** apresenta uma particularidade. Embora seja uma função *built-in* (que em geral não precisa ser importada), a função

`reduce()` precisa ser importada de ‘`functools`’. Vamos a um exemplo prático para esclarecer melhor tudo isso.

```
In [1]: from functools import reduce

In [2]: lista = [1, 2, -3, 4, 5, -9]
def soma(a, b):
    return a + b

In [3]: reduce(soma, lista)

Out[3]: 0
```

Vamos entender agora como chegamos ao resultado ZERO mostrado acima. As operações com a função `reduce()` são feitas em pares. Assim, `reduce()` aplica a função `soma` aos pares dos elementos da lista.

Em outras palavras, temos:

$$(((1 + 2) + (-3)) + 4) + 5) + (-9)) = 0$$

## Função Filter

Assim como as funções **map()** e **reduce()**, a função **filter()** também recebe dois argumentos, uma função predefinida e uma sequência de dados.

**filter(Fun, seq)**

A função **filter()** permite que possamos fazer filtros em elementos de uma sequência de valores de maneira otimizada. Uma vez encontrado o valor baseado no filtro da função (Fun), iremos ter um retorno **true** da função **filter()**.

Vejamos o exemplo abaixo:

```
In [5]: # Função para verificar se nº é par
def isPar(num):
    if num % 2 == 0:
        return True
    else:
        return False

In [6]: numeros = [2, 54, 87, 4, 90, 43, 26, 50]

In [8]: # Filtra os nº pares da Lista 'numeros'
list(filter(isPar,numeros))

Out[8]: [2, 54, 4, 90, 26, 50]
```

Poderíamos ter utilizado a expressão lambda para obter o mesmo resultado. Veja o exemplo abaixo:

```
In [9]: list(filter(lambda x: x%2==0, numeros))
```

```
Out[9]: [2, 54, 4, 90, 26, 50]
```

## Função Zip

A função `zip()` foi construída visando agregar valores de duas sequências retornando uma tupla. Esse tipo de operação é muito útil quando estamos querendo fazer *loops* em mais de uma lista ao mesmo tempo.

Veja abaixo a forma geral de uso:

`zip(seq, seq)`

seq - sequência. Portanto, essa função nos retorna o agrupamento de duas sequências em uma tupla. Vale ressaltar que podemos utilizar `zip()` em duas sequências com números diferentes de elementos. No entanto o número de tuplas será igual ao número da menor sequência de valores.

Veja esse exemplo:

`zip([1,2,3,4,5], [6,7,8]) = (1, 2) (2, 7) (3, 8)`

```
In [1]: # Função ZIP
In [2]: # Listas
L1 = [1,2,3,4,5,6]
L2 = [7,8,9,1,2,3]
In [3]: # Unir as listas
zip(L1,L2)
Out[3]: <zip at 0x59ec2c8>
In [4]: list(zip(L1,L2))
Out[4]: [(1, 7), (2, 8), (3, 9), (4, 1), (5, 2), (6, 3)]
```

O retorno da **zip()** (*zip at 0x59ec2c8*) é um iterator, por isso precisamos usar a função **list()** para obter uma lista. Vamos fazer um exemplo em que os tamanhos das listas são diferentes.

```
In [5]: # Listas tamanhos diferentes  
L3 = [1,2,3,4,5]  
L4 = [22,11]
```

```
In [6]: list(zip(L3,L4))  
Out[6]: [(1, 22), (2, 11)]
```

Podemos utilizar dicionários dentro da função **zip()**. Vale lembrar que os argumentos desta função devem ser sequências. Como já estudamos, as sequências podem ser de vários tipos: listas, dicionários, tuplas.

```
In [7]: # Dicionários  
d1 = {'a':1, 'b':2, 'c':3, 'd':4}  
d2 = {'aa':11,'bb':22,'cc':33,'dd':44}
```

```
In [8]: # A função zip() faz a união pelas chaves  
list(zip(d1,d2))  
Out[8]: [('a', 'aa'), ('b', 'bb'), ('c', 'cc'), ('d', 'dd')]
```

Se quisermos unir os dicionários pelos valores, devemos fazer o seguinte:

```
In [9]: # Unindo valores
list(zip(d1.values(),d2.values()))

Out[9]: [(1, 11), (2, 22), (3, 33), (4, 44)]
```

```
In [10]: # Unindo chaves com valores
list(zip(d1,d2.values()))

Out[10]: [('a', 11), ('b', 22), ('c', 33), ('d', 44)]
```

A função `zip()` é muito utilizada em processos de permutação de valores entre listas e dicionários. Essa é uma atividade rotineira de cientistas de dados.

Veja abaixo um exemplo de função que faz a troca de valores entre dois dicionários.

```
In [7]: # Dicionários
d1 = {'a':1, 'b':2, 'c':3, 'd':4}
d2 = {'aa':11,'bb':22,'cc':33,'dd':44}

In [11]: # Troca de valores entre dois dicionários
def trocaValoresDicionarios(da,db):
    dicAux = {}
    for dakey, dbkey in zip(da,db.values()):
        dicAux[dakey] = dbkey

    return dicAux

In [12]: trocaValoresDicionarios(d1,d2)

Out[12]: {'a': 11, 'b': 22, 'c': 33, 'd': 44}
```

## Função Enumerate

A função **enumerate()** fornece os índices de cada valor em uma sequência. Assim **enumerate()** retorna uma tupla com o formato: **tupla(indice, valor)**.

A função **enumerate()** recebe apenas um parâmetro que é exatamente a sequência que queremos retornar o par índice valor. Ver caso geral abaixo:

`enumerate(seq)`

Quando uma sequência tiver valores não numéricos, é bem mais fácil fazer a manipulação a partir dos índices da sequência. Veja o exemplo abaixo:

```
In [15]: # Cria uma sequência
          seq = ['a', 'b', 'c', 'd']

In [16]: enumerate(seq)
Out[16]: <enumerate at 0x59eaa20>

In [17]: list(enumerate(seq))
Out[17]: [(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd')]
```

A partir dos índices podemos fazer operações lógicas e matemáticas com mais controle uma vez que muitas vezes não sabemos quais elementos estão na sequência.

Para terminar o capítulo, vamos fazer um exemplo para encontrar os valores da sequência onde os índices são menores do que determinado valor k.

```
In [1]: # Função para encontrar elementos de sequências com
# índices menores do que k
def listaElementosIndiceMenoresQue(seq,k):
    for indice, valor in enumerate(seq):
        if indice > k:
            break
        else:
            print(valor)
```

```
In [2]: # Cria uma sequência
seq = ['casa', 'mesa', 'elevador', 'porta']
```

```
In [3]: listaElementosIndiceMenoresQue(seq,1)

casa
mesa
```

Muito legal tudo isso né verdade. Ótimo, estamos avançando bem.

No próximo capítulo iremos nos preparar para os inconvenientes que por ventura venham acontecer durante a execução dos nossos códigos.

Vamos estudar como gerenciar mensagens de aviso e erros. Vamos nessa.



# **Capítulo 8**

## Mensagens de erros, warnings

Na vida temos sempre que nos preparar para o pior, pois mais cedo ou mais tarde certamente iremos nos deparar com situações caóticas. Não devemos tentar fugir dos problemas a todo o momento evitando enfrentar situações que fazem parte da vida. Temos sim que saber o que fazer quando um problema acontecer, mesmo que seja através de avisos e alarmes.

O interpretador Python informa erros de compilação e sintaxe por meio de mensagens indicando a linha do arquivo onde se encontra o possível problema.

Nem sempre o que o interpretador nos alerta como errado a real causa do problema. Por isso devemos ter bastante cuidado com essas orientações de pontos de erros.

Não precisamos nos desesperar quando uma mensagem de erro aparecer. É comum quando estamos iniciando em programação ficarmos congelados ao nos depararmos com uma mensagem de erro.

Os erros são degraus que precisamos subir para o aprendizado, portanto, estarão presentes durante toda a nossa jornada, uma vez que o aprendizado nunca termina.

Muitas vezes o interpretador Python acusa um erro porque houve uma interrupção no andamento da leitura dos códigos devido algum problema de sintaxe, por exemplo. Todavia, mesmo quando uma expressão estiver sintaticamente correta poderemos receber mensagens de ‘erro’ que nesse caso são chamados de **Exceções**.

Assim os principais erros que podemos nos deparar com a linguagem Python são os seguintes: sintaxe, indentação, key (acesso a chaves), nome, valor, tipo e index. Vamos agora compreender melhor cada um deles.

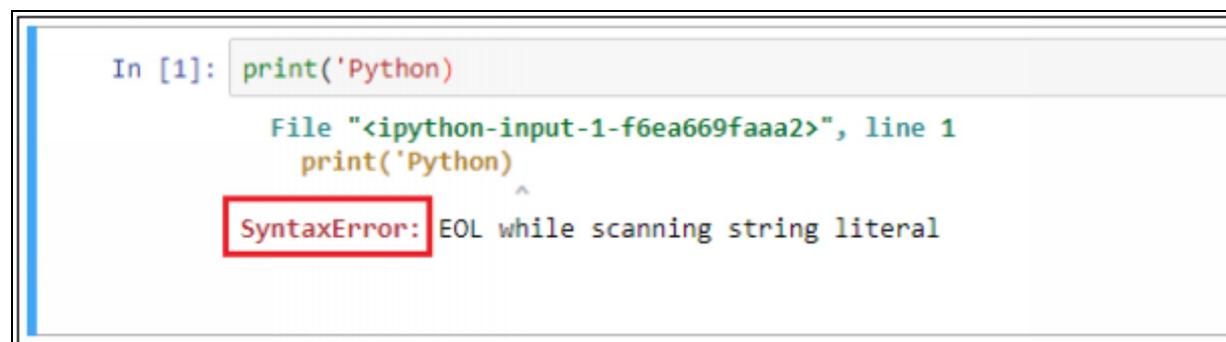
Se quiser mais detalhes a respeito favor visitar o seguinte local:  
<https://docs.python.org/3/tutorial/errors.html>



## Erro de Sintaxe (SyntaxError)

O erro de sintaxe ocorre quando o interpretador não consegue entender o que foi escrito. Esse é um dos erros mais frequentes, pois comumente esquecemos de fechar aspas, parênteses, colchetes etc.

Portanto, esse erro está geralmente ligado a alguma digitação indevida ou ausente. Vejamos um exemplo:



In [1]: `print('Python')`

File "<ipython-input-1-f6ea669faaa2>", line 1  
print('Python')  
^

SyntaxError: EOL while scanning string literal

The screenshot shows a Jupyter Notebook cell with the code `print('Python')`. The error message is displayed in red: `SyntaxError: EOL while scanning string literal`. The word `SyntaxError` is highlighted with a red box.

Note que no exemplo acima está faltando uma aspa simples para que a função `print()` execute seu trabalho.

Temos um erro de sintaxe, isto é, a linguagem da função `print()` dentro da Python foi criada para receber argumentos de strings dentro de aspas. Como isso não foi atendido, o interpretador retornou uma mensagem de erro (SyntaxError).

Atenção a outro exemplo onde estamos esquecendo de fechar um parênteses para a função `print()`.

```
In [2]: print('Olá'  
        File "<ipython-input-2-24187625feba>", line 1  
            print('Olá'  
                  ^  
SyntaxError: unexpected EOF while parsing
```

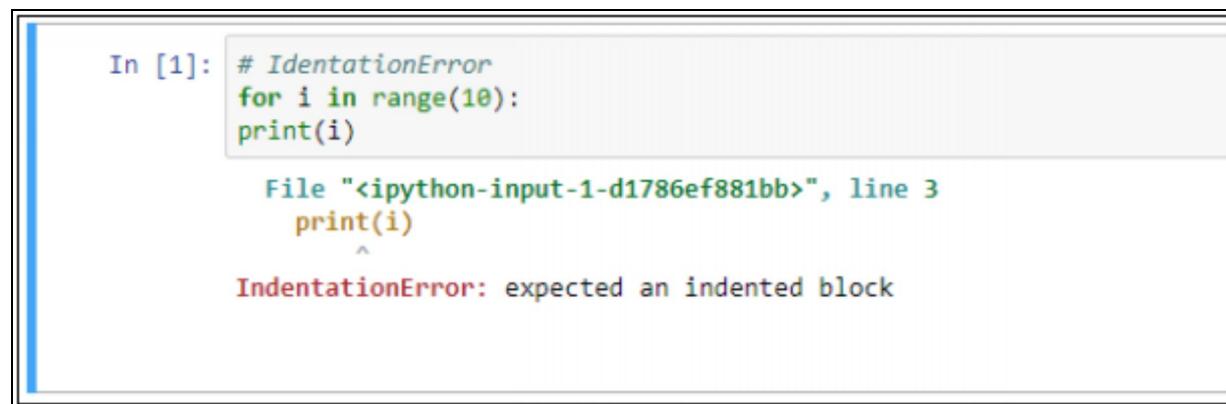
Agora, vejamos um caso onde estamos esquecendo de colocar dois pontos na expressão de um *loop ‘for’*:

```
In [3]: for i in range(2)  
        print(i)  
        File "<ipython-input-3-47dfaf615d55>", line 1  
            for i in range(2)  
                  ^  
SyntaxError: invalid syntax
```

## Erro de Indentação (IndentationError)

Quando não estamos utilizando uma IDE, os erros de indentação são bastante frequentes. Sabemos que, dentre outras coisas, principalmente para não utilizar chaves e colchetes dentro das suas estruturas de códigos, Python utiliza a indentação (espaços entre códigos).

Por exemplo, quando quisermos utilizar o loop ‘`for`’ precisamos de um espaço logo após os dois pontos. Se não respeitarmos isso teremos uma mensagem de erro:



In [1]: # IndentationError  
for i in range(10):  
 print(i)  
  
File "<ipython-input-1-d1786ef881bb>", line 3  
 print(i)  
 ^  
IndentationError: expected an indented block

Quando estamos utilizando editores de texto, podemos ter sérios problemas de indentação caso haja uma mistura entre espaços em branco com tabulações (tabs).

Devemos manter a consistência, por isso precisamos configurar nossos editores de textos para substituir tabs por espaços em branco ou vice-versa.

Portanto, a dica é a seguinte: nunca misture espacos em branco com tabulações nos códigos em Python.

## Erro de nome (NameError)

Lembrando que Python é **case sensitive**, ou seja, leva em conta se a palavra está sendo escrita com letras maiúsculas ou minúsculas. Veja o exemplo de erro de nome em Python.

```
In [2]: # NameError - tentar escrever print com 'p' maiúsculo
Print('Olá, mundo!')

-----
NameError                                 Traceback (most recent call last)
<ipython-input-2-f417129d3629> in <module>()
      1 # NameError
----> 2 Print('Olá, mundo!')

NameError: name 'Print' is not defined
```

Veja agora um erro de nome quando escrevemos errado o nome de alguma variável:

```
In [3]: var1 = 3
var2 = 4

var3 = var1 + var

-----
NameError                                 Traceback (most recent call last)
<ipython-input-3-15957b80b6b3> in <module>()
      2 var2 = 4
      3
----> 4 var3 = var1 + var

NameError: name 'var' is not defined
```

## Erro de Valor (ValueError)

Quando o objetivo é converter valores em tipos específicos de dados podemos nos deparar com uma mensagem de erro de valor. Por exemplo, se tentarmos transformar uma *string* em um tipo inteiro (int) ou flutuante (float).

```
In [2]: nome = 'Python'

In [3]: int(nome)

-----
ValueError                                Traceback (most recent call last)
<ipython-input-3-ed923cbc456f> in <module>()
----> 1 int(nome)

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

## Erro de Index (IndexError)

Para trabalhar com listas, tuplas, dicionários, vetores e matrizes, precisamos ter acesso a seus valores a partir de seus índices. Não custa lembrar que a indexação em Python começa com ZERO.

Vamos estudar um exemplo do acesso de índices de uma string que na Python é representado como uma lista de caracteres e, portanto, cada um desses caracteres pode ser acessado a partir de um índice.

```
In [1]: nome = 'Python'

In [2]: # Tamanho da string 'nome'
len(nome)

Out[2]: 6

In [3]: # Acessando o 6 elemento
nome[6]

-----
IndexError                                                 Traceback (most recent call last)
<ipython-input-3-0a4eccdbcccd6> in <module>()
      1 # Acessando o 6 elemento
----> 2 nome[6]

IndexError: string index out of range
```

Como podemos notar acima, a palavra Python possui 6 letras. A indexação, como sabemos, começa com zero, logo a posição de índice 6 não existe, pois o compilador está tentando acessar a sétima posição.

## Erro de chave (KeyError)

Para entender esse tipo de erro vamos começar com um dicionário chamado de ‘carro’. Lembrando que com dicionários podemos ter acesso a seus valores a partir de índices com strings. Assim, iremos tentar acessar um valor de uma chave inexistente dentro de um dicionário.

```
In [1]: carro = {"Portas": 4,"Ar":False,"Câmbio":"Manual"}  
In [2]: carro  
Out[2]: {'Ar': False, 'Câmbio': 'Manual', 'Portas': 4}  
In [3]: carro["Direção"]  
-----  
KeyError Traceback (most recent call last)  
<ipython-input-3-5901a8337756> in <module>()  
----> 1 carro["Direção"]  
KeyError: 'Direção'
```

## Erro de Tipo (TypeError)

Esse é um dos erros que mais pode causar confusão. Ele é um dos mais frequentes e abrangentes. Certamente, na sua caminhada de aprendizado e amadurecimento com a linguagem Python irá se deparar com muitos desses erros.

Por exemplo, quando estamos fornecendo para uma função mais parâmetros do que o necessário, teremos como retorno esse tipo de erro.

```
In [1]: # Usar a função int() que admite apenas um parâmetro de entrada
        int(77,8) # tentar colocar dois parâmetros como entrada

-----
TypeError                                         Traceback (most recent call last)
<ipython-input-1-057236c4f11d> in <module>()
      1 # Usar a função int() que admite apenas um parâmetro de entrada
----> 2 int(77,8) # tentar colocar dois parâmetros como entrada

TypeError: int() can't convert non-string with explicit base
```

Quando estamos tentando acessar índices de vetores a partir de índices com o tipo não numéricos teremos um erro.

```
In [2]: # Lista
        frutas = ['pera', 'uvas', 'bananas', 'abacate']

In [3]: frutas['uvas']

-----
TypeError                                         Traceback (most recent call last)
<ipython-input-3-a7b49dfbddea> in <module>()
----> 1 frutas['uvas']

TypeError: list indices must be integers or slices, not str
```

Vamos tentar somar um valor **string** com um valor **numérico**.

```
In [4]: nome = 'Python'

In [5]: x = 140

In [6]: nome + x

-----
TypeError                                 Traceback (most recent call last)
<ipython-input-6-e702bdd8a17a> in <module>()
      1 nome + x
----> 1 nome + x

TypeError: must be str, not int
```

## Diferença entre erro e exceção

Erros estão geralmente ligados a problemas de digitação, estrutura incompleta ou errada. Vamos fazer um resumo de como podemos identificar alguns eventuais erros:

Alguns erros não são diretamente indicados pelo interpretador, por isso devemos ter muita atenção e olhar linhas anteriores as indicadas. Assim, sempre que ocorrer um erro devemos:

- a) Verificar a linha onde ocorreu o erro;
- b) Prestar atenção se faltou completar alguma estrutura de algoritmos e elementos como aspas, colchetes, parênteses, vírgulas e pontos.
- c) Os dois pontos depois de **for**, **if**, **else**, **while** geralmente são esquecidos, portanto, bastante atenção neles.
- d) Como Python é *case sensitive* devemos observar se alguma função está sendo escrita com letra maiúscula ou minúscula indevidamente. Em outras palavras, se os nomes estão sendo escritos corretamente.

Mesmo se tudo o que foi feito no código estiver correto ainda podemos receber uma mensagem de **exceção**.

### Erros x Exceções

Vamos compreender agora a sutil diferença entre um erro e uma exceção. Erros estão relacionados a algum problema de interpretação com o compilador, por outro lado as exceções mostram problemas com a execução de alguma regra (matemática ou lógica) onde Python não consegue tratar.

Veja o exemplo de uma função utilizada para dividir dois números

```

In [1]: # Função para dividir dois números
def dividir(a,b):
    return a/b

In [2]: dividir(7,1)
Out[2]: 7.0

In [3]: # tentar dividir por zero
dividir(8/0)

-----
ZeroDivisionError                                 Traceback (most recent call last)
<ipython-input-3-5ebd6bab5eff> in <module>()
      1 # tentar dividir por zero
----> 2 dividir(8/0)

ZeroDivisionError: division by zero

```

Temos vários outros tipos de erro como o **IOError**, **ImportError**, **MemoryError**, **OSError**, dentre outros. Mais detalhes a respeito, favor consultar documentação <https://docs.python.org/2/library/exceptions.html>

No tópico a seguir iremos aprender como criar nossas mensagens de erros e exceções. De outro modo, como tratar ou lidar com esses casos.

### Tratando erros e exceções - Try, Except e Finally.

Para que nossos códigos tenham o desempenho e segurança devida, precisamos tratar os erros e exceções. Para isso a linguagem Python nos disponibiliza diretivas tais como: **try**, **except** e **finally**. Isso irá facilitar a vida dos usuários de nossos códigos.

Vejamos a tentativa de somar um número com uma *string*. Como vimos antes, teremos um erro do tipo **TypeError**. Podemos utilizar o **Try** e **Except** para lidar com esse possível problema.

```
In [1]: nome = 'Python'

In [2]: x = 777

In [3]: try:
        soma = nome + X
    except TypeError:
        print("Operação inválida!")

Operação inválida!
```

Vamos aproveitar o momento para estudar outro tipo de erro:  
**IOError** (erros *input* e *output*).

```
In [1]: # Tratando erros e exceções
try:
    fid = open('dados.txt', 'w')
    fid.write('Gravando dados')
except IOError:
    print("Erro: Arquivo não encontrado/salvo!")
else:
    print("Salvo com sucesso!")
    fid.close()

Salvo com sucesso!
```

Vamos agora tentar abrir um arquivo para leitura que não existe.

```
In [2]: # Tratando erros e exceções
try:
    fid = open('dadosNovo.txt', 'r')
    fid.write('Gravando dados')
except IOError:
    print("Erro: Arquivo não encontrado/salvo!")
else:
    print("Salvo com sucesso!")
    fid.close()

Erro: Arquivo não encontrado/salvo!
```

Vamos utilizar agora o **finally**. Quando estamos interessados em forçar algum tipo de execução de código, independentemente de haver ou não erros e exceções podemos utilizar o **finally**.

```
In [3]: # Tratando erros e exceções
try:
    fid = open('dados.txt', 'r')
except IOError:
    print("Erro: Arquivo não encontrado/salvo!")
else:
    print("Aberto com sucesso!")
    fid.close()
finally:
    print("Esta parte do código é sempre executada!")

Aberto com sucesso!
Esta parte do código é sempre executada!
```

Para terminar o capítulo, devemos notar que o maior desafio para tratar os erros e exceções é preparar o código para os tipos possíveis de erros em questão.

Sempre que bater a dúvida devemos consultar a documentação: <https://docs.python.org/2/library/exceptions.html>

Agora que você já possui uma visão mais ampla da linguagem Python está cada vez mais preparado(a) para começar a vasculhar a documentação.

Lembre-se que a documentação da linguagem é o melhor local para atualizar conhecimentos e confirmar especificações de uso.



# **Capítulo 9**

## Classes e objetos com Python

Nos capítulos anteriores, iniciamos nossos estudos com base na programação estruturada ou procedural. Esse paradigma de programação está alinhado a construção de códigos baseados em sequências, com decisões a serem tomadas muitas vezes com a utilização de processos iterativos.

No atual capítulo, daremos início aos estudos dos objetos que estão trás de todas as bibliotecas e interfaces do Python. Por isso, iremos aqui estudar a programação orientada a objetos.

O paradigma de Programação orientada a objetos (POO) é um modo de programação baseado no conceito de **classes** e **objeto**. Classes são tidas como os moldes formadores dos objetos.

O modo de abstração de um fenômeno a partir de um objeto é fundamentalmente eficaz para modelar seus princípios de funcionamento.

Vale lembrar que tudo em Python são objetos, como por exemplo: *list*, *tuple* e *dict*. Assim neste capítulo você entenderá melhor essa afirmação.

Na POO todo objeto apresenta pelo menos duas características principais: **os atributos e métodos**.

Os atributos dizem respeito às principais características estáticas de um objeto, por outro lado, os métodos tentam representar a dinâmica das ações que podem ser admitidas.

A POO tem uma principal vantagem com relação à programação procedural (aquele que estamos estudando desde o início deste livro) que é a capacidade de aproveitar códigos e fazer manutenções, ajustes e melhorias de maneira muito mais prática, simplificada e rápida.

O uso da POO ganhou popularidade em princípios da década de 1990. Atualmente, existe uma grande variedade de linguagens de programação que suportam a orientação a objetos, tais como: C ++,

Java, C#, e Python claro. Algumas dessas são linguagens de programação também chamadas de multiparadigmas, ou seja, suportam programação orientada a objetos e programação procedural.

Nos subtópicos abaixo apresentaremos uma visão geral das principais características por trás da POO.

Leva certo tempo para que possamos consolidar os conhecimentos desse paradigma de programação, pois é um tema avançado e precisa de muita atenção e esforço nos estudos preliminares.

## Principais características da POO

A POO apresenta quatro características principais que a tornam distintivamente útil, a saber: **abstração, encapsulamento, herança e polimorfismo**.

Fazendo uso de todas essas características, a POO ajuda a reduzir a complexidade, implicando em menos erros. Assim, temos um código mais seguro e sustentável no médio e longo prazo.

### Abstração

A abstração consiste numa maneira de representação de um objeto dentro de um sistema ou fenômeno. Dessa forma, temos que representar/modelar o que esse objeto irá realizar dentro de nosso sistema. Nessa abstração devemos considerar três pontos principais que são: **identidade, propriedade e método**.

Para que não haja conflito, a identidade para um objeto deve ser única dentro do sistema. Na maior parte das linguagens há o conceito de pacotes (ou **namespaces**) nos quais **não devemos** repetir a identidade do objeto.

As propriedades de um objeto referem-se às características que ele possui. No mundo real qualquer objeto possui elementos que o definem. Dentro dos nossos modelos com POO, essas características são nomeadas propriedades.

Por exemplo, as propriedades de um objeto “Elefante” poderiam ser “Tamanho”, “Peso” e “Idade”.

Por fim, mas não menos importante, temos as propriedades dinâmicas de um objeto, que são as ações ou os chamados métodos. Esses métodos podem ser os mais diversos possíveis, como, por exemplo, para o elefante temos comer(), andar() e dormir().

## Encapsulamento

O encapsulamento é uma característica marcante da POO. Está relacionado à segurança de funcionamento da representação de objetos pelo fato de esconder as propriedades, criando uma espécie de caixa preta. Dessa maneira, detalhes internos de uma classe podem permanecer ocultos e/ou protegidos para um objeto.

As linguagens orientadas a objetos implementam o encapsulamento baseado em propriedades privadas. Essa atitude evita o acesso direto à propriedade do objeto, adicionando uma camada a mais de segurança à aplicação.

## Herança

A reutilização de códigos é uma das melhores vantagens da POO. O conceito de herança surge com este objetivo de reutilização de códigos. Essa característica otimiza a produção de programas em tempo e linhas de código escritas.

## Polimorfismo

O polimorfismo permite que nossos objetos que herdam características possam alterar seu funcionamento interno a partir de métodos herdados de um objeto pai.

É normal que boa parte desses conceitos, em um primeiro momento, fiquem obscuros para você que está iniciando. Mesmo assim continue avançando a leitura e depois releia as partes que não ficaram tão claras.

# Classes e Objetos no Python

O mundo real é repleto de objetos que possuem propriedades e métodos próprios. Com Python podemos fazer a representação virtual de tudo isso. E o nível de detalhamento dos objetos somos nós que definiremos.

Vejamos a criação de uma **classe** chamada **Animal** com o Python. Por convenção, o nome de uma classe deve **começar** com letra **maiúscula**.

A palavra reservada **class** em Python é utilizada para criar classes. Quando estamos declarando classe, estamos criando um molde que possibilitará a criação de novos objetos. Dentro das classes devemos especificar os métodos e atributos que serão concedidos aos objetos da classe.

```
In [1]: # Cirar uma classe
class Animal:
    def __init__(self,tipo, nome=None, idade=None, sexo=None):
        self.tipo = tipo
        self.nome = nome
        self.idade = idade
        self.sexo = sexo

    def comer(self):
        return print("O ", self.tipo, " chamado ", self.nome, " está comendo!")
```

Como já discutimos, métodos são funções/ações associadas a um objeto. A diretiva **\_\_init\_\_** define um método especial. O método **\_\_init\_\_** será chamado sempre que estamos instanciando (criamos) um determinado objeto a partir de uma classe específica. Esse método é chamado de construtor, que inicia nosso objeto com alguns valores padrão. Assim, toda vez que instanciamos um novo objeto o construtor é chamado.

No início dos estudos de POO ficamos confusos com algumas etapas. Fique calmo que isso é normal. Por exemplo, o parâmetro

`self` está sendo inserido dentro do método `__init__`. Esse é um padrão utilizado pelos desenvolvedores.

O `self` é uma referência a cada atributo de um objeto criado a partir da classe. Nos exemplos a seguir esse uso ficará mais evidente.

Note que na classe **Animal** tem um método chamado `comer()`. Uma vez instanciado um objeto nós teremos acesso aos atributos tipo, nome, idade e sexo bem como do método ‘comer’. Veja abaixo como criar uma instância de um animal.

```
In [2]: # Criar um objeto animal chamado cachorro
cachorro = Animal("Cachorro", "Bob")

In [ ]: cachorro.|
```

The screenshot shows a Jupyter Notebook cell with the following code:

```
In [2]: # Criar um objeto animal chamado cachorro
cachorro = Animal("Cachorro", "Bob")
```

Below the cell, the cursor is at the end of the line `cachorro.|`. A code completion dropdown menu is open, listing the following methods and attributes:

- cachorro.comer
- cachorro.idade
- cachorro.nome
- cachorro.sexo
- cachorro.tipo

Os atributos e métodos da classe **Animal** agora podem ser encontrados no objeto `cachorro`. Para chamarmos os atributos de um objeto não precisamos abrir e fechar parênteses.

```
In [3]: cachorro.tipo
Out[3]: 'Cachorro'

In [4]: cachorro.idade

In [5]: cachorro.comer()
        O Cachorro chamado Bob está comendo!

In [6]: type(cachorro)
Out[6]: __main__.Animal
```

The screenshot shows a Jupyter Notebook cell with the following interactions:

- `In [3]: cachorro.tipo` → `Out[3]: 'Cachorro'`
- `In [4]: cachorro.idade`
- `In [5]: cachorro.comer()` → Output:  
O Cachorro chamado Bob está comendo!
- `In [6]: type(cachorro)` → `Out[6]: __main__.Animal`

Lembrando que quando criamos o objeto cachorro não adicionamos a idade e o sexo. Vamos agora criar outro objeto chamado gato utilizando todas as possibilidades fornecidas pelo construtor da classe.

```
In [8]: gato = Animal("Felino","Felix",6,"M")
In [9]: type(gato)
Out[9]: __main__.Animal
In [10]: gato.idade
Out[10]: 6
In [11]: gato.nome
Out[11]: 'Felix'
In [12]: gato.sexo
Out[12]: 'M'
```

Agora vamos pedir para o objeto **gato** executar o método **comer**. Métodos, diferentemente dos atributos, são sempre chamados abrindo e fechando parênteses.

```
In [13]: gato.comer()
O Felino chamado Felix está comendo!
```

Assim o objeto gato é uma instância da classe Animal. Portanto, o gato, uma vez instanciado, passa a ter os mesmos atributos e métodos da classe animal.

Veja outra maneira de instanciar um objeto atribuindo um valor específico para o construtor.

```
In [15]: girafa = Animal(tipo = "Girafa", idade="30")
In [17]: girafa.comer()
          O Girafa chamado None , está comendo!
```

## Em Python tudo é um objeto

Agora que já temos uma ideia do que são classes e objetos vamos fazer um estudo curioso e comprovar que tudo em Python é um objeto. Vamos pedir para a Python retornar os tipos das estruturas formadas com ( ), { } e [ ], por exemplo.

```
In [1]: # colchetes definem listas
        print(type([]))

        <class 'list'>

In [2]: # chaves definem dicionários
        print(type({}))

        <class 'dict'>

In [3]: # parênteses definem tuplas
        print(type(()))

        <class 'tuple'>
```

Agora sabemos como construir as nossas próprias classes.

## Herança em Python

De modo resumido, herança é uma maneira de gerar novas classes utilizando outras classes (classe pai) previamente definidas.

A herança ocorre quando uma classe (filha) herda características e métodos de outra classe (pai), mas não impede que a classe filha possua seus próprios métodos e atributos.

A principal vantagem aqui é poder reutilizar o código reduzindo assim a complexidade.

Vamos criar a classe **Pessoa** depois utilizar a herança para criar particularidades como **PessoaFísica** e **PessoaJurídica**.

```
In [1]: # Cirar uma classe
class Pessoa:
    def __init__(self, nome, idade):
        self.nome = nome
        self.idade = idade

    # Encapsulamento de métodos para
    # setar e obter nomes e idades
    def setNome(self, nome):
        self.nome = nome

    def setIdade(self, idade):
        self.idade = idade

    def getNome(self):
        return self.nome

    def getIdade(self):
        return self.idade
```

Vamos agora instanciar uma pessoa:

```
In [2]: p1 = Pessoa("Paulo",21)
```

```
In [3]: p1.getNome()
```

```
Out[3]: 'Paulo'
```

```
In [4]: p1.getNome()
```

```
Out[4]: 'Paulo'
```

```
In [5]: p1.getIdade()
```

```
Out[5]: 21
```

Os métodos **getNome**, **getIdade**, **setNome** e **setIdade** servem para **encapsular** a manipulação de variáveis dentro dos objetos. Podemos a partir desses métodos criar critérios específicos para a modelagem dos nossos problemas.

Vamos criar agora a classe **PessoaFísica** que herda as características da classe Pessoa. Estamos fazendo isso para não precisarmos reescrever características da classe pai Pessoa na classe filho **PessoaFísica**.

Veja abaixo como se dá esse processo de herança no Python

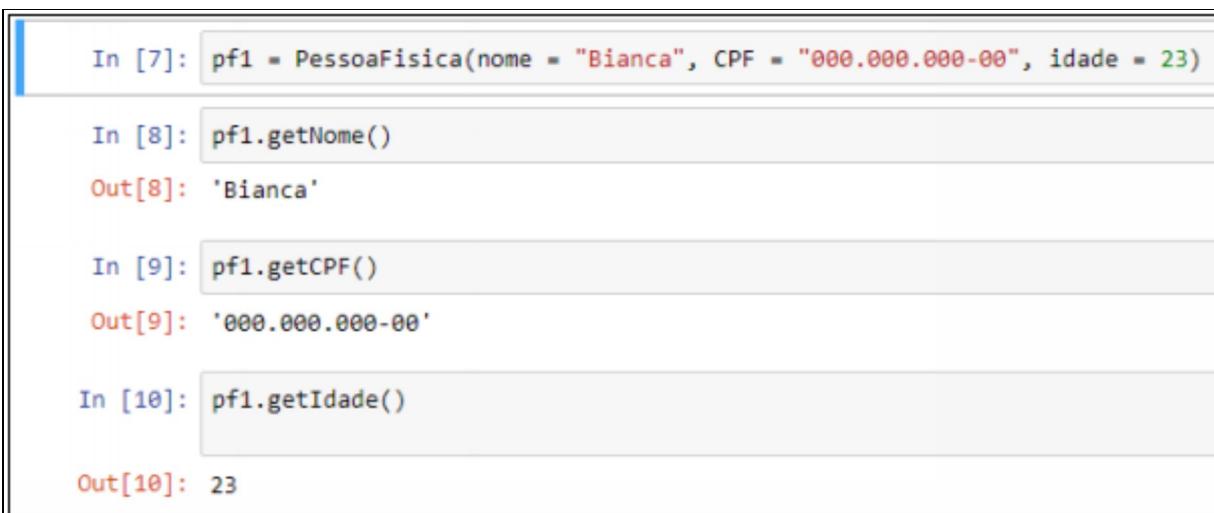
```
In [6]: # Classe PessoaFísica herda da classe Pessoa
class PessoaFísica(Pessoa):
    # Construtor para classe pessoa física
    def __init__(self, CPF, nome, idade):
        # importa características da classe Pessoa
        super().__init__(nome,idade)
        self.CPF = CPF

    # pessoas físicas possuem cpf
    def setCPF(self, CPF):
        self.CPF = CPF

    def getCPF(self):
        return self.CPF
```

O método **super()** faz a coleta dos atributos da superclasse (classe pai) que no nosso caso é a classe **Pessoa**. Com isso, utilizamos o `super().__init__(nome, idade)` para criar um construtor com os parâmetros da classe pais.

Note que temos para a classe **PessoaFisica** os mesmos métodos da classe **Pessoa** mais dois outros métodos **setCPF** e **getCPF**. Isso porque é inerente para uma pessoa física ter um CPF.



The screenshot shows a Jupyter Notebook interface with five code cells. Cell In [7] creates a `PessoaFisica` object named `pf1` with attributes `nome = "Bianca"`, `CPF = "000.000.000-00"`, and `idade = 23`. Cells In [8], Out[8], In [9], Out[9], and In [10] show the results of calling `getNome()`, `getCPF()`, and `getIdade()` respectively on `pf1`.

```
In [7]: pf1 = PessoaFisica(nome = "Bianca", CPF = "000.000.000-00", idade = 23)
In [8]: pf1.getNome()
Out[8]: 'Bianca'
In [9]: pf1.getCPF()
Out[9]: '000.000.000-00'
In [10]: pf1.getIdade()
Out[10]: 23
```

Podemos fazer o mesmo agora para outra classe chamada de **PessoaJuridica**, herdando todas as características da classe `pessoa` e criando um atributo novo chamado de CNPJ que é inerente a toda empresa.

Note com isso a abstração que tomamos, em que uma classe **Pessoa** tem atributos e métodos comuns para **PessoaFisica** e **PessoaJuridica**.

## Métodos especiais em Python

Até o presente momento fizemos uso de apenas um método especial em Python que foi o `__init__`. No entanto, o Python apresenta um grande número desses métodos que nos permitem executar ações específicas. Veja abaixo alguns dos principais métodos especiais em Python.

<code>__abs__</code>	<code>__delattr__</code>	<code>__getattribute__</code>	<code>__index__</code>
<code>__add__</code>	<code>__delete__</code>	<code>__getitem__</code>	<code>__init__</code>
<code>__and__</code>	<code>__delitem__</code>	<code>__getslice__</code>	<code>__instancecheck__</code>
<code>__call__</code>	<code>__delslice__</code>	<code>__gt__</code>	<code>__int__</code>
<code>__class__</code>	<code>__dict__</code>	<code>__hash__</code>	<code>__invert__</code>
<code>__cmp__</code>	<code>__div__</code>	<code>__hex__</code>	<code>__ior__</code>
<code>__coerce__</code>	<code>__divmod__</code>	<code>__iadd__</code>	<code>__ipow__</code>
<code>__complex__</code>	<code>__eq__</code>	<code>__iand__</code>	<code>__irshift__</code>
<code>__contains__</code>	<code>__float__</code>	<code>__idiv__</code>	<code>__isub__</code>
<code>__del__</code>	<code>__floordiv__</code>	<code>__ifloordiv__</code>	<code>__long__</code>
<code>__itruediv__</code>	<code>__ge__</code>	<code>__ilshift__</code>	<code>__lshift__</code>
<code>__ixor__</code>	<code>__get__</code>	<code>__imod__</code>	<code>__mod__</code>
<code>__len__</code>	<code>__getattr__</code>	<code>__imul__</code>	<code>__new__</code>

Note que identificamos um método especial pelos dois underscores (`_`) antes e após seu nome. Vale lembrar que *underline* é **diferente** de *underscore*. **Underline** significar um sublinhamento. Por outro lado, o **underscore** é traço inferior utilizado para separar palavras, por exemplo. Note que muita gente confunde isso. Agora você nunca mais falará errado.

```
In [1]: # Criar classe
class Filme():
    def __init__(self, titulo, diretor, duracao):
        print("Filme criado")
        self.titulo = titulo
        self.diretor = diretor
        self.duracao = duracao

    def __str__(self):
        # usamos a contra barra '\ para continuar o comando na
        # proxima linha!
        return "Filme: %s , Diretor: %s , Duração (min): %s"\ 
            %(self.titulo, self.diretor, self.duracao)

    def __len__(self): # este é um método especial modificado
        return self.duracao

    def len(self): # este nós estamos criando
        return print("Duração do filme em minutos: ", self.duracao)
```

```
In [2]: filme1 = Filme("Sétimo Selo", "Ingmar Bergman", 96)
Filme criado

In [3]: # Por causa do método __str__ iremos ter uma resposta de texto:
print(filme1)

Filme: Sétimo Selo , Diretor: Ingmar Bergman , Duração (min): 96

In [4]: # Chamar uma função especial
str(filme1)

Out[4]: 'Filme: Sétimo Selo , Diretor: Ingmar Bergman , Duração (min): 96'
```

```
In [5]: # Usar o método especial modificado por nós:
len(filme1)
```

```
Out[5]: 96
```

```
In [6]: # Usar o método Len() criado por nós
filme1.len()
```

```
Duração do filme em minutos: 96
```

Vale ressaltar que vimos apenas o básico introdutório desse campo tão vasto e importante. Para que você possa se aprofundar e utilizar com mais confiança a POO certamente precisará de leitura de materiais mais avançados. Para isso é extremamente aconselhável a leitura do excelente e completo livro: [Python Fluente](#) do autor *Luciano Ramalho* publicado pela editora Novatec.



# **Capítulo 10**

## Conexão com banco de dados

Depois do grande poder de processamento computacional, o armazenamento de dados é um dos maiores benefícios trazidos pelos computadores. A memória humana é bastante limitada e volátil.

Estima-se que temos algo em torno de **1.000 terabytes** de memória, que por si só é um valor surpreendente, porém bastante limitado comparada aos datacenters espalhados pelo mundo.

O conhecimento se baseia na coleta e armazenamento de dados e informações. Na computação científica, a maneira mais segura e eficiente de armazenar e recuperar dados são a partir de bancos de dados.

Os dados que precisamos armazenar podem ser classificados em dois grandes grupos principais: dados relacionais e não relacionais.

Bancos de dados relacionais são aqueles que apresentam uma estrutura de informações bem definidas como, por exemplo, uma tabela contendo informações de um cliente.

Já banco de dados não relacionais são aqueles cujos dados são armazenados e não temos especificações prévias bem estabelecidas de como os dados serão armazenados.

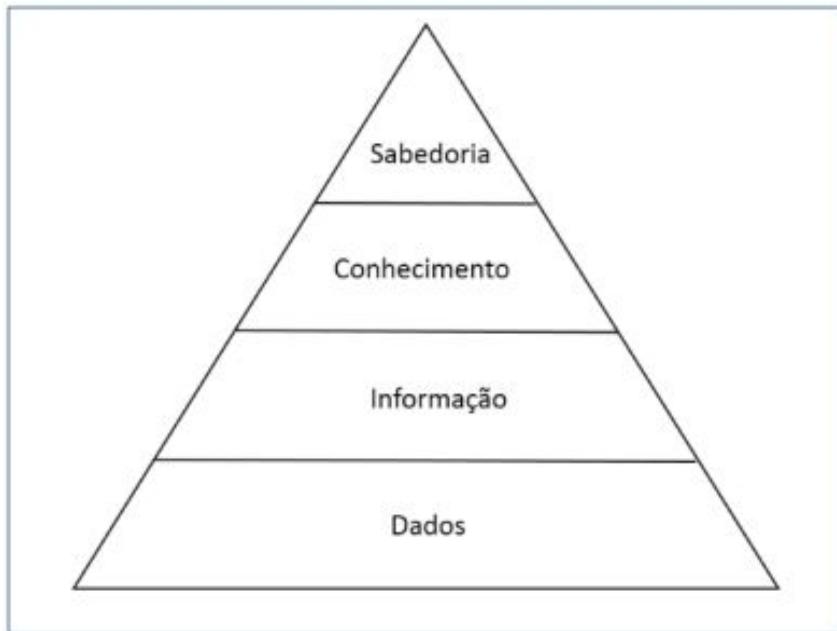
O mundo do **big data** está cada vez mais abrindo portas para diversos segmentos de estudos. A grande maioria dos dados que estão sendo gerados atualmente é de natureza não estruturada, tais como textos, fotos, vídeos, áudios etc.

Os bancos de dados relacionais são gerenciados por um **RDBMS** (*relational database management system*). Os dados nesse tipo de sistema são armazenados em um formato de tabela e o relacionamento entre elas é gerenciado pelo RDBMS.

Existe uma cadeia evolutiva de chegada dos dados até o conhecimento. Podemos extrair informações de dados quando esses estão agregados em contextos.

As informações conectadas e agregadas de tal maneira a permitir que novas informações sejam criadas podemos dar o nome de conhecimento.

A esses conhecimentos emaranhados em uma estrutura sólida e harmoniosa com potencial gerador de novos conhecimentos damos o nome de sabedoria. Veja o diagrama evolutivo abaixo:



Neste capítulo iremos nos deter ao estudo de dados estruturados com o Python utilizando o **SQLite**.

Abordaremos os conceitos introdutórios do SQLite que é um banco de dados leve e completo, utilizado na maioria dos dispositivos móveis.

## Linguagem SQL

Existe uma linguagem de consulta estruturada de dados bastante utilizada chamada de **SQL** (*Structured Query Language*). Com essa linguagem podemos criar banco de dados, fazer consultas, bem como, manipulações de inserção, atualização, alteração e remoção de registos.

Assim, a SQL é uma linguagem especializada na manipulação de dados baseada numa lógica relacional ou modelo relacional.

Portanto, a **SQL** é uma linguagem para trabalhar com banco de dados relacionais onde podemos, a partir de um conjunto de dados estruturalmente organizados, facilmente acessar, gerenciar e atualizar os dados.

Os dados são manipulados a partir dos comandos:

- **SELECT**: *usado para pesquisar dados.*
- **UPDATE**: *atualiza dados.*
- **DELETE**: *eliminação de dados.*
- **INSERT**: *inserção de dados.*

Os comandos auxiliares de porte da linguagem SQL são:

- **CREATE**: *define um objeto (tabela, índice).*
- **ALTER**: *altera um objeto.*
- **DROP**: *elimina um objeto.*

Por fim, temos os comandos de suporte e desenvolvimento:

- **GRANT**: *fornecce privilégios aos objetos.*
- **REVOKE**: *tira privilégios.*

Existem diversos sistemas gerenciadores de bancos de dados relacionais no mercado, todos eles trabalham com linguagem **SQL** em *background*, os principais são:

- MySQL

- Oracle
- SQL Server
- PostgreSQL
- SQLite



Entretanto, como mencionado anteriormente iremos utilizar o SQLite.

## **SQLite**

Quando estamos interessados em trabalhar com um inter-relacionamento de dados dentro de um domínio específico, a linguagem Python nos fornece acesso ao **SQLite**. Esse é um sistema gerenciador de banco de dados (SGBD) muito utilizado por ser bastante leve e prático.

A grande vantagem de utilizar o **SQLite** ao invés de um arquivo **.txt** ou **.dat**, por exemplo, é que ganhamos desempenho com relação ao uso de memória, uma vez que os arquivos de um banco de dados são carregados parcialmente na memória, o que não acontece com um arquivo **.txt**, por exemplo.

Veja abaixo outras **vantagens** da utilização do **SQLite**:

- Ele não precisa de um processo dedicado de servidor;
- Não é preciso fazer configurações por parte do administrador;
- Um banco de dados SQLite é carregado em apenas um arquivo multiplataforma (Linux, Mac OS, Android, iOS e Windows);
- O SQLite é bastante leve, possui menos de **400 KB**.

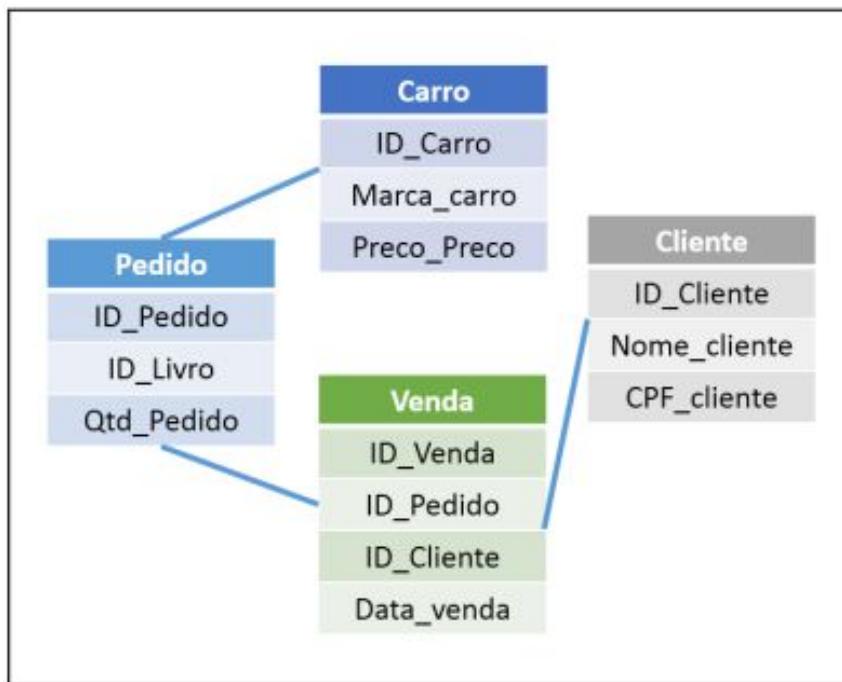
O SQLite trabalha sobre o ponto de vista relacional e, portanto, faz uso dos seguintes parâmetros de estruturação:

- Entidades (tabelas e atributos).
- Registros (**tuplas**).
- Chaves (primária e estrangeira).
- Relacionamentos entre entidades (tabelas).
- Integridade referencial.
- Normalização.

### *Entidades*

As entidades são as tabelas que podem ser armazenadas dentro de um banco de dados.

Em um banco de dados SQLite pode existir uma ou dezenas de centenas de tabelas. Veja o diagrama abaixo que apresente um exemplo de estrutura de banco de dados relacionais.



## Registros

Os registros são cada linha formada por uma lista ordenada de colunas. Os registros também são chamados de tupla (**atenção** não confundir com tupla em Python).

Código	Nome	Descrição	Uso	Ano de Lançamento
P-1200	Python	Linguagem de uso geral	Análise de Dados	1991
R-1300	R	Linguagem Estatística	Análise de Dados	1990
J-1400	Scala	Linguagem de uso geral	Processamento de Big Data	2001

As colunas de uma tabela são também chamadas de atributos. E o domínio é o conjunto de valores que um atributo pode assumir, por exemplo, domínios numéricos, *strings* e booleanos.

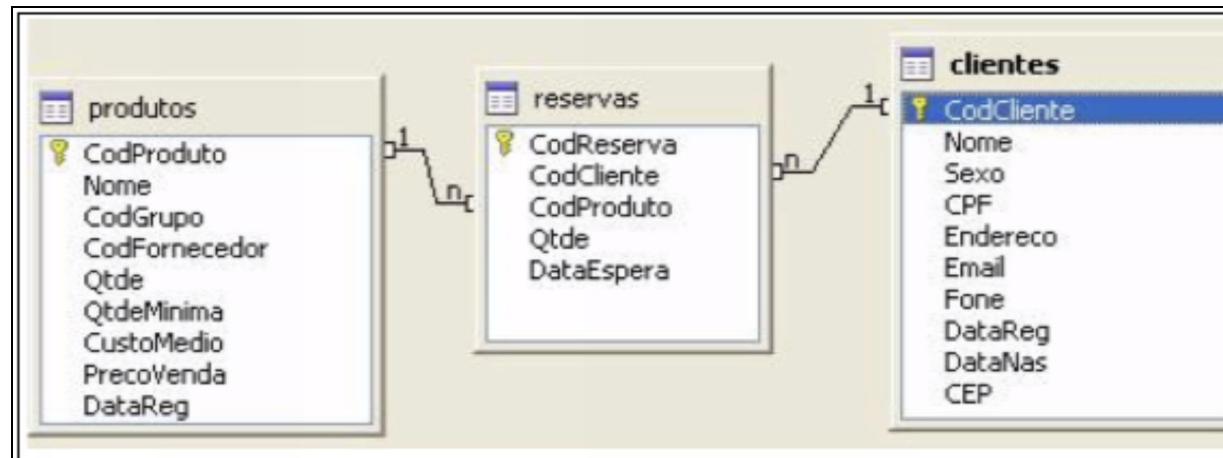
## Chaves

O conceito que vai fazer a conexão relacional entre as tabelas dentro de um banco de dados é o de chave. A chave permite identificar e diferenciar uma tupla de outra. Portanto, com as chaves podemos estabelecer os possíveis múltiplos relacionamentos entre as tabelas. As chaves determinam a unicidade de cada registro.

Existem dois tipos de chaves: **primárias** (PK) e **estrangeira** (FK). Uma tabela só pode ter uma chave primária que garante a unicidade de uma ou mais linhas (registros). Por outro lado, a chave estrangeira são valores da PK de outra tabela, ou seja, faz o relacionamento com a chave primária de outra tabela. Uma tabela pode ter mais de uma chave estrangeira.

## Relacionamentos

Os relacionamentos entre as tabelas são feitos a partir do uso das chaves primárias e estrangeiras. Veja o exemplo abaixo:



## Integridade referencial

Para que o relacionamento entre as tabelas dentro de um banco de dados esteja coerente temos o conceito de integridade referencial que trabalha com ações de adição, exclusão e atualização em cascata. Por exemplo, eu não posso remover um registro sem antes remover os outros relacionados a este primeiro.

## Normalização de dados

Quando estamos pretendendo reduzir a redundância e dependência entre as tabelas no nosso banco de dados podemos utilizar a normalização de dados. Este processo geralmente envolve a redução de tabelas grandes em tabelas menores. Isso está ligado ao trabalho do módulo de integridade referencial.

O objetivo com a normalização de dados é estruturar as tabelas de maneira que a alteração de exclusão de uma entidade possa ser feita em apenas uma tabela e se propagar para as demais através dos relacionamentos.

## Python e SQLite

Vamos agora estudar na prática o funcionamento do SQLite utilizando o Python. Primeiro precisamos aprender como criar um banco de dados e abrir uma conexão com ele. Veja o código abaixo:

```
In [1]: # Remover algum banco de dados caso exista
import os
os.remove('morador.db') if os.path.exists('morador.db') else None

In [2]: # Importar o SQLite - já vem com o anaconda!
import sqlite3

In [3]: # Criar uma conexão com o banco de dados.
con = sqlite3.connect('morador.db')

In [4]: type(con)

Out[4]: sqlite3.Connection
```

Estamos removendo o arquivo, caso exista, para fins didáticos. A depender do nosso sistema não precisamos fazer essa remoção ao executar o *Notebook*.

O método **connect()** tenta conectar um banco de dados, caso exista, do contrário ele cria um novo banco de dados com o nome especificado. Caso não seja especificado um caminho do diretório para o arquivo do banco de dados o mesmo será criado no diretório atual do Jupyter *Notebook*.

Uma vez criada a conexão, precisamos criar um **cursor()** a partir dela que nos permita percorrer e fazer modificações em todos os registros do banco de dados.

```
In [5]: # Criar um cursos  
cur = con.cursor()
```

```
In [6]: type(cur)
```

```
Out[6]: sqlite3.Cursor
```

Com o **cursor()** criado temos que dominar a linguagem SQL para fazer as modificações requeridas dentro do banco de dados. Neste capítulo iremos aprender os principais comandos da linguagem.

Vamos começar com a criação de uma tabela. Precisamos passar para o cursor uma *string* contendo os comandos SQL<sup>[13]</sup> necessários para essa criação.

```
In [7]: # Criar uma tabela - comando SQL  
sql_create = 'create table apartamento '\  
'(id integer primary key, '\  
'morador varchar(100), '\  
'andar varchar(50))'
```

```
In [8]: # Executar comando SQL no cursor()  
cur.execute(sql_create)
```

```
Out[8]: <sqlite3.Cursor at 0x57b06c0>
```

Nossa tabela tem uma **chave primária** (PK - *primary key*) que escolhemos o **id** (nº de identificação) para representá-la.

A chave primária indica que os valores de **id** não podem ser repetidos. Criamos duas colunas: morador e andar. Ambas são do tipo **varchar(100)** com 100 posições para alocar caracteres para o morador e **varchar(50)** com cinquenta posições para o andar.

Assim criamos a tabela com suas colunas e valores que elas podem admitir. Precisamos inserir algum dado a nossa tabela. Vejamos como fazer:

```
In [9]: # Sentença SQL para inserir registros na tabela criada  
sql_insert = 'insert into apartamento values (?, ?, ?)'
```

```
In [10]: # Dados que serão inseridos  
registros = [(100, 'Ana Paula', 'Térreo'),  
             (101, 'Paulo Victor', 'Primeiro'),  
             (102, 'Roberta Silva', 'Segundo'),  
             (103, 'Adalto Farias', 'Terceiro')]
```

```
In [11]: # Inserir registros  
for registro in registros:  
    cur.execute(sql_insert,registro)
```

```
In [12]: # Precisamos gravar a transação no banco de dados  
# Sem o commit os dados não são gravados!  
con.commit()
```

Agora que já gravamos nossos dados vamos averiguar se eles realmente estão lá. Para isso devemos utilizar o comando SQL chamado de **select**.

```
In [13]: # Sentença SQL para buscar (selecionar) registros  
sql_select = 'select * from apartamento'
```

```
In [14]: # Executar comando select  
cur.execute(sql_select)
```

```
Out[14]: <sqlite3.Cursor at 0x58306c0>
```

```
In [15]: # Recuperar os registros e armazená-los em dados  
dados = cur.fetchall()
```

```
In [16]: dados
```

```
Out[16]: [(100, 'Ana Paula', 'Térreo'),  
           (101, 'Paulo Victor', 'Primeiro'),  
           (102, 'Roberta Silva', 'Segundo'),  
           (103, 'Adalto Farias', 'Terceiro')]
```

Como mostrado acima podemos comprovar que os dados foram realmente gravados. Vejamos outra maneira de retornar os dados.

```
In [17]: for linha in dados:  
    print('Apartamento Id: %d. Morador: %s, Andar: %s \n' %linha)  
  
Apartamento Id: 100. Morador: Ana Paula, Andar: Térreo  
  
Apartamento Id: 101. Morador: Paulo Victor, Andar: Primeiro  
  
Apartamento Id: 102. Morador: Roberta Silva, Andar: Segundo  
  
Apartamento Id: 103. Morador: Adalto Farias, Andar: Terceiro
```

Dessa forma podemos ir salvando os dados necessários no nosso banco. Sempre quando terminarmos de trabalhar com o banco de dados devemos fechar a conexão com o comando **close()**.

```
In [18]: con.close()
```

A linguagem SQL é bastante extensa e completa. Podemos fazer filtros, atualizações, remoções e muito mais. Agora que você conhece os primeiros passos, ficará mais fácil prosseguir com estudos mais avançados. Para isso, nada melhor do que acessar a documentação oficial do SQLite (<https://www.sqlite.org/docs.html>) .



# Agradecimentos

Se você chegou até aqui está certamente munido de uma **boa introdução** a uma das linguagens mais interessantes e importantes para qualquer programador que é Python.

O seu crescimento e desenvolvimento com Python agora está em suas mãos. Tudo dependerá de seu interesse e curiosidade daqui para frente.

Agradeço a você por confiar na aquisição deste material.

Espero que tenha ajudado a plantar a sementinha da curiosidade e motivado você para continuar avançando os estudos nessa tão relevante linguagem de programação.

A partir de agora, tenha muita paciência, força e dedicação para continuar estudando coisas mais avançadas.

Muito sucesso para todos nós!

Mais uma vez obrigado!

- *Rafael F.V.C. Santos*

## Conheça outros cursos do Autor:

- [Python Fundamentos](http://bit.ly/python-fundamentos) (<http://bit.ly/python-fundamentos>) GRÁTIS
- [Estatística Descritiva com Python](http://bit.ly/stats-descritiva-python) (<http://bit.ly/stats-descritiva-python>)
- [Álgebra Linear com Python](http://bit.ly/alg-linear-python) (<http://bit.ly/alg-linear-python>)
- [Crie seu 1º Robô de Investimentos com MQL5](http://bit.ly/1-robo-invest-mql5) (<http://bit.ly/1-robo-invest-mql5>)
- [Estratégias avançadas para robôs invest.](http://bit.ly/mql5-avancado-1) (<http://bit.ly/mql5-avancado-1>)
- [Como criar indicadores técnicos de Invest.](http://bit.ly/ind-tecnicos-mql5) (<http://bit.ly/ind-tecnicos-mql5>)
- [Matemática para Data Science – Pré-Cálculo](http://bit.ly/mat-pre-calculo) (<http://bit.ly/mat-pre-calculo>)
- [Manipulação e análise de dados com Pandas Python](http://bit.ly/pandas-python) (<http://bit.ly/pandas-python>)
- [Fundamentos da linguagem R – Data Science Tool](http://bit.ly/aprenda-linguagem-r) (<http://bit.ly/aprenda-linguagem-r>)
- [Análise técnica para investidores e analistas Quant.](#)

## Mais e-books do autor

- [Introdução ao MetaTrader 5 e programação com MQL5.](#)
- [Álgebra Linear com Python: Aprenda na prática os principais conceitos.](#)
- [Estudo Estatístico dos Padrões de Candles: PIN BARS \(Martelos\).](#)
- [Como Otimizar Robôs de Investimentos com MetaTrader 5.](#)
- [Introdução ao Autocoaching.](#)

--\*\*> **FIM** <\*\*--

---

[1] <https://pt.wikipedia.org/wiki/Framework>

[2] <https://www.tiobe.com/tiobe-index/>

[3] <https://insights.stackoverflow.com/survey/2019/>

[4] <https://github.com/>

[5] <https://pypi.org/>

[6] [https://pt.wikipedia.org/wiki/C%C3%B3digo\\_aberto](https://pt.wikipedia.org/wiki/C%C3%B3digo_aberto)

[7] <https://www.anaconda.com>

[8] <https://www.w3schools.com/css/>

[9] <https://pt.wikipedia.org/wiki/Case-sensitive>

[10] <https://pt.wikipedia.org/wiki/Itera%C3%A7%C3%A3o>

[11] Mais detalhes no Livro: [Álgebra Linear com Python](#) - <https://goo.gl/kDhYRf>

[12] <https://pandas.pydata.org/>

[13] <https://www.w3schools.com/sql/>