

CURSO PROGRAMAÇÃO EM PYTHON

Sobre o Curso:

Aprenda os principais conceitos da Linguagem Python de uma forma prática e intuitiva com diversas situações cotidianas.

APRESENTAÇÃO

O MATERIAL

Este material foi construído pensando em seu processo de aprendizagem. Nele você encontrará informações importantes que poderão ser usadas como referência para criar coisas fantásticas usando Python.

Python é uma poderosa linguagem, atual e utilizada em larga escala para aplicações e análise de dados, logo aprender como ela funciona pode ser o que você precisa.

O CURSO

Este curso ensina Python por meio da construção de algoritmos que simulam situações cotidianas com foco na automação de tarefas.

Você aprenderá os conceitos da linguagem abordadas com situações práticas usando recursos como ler planilhas, mandar e-mails e ler páginas web.

APRENDIZAGEM

O curso é totalmente voltado para a prática, e, ao longo do mesmo você irá desenvolver algoritmos e sistemas que lhe ajudarão a fixar os conceitos apresentados em suas aulas.

CONTROLE DE VERSÃO

As informações contidas neste material se referem ao curso de Programação em Python. Abaixo é apresentado o controle de versão deste material e respectivos autores e revisores.

Data	Versão	Autor(es)	Revisor(es)
18/11/2019	1.0	Vinício Schmidt	Marcello Avella
02/12/2019	1.1	Vinício Schmidt	Marcello Avella
13/12/2019	1.2	Vinício Schmidt	Marcello Avella

SUMÁRIO

O Material.....	2
O Curso.....	2
Aprendizagem	2
Controle de Versão	2
Introdução.....	7
O início.....	7
Preparando o Ambiente.....	10
Instalação do Python	10
Utilização do terminal	12
IDLE do Python.....	14
VSCode	18
Fundamentos da Linguagem Python.....	30
Operadores	30
Variáveis	32
Tipos de dados:	35
Operadores de Comparação	37
Operadores booleanos.....	38
Estruturas Condicionais.....	39
Tomada de Decisões	39
Exercício proposto	42
Exercícios Complementares.....	43
Estruturas de repetição.....	45
While.....	45
O Debug.....	46
Break.....	50
Escopo de variáveis	51
Módulos.....	52
Exercício proposto	54
Exercícios complementares.....	54
Coleções Python.....	56
Listas	56
Tuplas	66
Dicionários.....	69
Erros comuns ao trabalhar com dados estruturados.....	75

Exercício proposto	78
Exercícios Complementares.....	78
Tratamento de Erros em Python.....	81
Try Except	81
Else no tratamento de exceções.....	82
Finally	82
Tratamento de erro por tipo.....	83
Funções	87
Definindo Funções	87
Parâmetros em Funções.....	88
Exercício proposto	95
Exercícios Complementares.....	97
Funções Lambda	99
Criando uma função Lambda.....	99
Função Map.....	100
Filter	101
Reduce.....	103
Funções All e Any	104
Trabalhando com Módulos	107
O gerenciador de pacotes Pip	107
Ambientes Virtuais.....	109
Trabalhando com Arquivos.....	112
Gerenciamento de Arquivos CSV	112
Gerenciamento de Arquivos XLSX	115
Gerenciamento de Arquivos DOCX.....	118
Navegação e Manipulação no Sistema de Arquivos	124
Gerenciamento de Variáveis de ambiente e Arquivos	124
Exercício proposto	127
Exercícios complementares.....	127
Orientação a Objetos.....	128
Classes.....	128
Métodos de objetos.....	130
Alteração de propriedades em objetos.....	131
Herança.....	133
Herança de métodos.....	136
Reescrita de métodos.....	137

Os métodos mágicos	139
Modularização e Reuso	143
Testes em Python	147
Exercício Proposto	150
Exercícios Complementares	151
Generators	154
Definição de Generators	154
Generators Expressions	155
Uso de generators	156
List Comprehensions	159
Compressões de Dicionários (Dict Comprehensions)	162
Compressão de Sets (Set Compressions)	165
Exercício Proposto	166
Exercícios complementares	166
Anaconda e Jupyter	168
Introdução ao Anaconda e Jupyter	168
Instalação do Anaconda	168
Iniciando o Jupyter Notebook	173
Criando um Notebook	177
Conjuntos ou sets	179
Criando um conjunto	179
Operações com conjuntos	180
Alguns métodos para conjuntos	182
Geração de Mapas	184
Latitude e Longitude	184
Dimensões	184
Zoom	185
Tiles ou Mapas	185
Marcadores	186
Camadas	190
HeatMap	194
Conceitos básicos de HTML	196
Django – Python para Web	201
Criando um projeto Django	201
Iniciando o servidor	201
Ciclo básico do Django	203

Criando uma App.....	203
Exercício Proposto.....	207

CAPÍTULO 1

Introdução

O INÍCIO

Criada no fim da década 1980 por Guido van Rossum, Python hoje é um fenômeno mundial. Utilizada em praticamente todas as plataformas e dispositivos, e muito visada na análise de dados e aprendizado de máquina.

Para entender mais sobre esta linguagem e compreender seu sucesso é necessário entender a sua história primeiro. Rossum trabalhava no CWI (National Institute for Scientific Research) ou como é conhecido “Instituto de Pesquisa Nacional para Matemática e Ciência da Computação” com o desenvolvimento da linguagem ABC (da qual ele não era o criador). Porém, no Natal de 1989, Rossum precisou resolver um problema de programação, e percebeu que se escrevesse em C iria levar muito tempo. Foi então que em pleno Natal Rossum decidiu criar uma linguagem que fosse poderosa e ao mesmo tempo fácil de usar. Neste momento em diante surge o Python. O nome Python veio da série de TV Monty Python da qual a equipe que trabalhou na linguagem era fã.

Rossum trabalhou de forma privada na linguagem até 1991 quando lançou a versão 0.9.0. Nesta primeira versão era possível a criação de estruturas como Classes, Funções e já possuía tipos de dados nativos como Listas, Dicionários e Strings, além do tratamento de Exceções, característica pensada para uma futura aplicação em dispositivos móveis.

Apenas em 1994 a versão 1.0 do Python foi lançada. Como principal novidade desta versão está a inclusão das funções lambda, map, filter e reduce.

Após o lançamento da versão 1.2 em 1995, Guido se mudou para Virgínia onde começou a trabalhar para a Corporation for National Research Initiatives, um órgão sem fins lucrativos responsável por grandes feitos como o algoritmo que busca, verifica e proporciona a venda de direitos autorais na internet.

Enquanto trabalhava para a CNRI, Rossum lançou uma iniciativa para a popularização da programação chamado CP4E (Computer Programming for Everyone) “Programação de Computadores para todos” utilizando a linguagem Python como principal ferramenta.

Em 2000, o time de desenvolvimento da linguagem se mudou para a BeOpen a fim de formar o time PythonLabs. A CNRI pediu que a versão 1.6 fosse lançada para marcar o fim de desenvolvimento da linguagem naquele local.

Houve apenas um único lançamento na BeOpen: o Python 2.0. Após o lançamento o grupo de desenvolvedores da PythonLabs agrupou-se na Digital Creations.

O Python 2.0 implementou list comprehension, uma importante funcionalidade que permite unir e manipular duas listas.

A versão 2.0 era completamente Open Source (Código Aberto) além de ter sido movido para o SourceForge tornando muito mais acessível para download e correção de bugs.

A versão 2.1 mudou-se para o Zope (um servidor de aplicações web de código aberto escrito na linguagem Python).

Um importante passo tomado aqui foi a mudança de sua licença de uso que foi renomeada para Python Software Foundation License. Com isso, todo código, documentação e especificações desde o lançamento da versão alfa da 2.1 é de propriedade da Python Software Foundation (PSF), uma organização sem fins lucrativos fundada em 2001. Isso fez com que muitas pessoas se tornassem adeptas da linguagem, fazendo correções e implementações além de impulsionar a criação de módulos e do código livre em si.

Em 2008, foi lançado o Python 3. Esta versão é focada em remoção de códigos duplicados, e melhorias estruturais que tornam ela incompatível com códigos das versões 2.x.

Por ser de fácil compreensão, relativamente rápida e com tantos adeptos de código livre, Python possui hoje inúmeras aplicações.

Existem, por exemplo, motores (engines) de jogos que permitem o uso de Python como linguagem de script, como por exemplo o Unreal, a Unity e a Blender. Para este fim, o Python ainda dispõe de sua própria engine criada em software livre chamada PyGame.

Existem jogos de muito sucesso que foram criados em Python, como Civilization IV e Battlefield 2.

Python também é usado em modelagem e animação 3D em softwares como o já citado Blender, o GIMP e até mesmo em aplicações mais conservadoras como o AutoCad e o Maia.

Porém, o principal campo que a linguagem ganhou notoriedade é o de Machine Learning (Aprendizagem de Máquina). Existem diversos pacotes que facilitam as técnicas para o desenvolvimento de inteligências capazes de por exemplo fazer o reconhecimento facial e o reconhecimento de voz.

Uma outra aplicação muito poderosa é a programação de páginas WEB com Python. Existem Frameworks que proporcionam uma produtividade muito alta como o Flask, o Django e o Pyramid.

A linguagem é tão querida que é usada até mesmo em automação e em robótica como por exemplo no RaspBerry Pi, uma placa de circuito semelhante ao Arduino que recebe o sobrenome “Pi” em homenagem ao Python. Esta poderosa placa geralmente roda uma distribuição Linux e incentiva o uso de Python no seu desenvolvimento.

Com todas as vantagens e aplicações não é nenhuma surpresa que empresas como Google, AutoDesk, Microsoft, Amazon, Globo e a Nasa façam proveito desta linguagem em suas aplicações.

No caso do Google, por exemplo, os mecanismos de busca são construídos em Python. Outro caso famoso de uso é o Youtube que é criado com o framework Django.

Na AutoDesk, os softwares Maia e SoftImage possuem suporte oficial da linguagem. O AutoCAD pode ter rotinas implementadas na linguagem também.

Os sistemas do Globo são intimamente ligados com Python e os seus sites são criados no framework Django também.

Sistemas como BitTorrent são um outro exemplo de sucesso do uso de Python.

Talvez a os usos mais severos e incríveis são da Nasa, que inclusive, mostrou a primeira foto de um buraco negro usando como principal linguagem o Python.

Até mesmo Star Wars usa Python pois a empresa Industrial Light and Magic que é responsável por renderizar as imagens do filme, usa Python.

Com tantos exemplos é inevitável que Python se mostre uma linguagem muito promissora e encante tantos programadores. Esperamos que com esta introdução você se sinta inspirado a aprendê-la e a tornar-se parte deste mundo.

CAPÍTULO 2

Preparando o Ambiente

INSTALAÇÃO DO PYTHON

Para iniciar os estudos em Python, inicialmente é necessário baixar e instalar o interpretador Python. Um interpretador é responsável por compreender o código criado e transmitir esta mensagem de uma forma que o sistema operacional entenda.

O primeiro passo é fazer o download do arquivo do interpretador acessando o site <https://www.python.org/downloads/> e selecionar a versão mais atual do Python, no exemplo Python 3.8.0.

Looking for a specific release?

Python releases by version number:

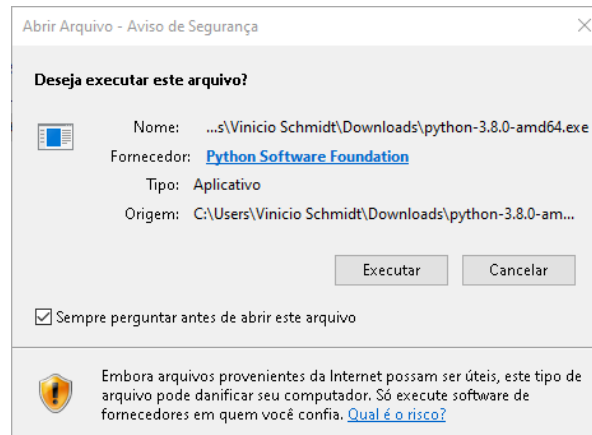
Release version	Release date		Click for more
Python 2.7.17	Oct. 19, 2019	Download	Release Notes
Python 3.7.5	Oct. 15, 2019	Download	Release Notes
Python 3.8.0	Oct. 14, 2019	Download	Release Notes
Python 3.7.4	July 8, 2019	Download	Release Notes
Python 3.6.9	July 2, 2019	Download	Release Notes
Python 3.7.3	March 25, 2019	Download	Release Notes
Python 3.4.10	March 18, 2019	Download	Release Notes
Python 3.5.7	March 18, 2019	Download	Release Notes

[View older releases](#)

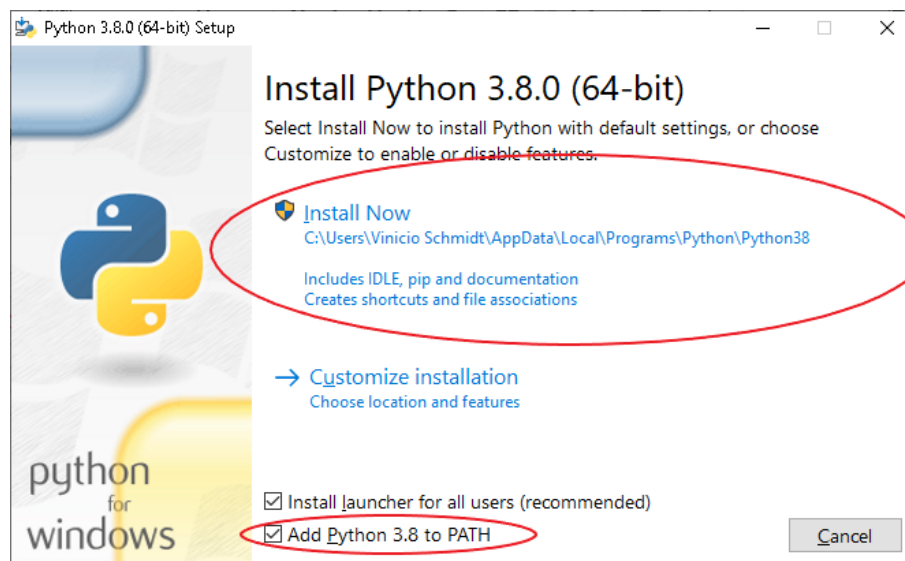
Rolando até o fim da página, é possível fazer o download do arquivo Windowsx86-64-executable-installer clicando sobre ele.

Files					
Version	Operating System	Description	MD5 Sum	File Size	GPG
Gzipped source tarball	Source release		e18a9d1a0a6d858b9787e03fc6daa20	23949883	SIG
XZ compressed source tarball	Source release		dbac8df9d8b9edc678d0f4cacdb7dbb0	17829824	SIG
macOS 64-bit installer	Mac OS X	for OS X 10.9 and later	f5f9ae9f416170c6355cab7256bb75b5	29005746	SIG
Windows help file	Windows		1c33359821033ddb3353c8e5b6e7e003	8457529	SIG
Windows x86-64 embeddable zip file	Windows	for AMD64/EM64T/x64	99cca948512b53fb165084787143ef19	8084795	SIG
Windows x86-64 executable installer	Windows	for AMD64/EM64T/x64	29ea87f24c32f5e924b7d63f8a08ee8d	27505064	SIG
Windows x86-64 web-based installer	Windows	for AMD64/EM64T/x64	f93f7ba8cd48066c59827752e531924b	1363336	SIG
Windows x86 embeddable zip file	Windows		2ec3abf05f3f1046e0dbd1ca5c74ce88	7213298	SIG
Windows x86 executable installer	Windows		412a649d36626d33b8ca5593cf18318c	26406312	SIG
Windows x86 web-based installer	Windows		50d484ff0b08722b3cf51f9305f49fdc	1325368	SIG

Após o download terminar abra o arquivo baixado. Uma permissão de administrador é solicitada. Clique em “Executar”.

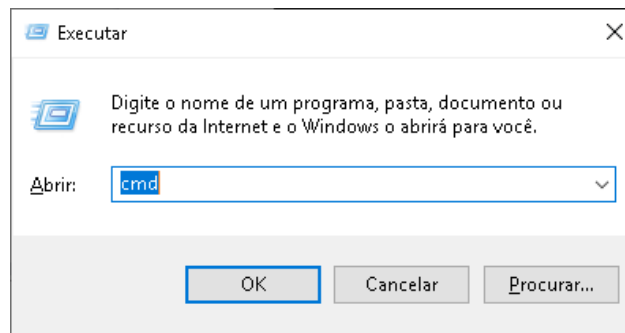


Na interface do assistente de instalação marque a opção Add Python 3.8 to PATH e clique em Install Now.



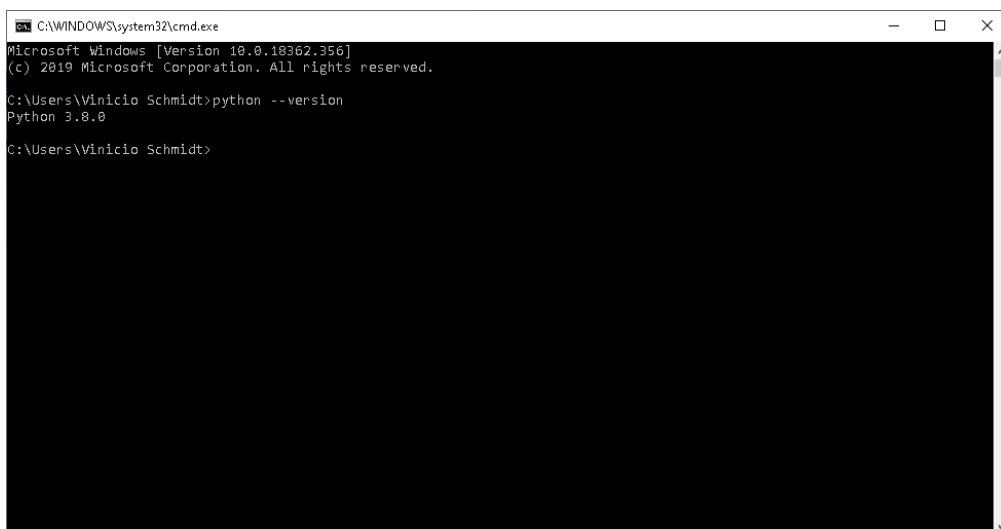
UTILIZAÇÃO DO TERMINAL

Ao concluir a instalação é necessário verificar se o Python foi efetivamente instalado e está disponível no sistema. Para isso, basta acessar o terminal do Windows. A forma mais rápida é usando as teclas “win”+“r”, digitando “cmd” e pressionando “Enter”. Também é possível



buscar por “Prompt de Comando” nos aplicativos.

Com o terminal aberto, é só digitar “python --version” e pressionar “Enter”. Com este comando é possível ver qual é a versão do Python que está instalada, ela deve iniciar com 3.



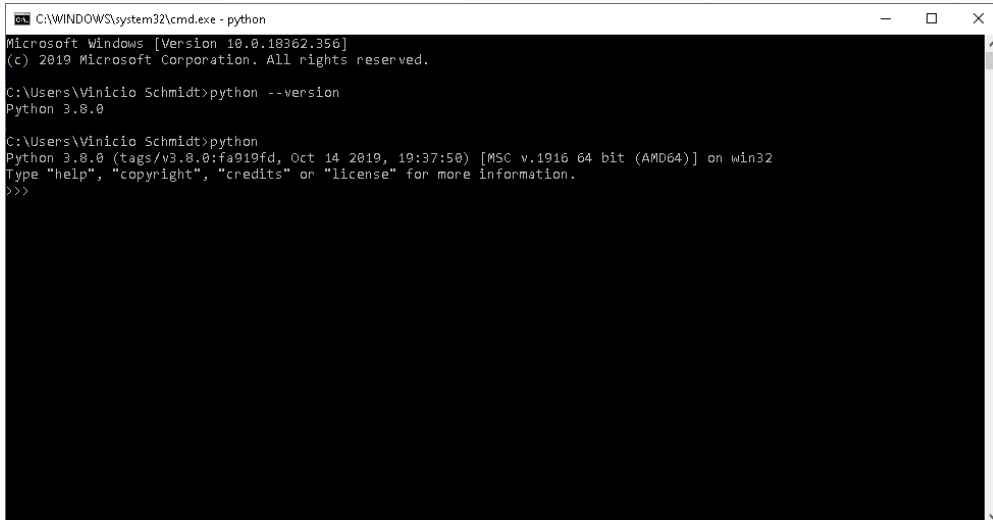
Por padrão é interessante checar ainda se o Python está em suas variáveis de ambiente, uma variável de ambiente é como se fosse um apelido para um comando. Para testar basta escrever “python” no terminal.

Se tudo correu bem, agora você está em um terminal Python.

Experimente um comando escrevendo:

```
print("Olá Mundo")
```

A função `print()` vem do inglês e significa "imprimir" e como resposta deste comando, o terminal escreveu "Olá Mundo".



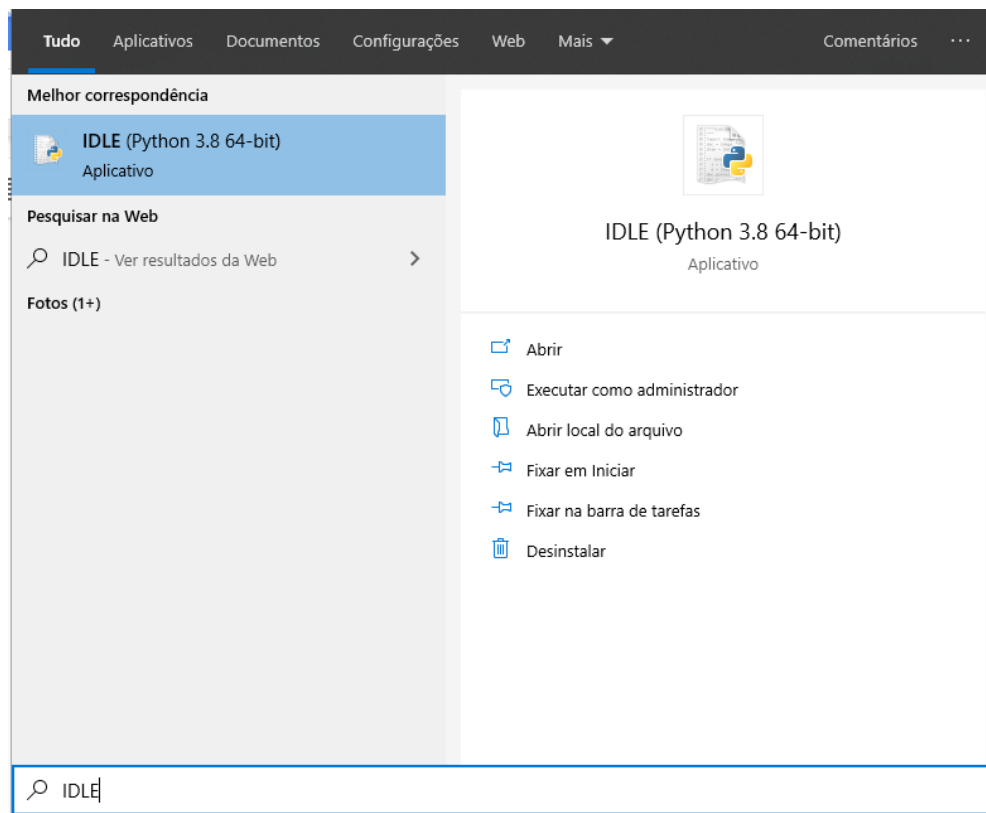
```
C:\WINDOWS\system32\cmd.exe - python
Microsoft Windows [Version 10.0.18362.356]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\Vinicio Schmidt>python --version
Python 3.8.0

C:\Users\Vinicio Schmidt>python
Python 3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:37:50) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

IDLE DO PYTHON

Programar no terminal é um pouco difícil, e muito trabalhoso. Porém, ao instalar o Python um aplicativo chamado de “IDLE” vem instalado por padrão. O IDLE é uma IDE (Integrated Development Environment) do português Ambiente de Desenvolvimento Integrado, nela é possível escrever códigos com uma facilidade maior além de poder salvar, editar reproduzir os códigos. Para fazer uso desta aplicação é necessário abri-lo buscando por seu nome nos aplicativos.




Ao abrir o aplicativo, um terminal chamado “Shell” é aberto. Ele se comporta de forma muito semelhante ao terminal inicializado no Windows após digitar python. A fim de fazer um teste, digite o código:

```
print("Olá Mundo")
```

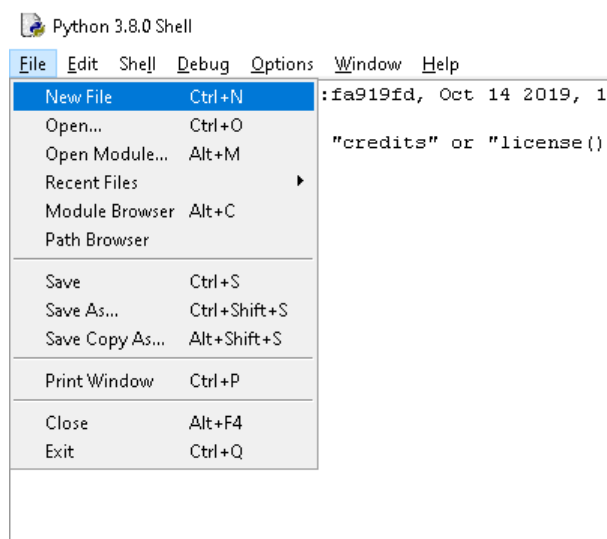
e pressione “Enter”.

Assim como no terminal uma mensagem é impressa.



```
Python 3.8.0 Shell
File Edit Shell Debug Options Window Help
Python 3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:37:50) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> print("Olá mundo")
Olá mundo
>>> |
```

Mas e se um algoritmo possuir mais de uma linha? E se for necessário salvá-lo? É possível fazer isso utilizando o editor de códigos da IDLE do Python. O primeiro passo é criar um novo arquivo que vai conter as linhas de código pelo menu "File">"New File" ou pelo atalho "Ctrl"+"N".

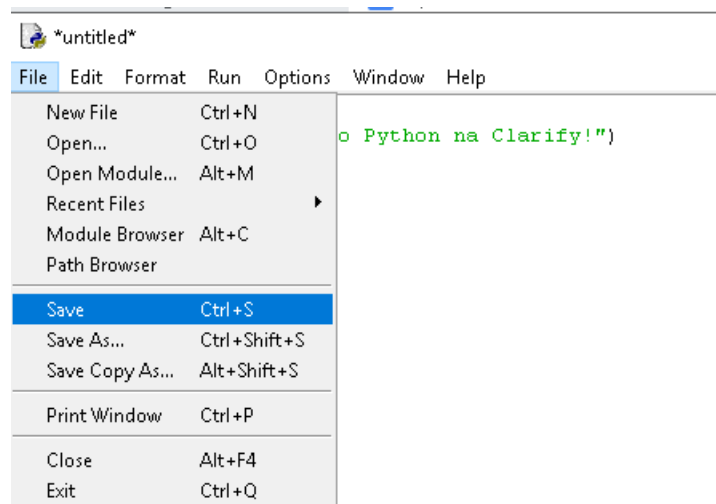


Neste arquivo ficaram todas as instruções que serão dadas ao computador.

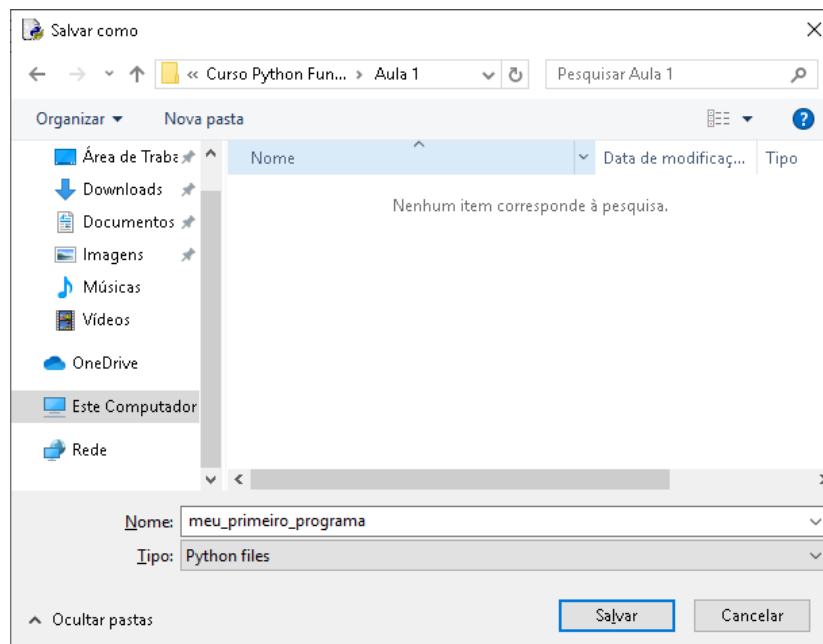
Por exemplo, é desejado escrever duas linhas de código, uma que diga ao computador para escrever “Olá mundo” e outra para dizer “Eu estou aprendendo Python na Clarify!”. esta tarefa pode ser realizada usando novamente a função print como no seguinte código:

```
print("Olá mundo")  
print("Eu estou aprendendo Python na Clarify!")
```

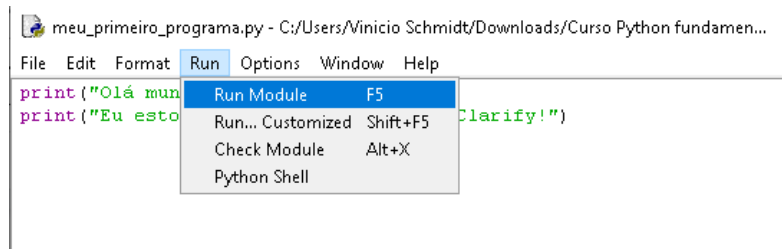
Antes de executar este código é necessário salvar ele em algum lugar com menu “File”>“Save” ou usando o comando “Ctrl”+“S”.



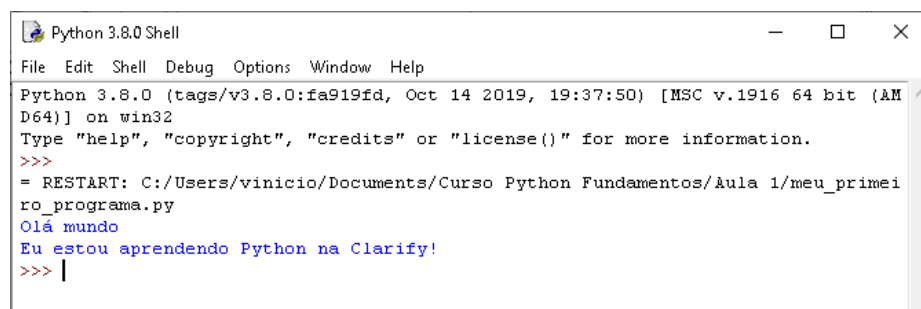
Para salvar, basta apenas navegar até a pasta desejada, dar o nome para o arquivo, neste exemplo “meu_primeiro_programa” e clicar em Salvar.



Agora, com o arquivo salvo, é possível executá-lo no menu "Run" ou usando o atalho F5.



Ao executar o código pode-se observar que o computador executou todos os comandos no Shell.

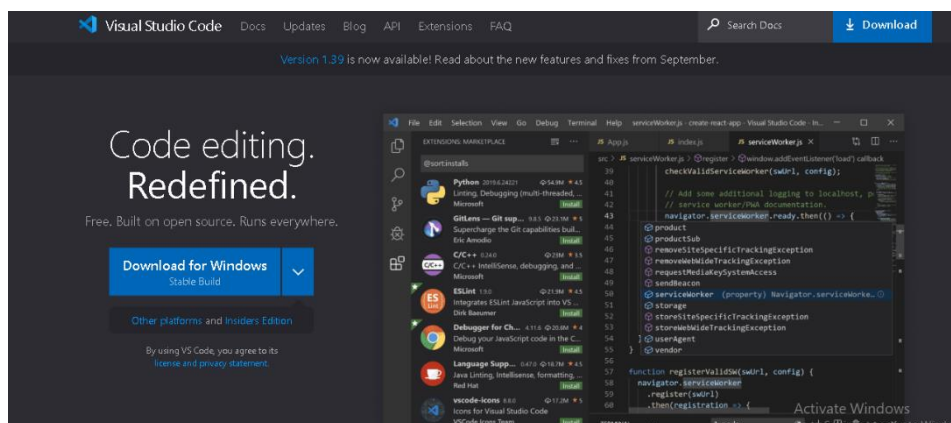


VSCODE

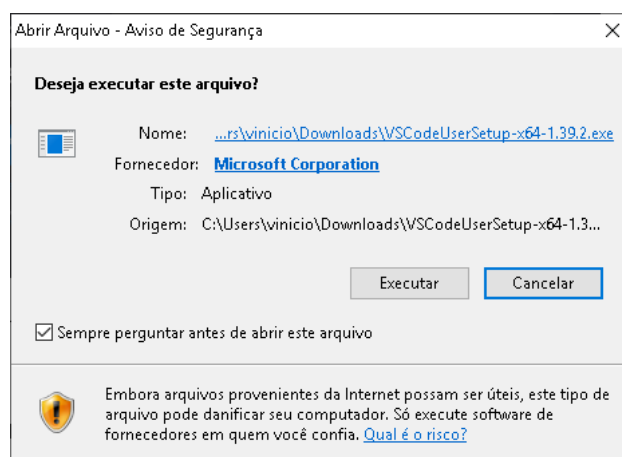
A IDE do Python é bem limitada em recursos e se torna cansativa com o passar do tempo por ter um fundo branco que machuca os olhos. Para resolver isso, é possível instalar o VSCode, uma IDE profissional, gratuita e com recursos muito abrangentes.

INSTALAÇÃO

Para iniciar é necessário baixar o VSCode acessando o site: <https://code.visualstudio.com> e clicando em “Download for Windows”

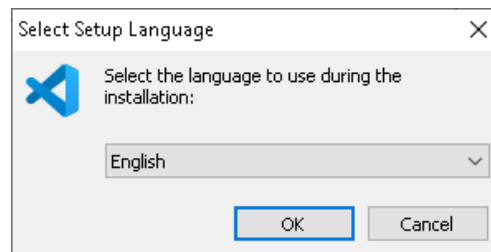


Após o download ter terminado, é factível a instalação abrindo o arquivo. Uma permissão de administrador será solicitada:

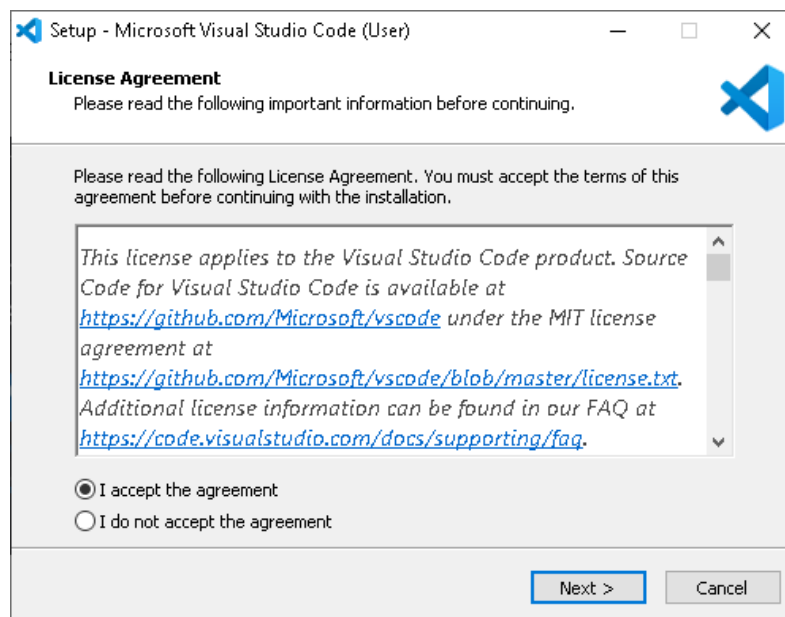


Basta clicar em “**Executar**”.

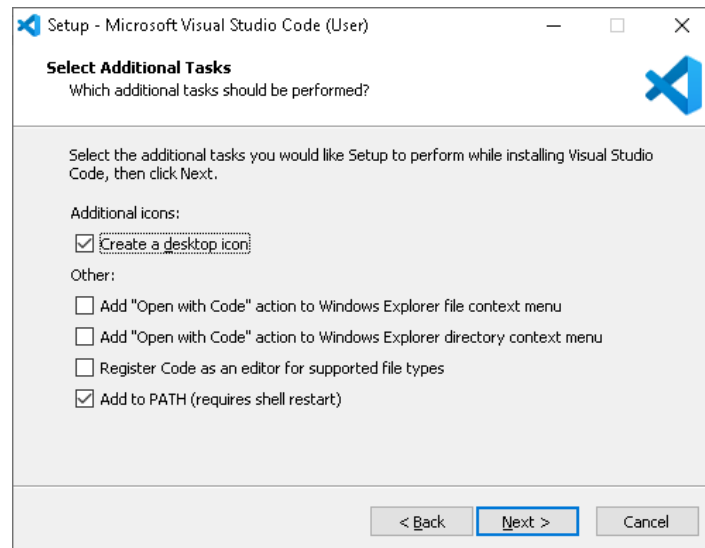
Será solicitada a linguagem, por hora, deve-se selecionar inglês (English).



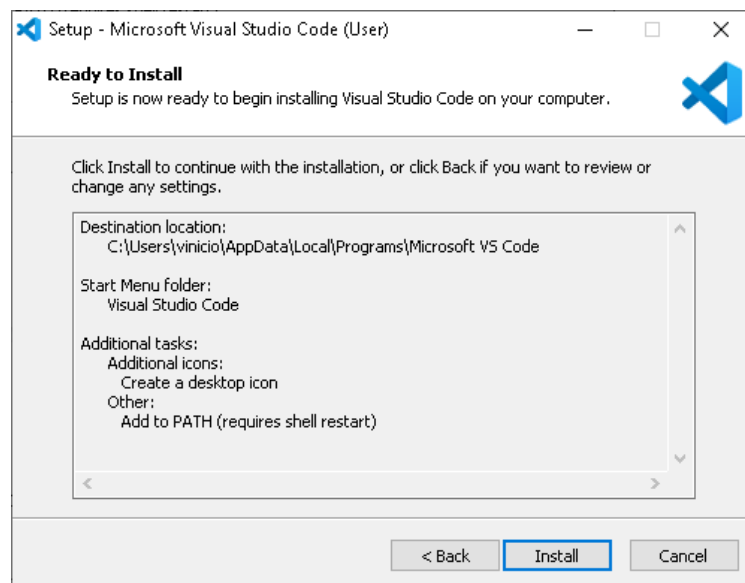
Na primeira tela, basta aceitar os termos de contrato e clicar em "Next".



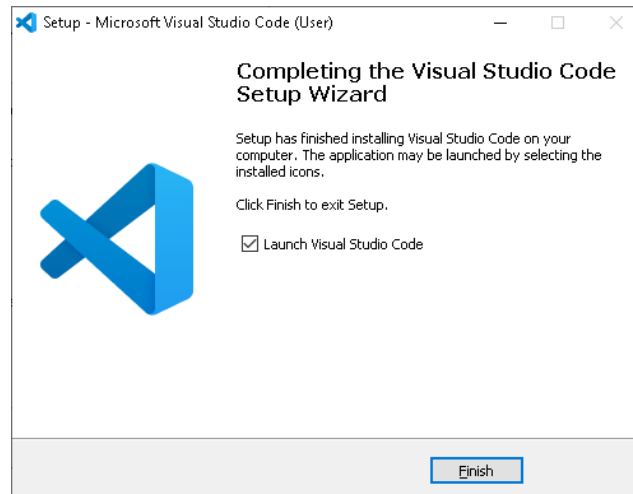
Marque as opções "**Create a desktop icon**" se desejar criar um atalho na área de trabalho e "**Add to PATH**". Adicionar ao PATH significa, como no Python, criar uma variável de ambiente e poder chamar o programa com um apelido no terminal.



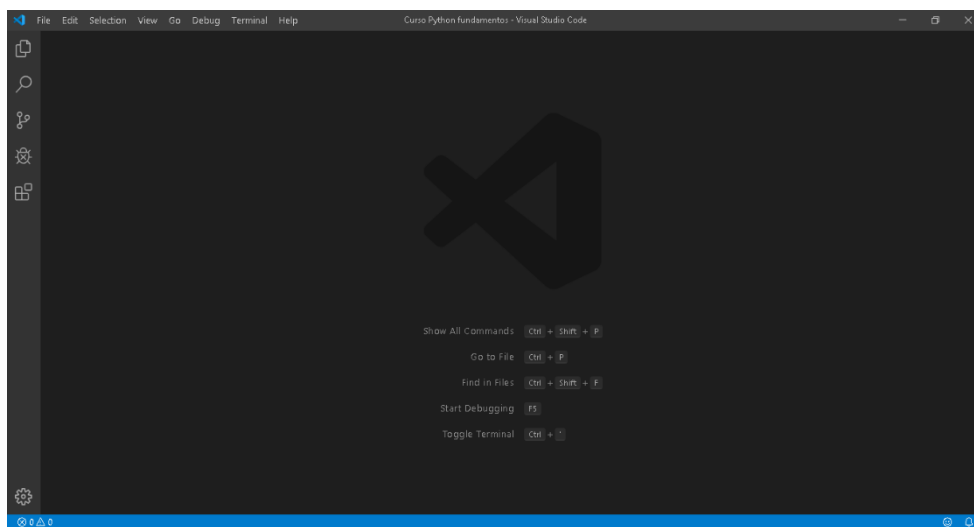
Por fim, basta pressionar “**Install**”.



Uma tela de carregamento será exibida e após os arquivos serem instalados a janela de sucesso será exibida.

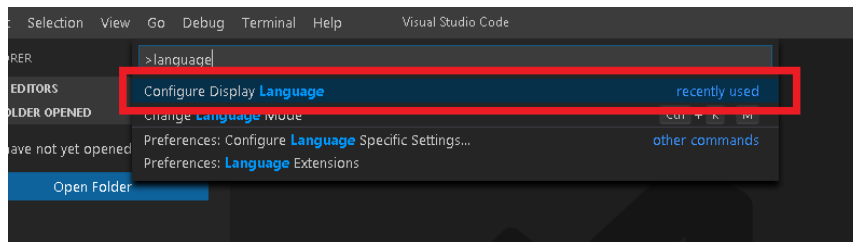


Se a instalação procedeu conforme esperado basta buscar o VSCode em seus aplicativos e abri-lo.

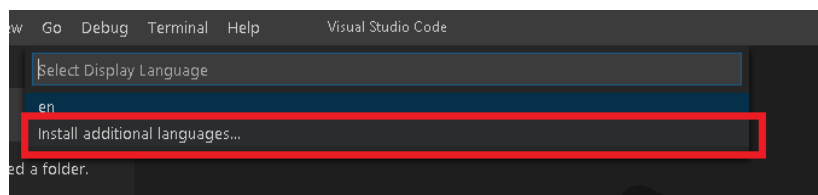


INSTALAÇÃO DE IDIOMA

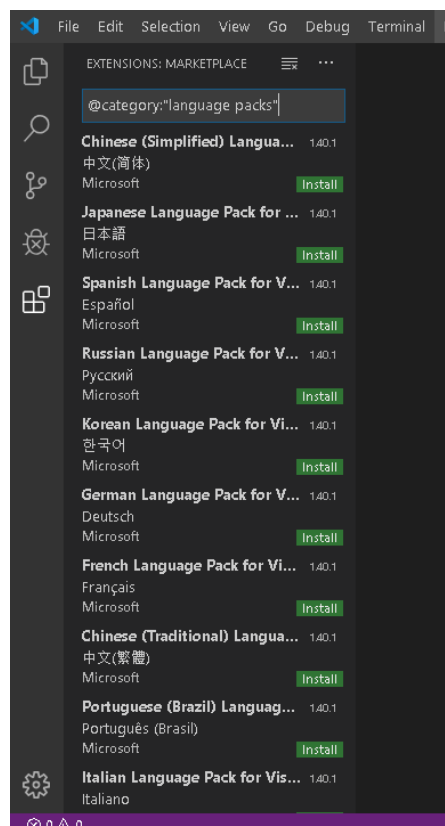
Para tornar o VS Code mais amigável, é interessante instalar um pacote do idioma português. Para isso, pode-se usar o comando "Ctrl+Shift+P" e digitar "language". Nas opções que são observadas na imagem abaixo selecionar "Configure Display Language":



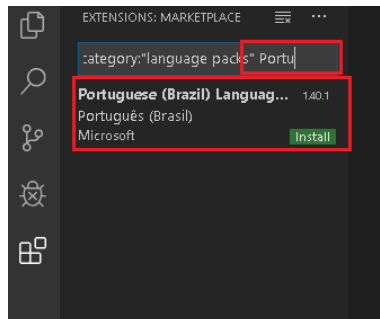
Uma nova caixa de opções será mostrada. Clicar na opção “Install additional languages...”:



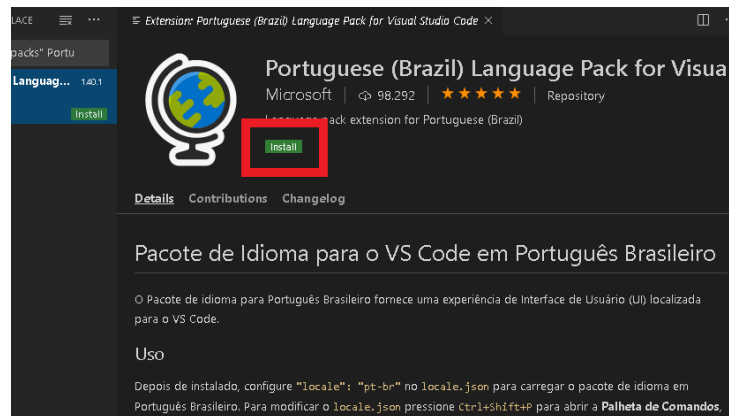
Um menu lateral irá surgir. Este é o menu de extensões, nele é possível instalar configurações personalizadas para utilizar e tornar a vida do programador mais fácil.



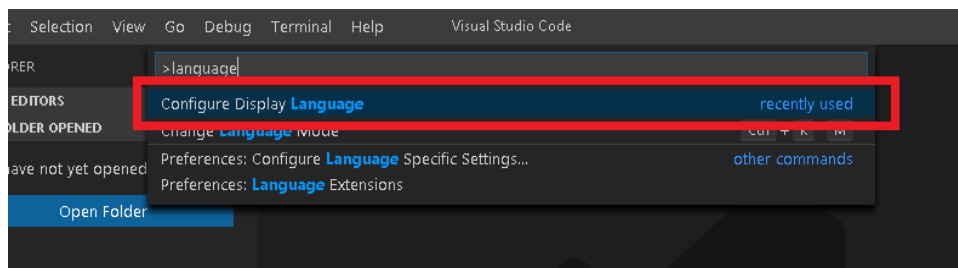
Neste menu, digite "Portu" e selecione o pacote de linguagem correspondente como na imagem abaixo.



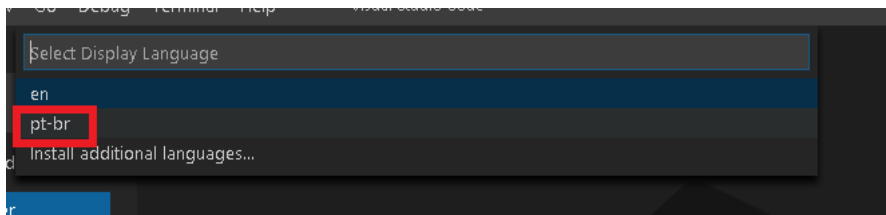
Basta agora clicar em "Install".



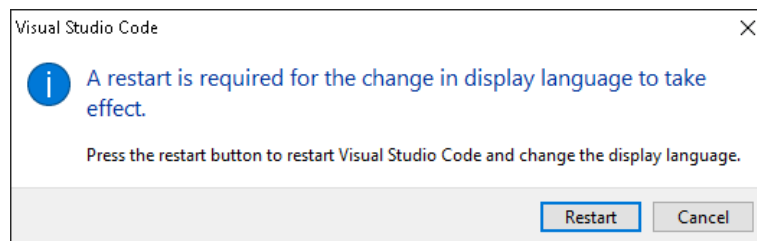
Após instalado o programa deve ser fechado e aberto novamente. Agora, o comando "Ctrl+Shift+P" deve ser usado novamente e o menu "Configure Display Language" deve ser selecionado.



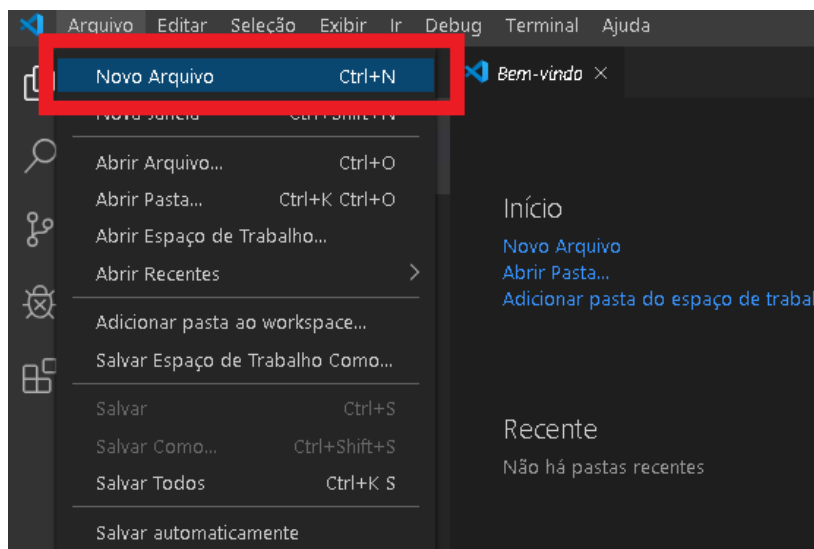
A última etapa desta configuração é selecionar a opção “**pt-br**”:



Um alerta indicando que a IDE precisa ser novamente reiniciada é exibido. Basta clicar em **Restart**.

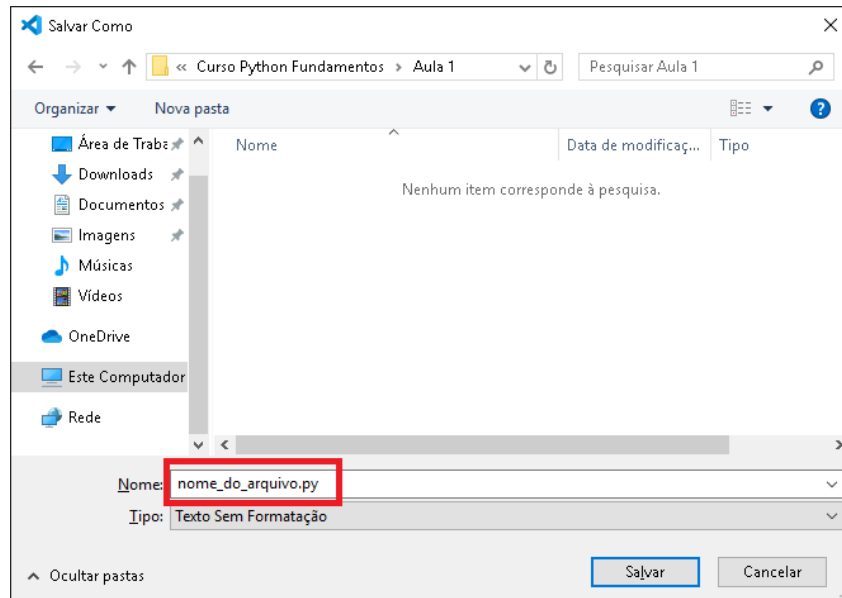


Com o pacote de tradução instalado é hora de criar um arquivo. De forma semelhante à IDLE do Python, basta clicar no menu “**Arquivo**”>”**Novo Arquivo**” ou usar o atalho “**Ctrl+N**”.

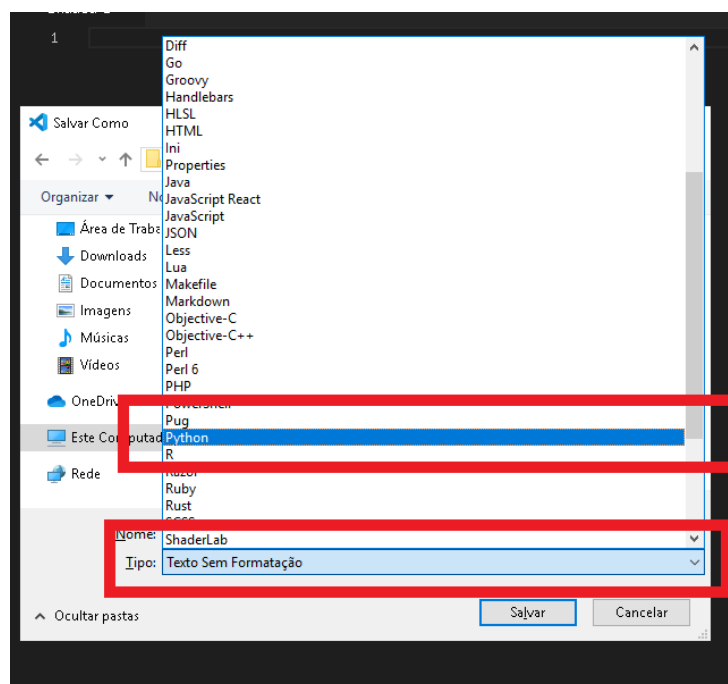


Por padrão, o primeiro passo é salvar o arquivo no menu “**Arquivo**”>”**Salvar**” ou pelo atalho “**Ctrl+S**”. Uma sutil diferença terá que ser observada em relação a IDLE do Python.

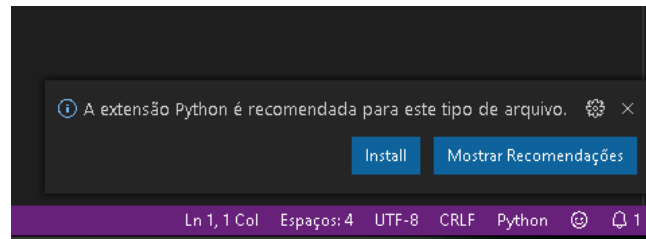
Ao salvar um arquivo no VSCode deve-se adicionar a sua extensão “.py” ao fim do arquivo para definir um arquivo Python.



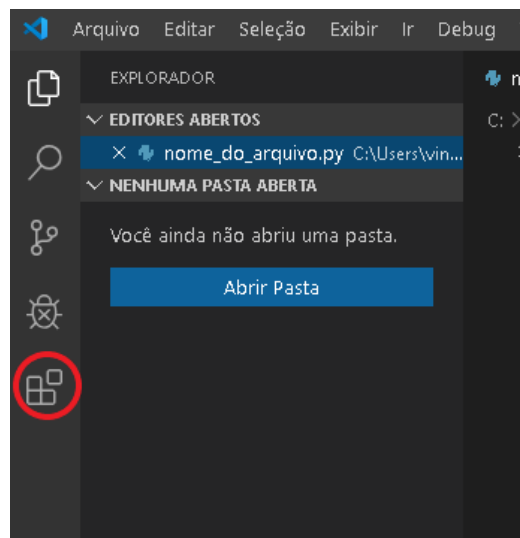
Também é permitido usar o menu de seleção do tipo de arquivo como visto abaixo:



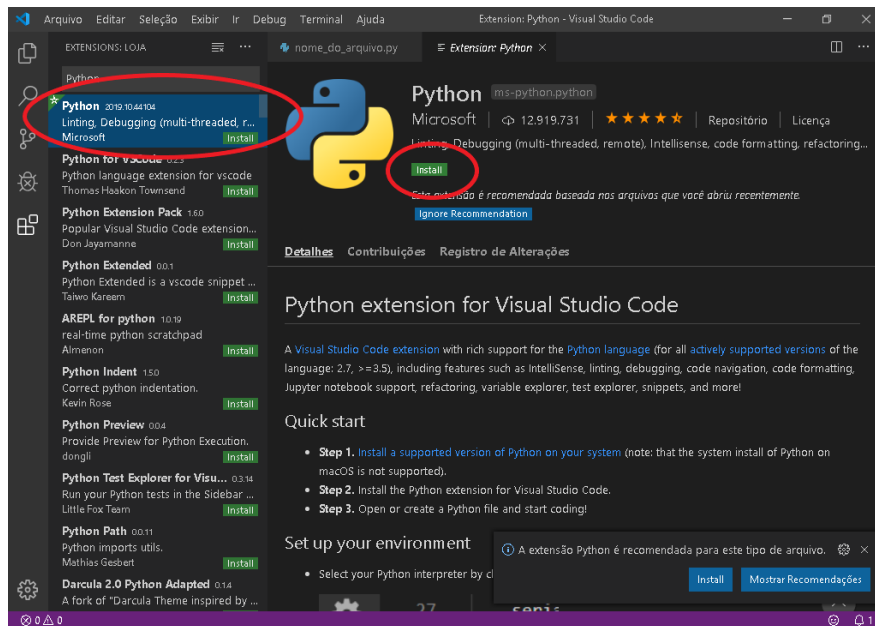
Ao salvar o primeiro arquivo Python uma janela no canto inferior direito é aberta. Ela mostra que não existe um interpretador Python instalado na IDE.



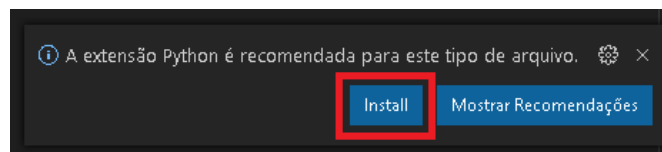
Para instalar o interpretador, é viável usar o menu de extensões no canto esquerdo.



Digitando “**Python**” no campo de pesquisa surgem diversas opções de interpretador. Deve-se selecionar a primeira e instalar clicando no botão “**Install**”.



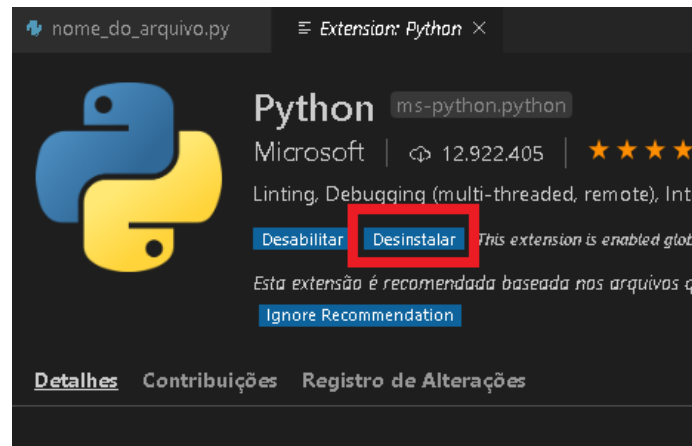
Alternativamente, basta clicar em **"Install"** na caixa que apareceu anteriormente:



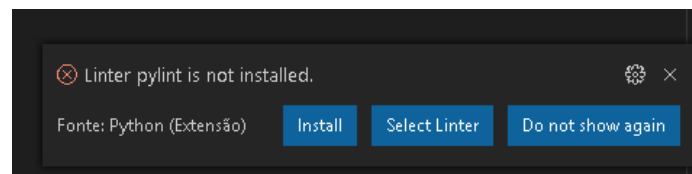
Se a instalação correr bem ele irá mostrar o status de instalado e ao voltar para o arquivo, será exibido no canto inferior esquerdo a versão do Python usada para interpretar.



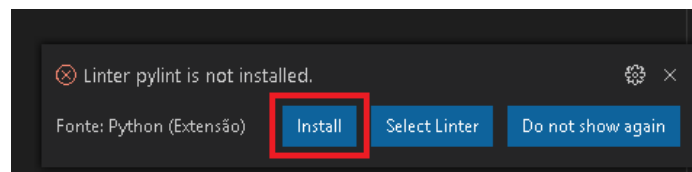
De forma análoga, para desinstalar basta acessar a aba de extensões e clicar em desinstalar:



Uma outra extensão muito importante é sugerida no canto inferior direito:



Trata-se de um validador de sintaxe. Este validador é importante pois corrige erros que rotineiramente se comete, de forma semelhante ao corretor de um editor de texto como o Word. Para instalar basta clicar em “**Install**”.



Com esta extensão instalada, ao cometer um erro, semelhante a um programa de edição de texto, a IDE mostra de forma sublinhada o trecho que ele entende que está errado.

Por exemplo, ao remover um parêntese que não poderia ser removido a palavra seguinte é grifada de vermelho para denotar o erro.

```
nome_do_arquivo.py X
C: > Users > Vinicio Schmidt > Desktop > nome_do_arquivo.py
1 print("Olá Mundo")
2 print("Estou aprendendo Python na Clarify")
3
4
5
6
7
8
```

Para executar o programa (após fazer a correção anteriormente feita) pode-se clicar com o botão direito do mouse e selecionar a opção "Executar arquivo no Terminal"

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
PS C:\Users\Vinicio Schmidt> & "C:/Users/Vinicio Schmidt/AppData/Local/Programs/Python/Python38/python.exe" "C:/Users/Vinicio Schmidt/Desktop/nome_do_arquivo.py"
Olá Mundo
Estou aprendendo Python na Clarify
PS C:\Users\Vinicio Schmidt>
```

Uma interface é aberta na parte inferior da tela e o resultado do algoritmo criado é executado:

As configurações iniciais foram feitas, agora o trabalho pode se iniciar.

```
print("Olá mundo")
print("Eu estou aprendendo Python na Clarify!")
```

Ir para Definição	F12
Inspecionar definição	Alt+F12
Find All References	Shift+Alt+F12
Inspecionar Referências	Shift+F12
Renomear Símbolo	F2
Alterar todas as ocorrências	Ctrl+F2
Formatar Documento	Shift+Alt+F
Formatar documento com...	
Ação de Origem...	
Recortar	Ctrl+X
Copiar	Ctrl+C
Colar	Ctrl+V
Executar Arquivo no Terminal	
Executar o Arquivo de Testes Unitários Atual	
Executar Seleção/Linha no Terminal	Shift+Enter
Run Current File in Python Interactive Window	
Ordenar Importações	
Paleta de comandos...	Ctrl+Shift+P

CAPÍTULO 3

Fundamentos da Linguagem Python

OPERADORES

Sem dúvidas um programa pode fazer muito mais que apenas imprimir dados na tela. As funções primordiais são realizáveis com o uso de operadores como por exemplo:

```
5 + 5
```

Como saída desta operação o Python nos devolve "10", que é o resultado da operação.

Podemos usar diversos operadores como o "-" para fazer uma subtração:

```
3 - 2
```

De forma semelhante é possível usar a multiplicação "*" e divisão "/":

```
2*2
```

```
6/2
```

O Python traz nativamente o recurso de exponenciação usando "**"):

```
2**4
```

A tabela abaixo traz os operadores nativamente disponíveis com algumas operações.

Operador	Operação	Exemplo	Resultado
**	Exponenciação	2**3	8
%	Módulo (Resto da divisão)	22%8	6
//	Parte inteira de uma divisão	22//8	2
/	Divisão	22/8	2.75
*	Multiplicação	3*5	15

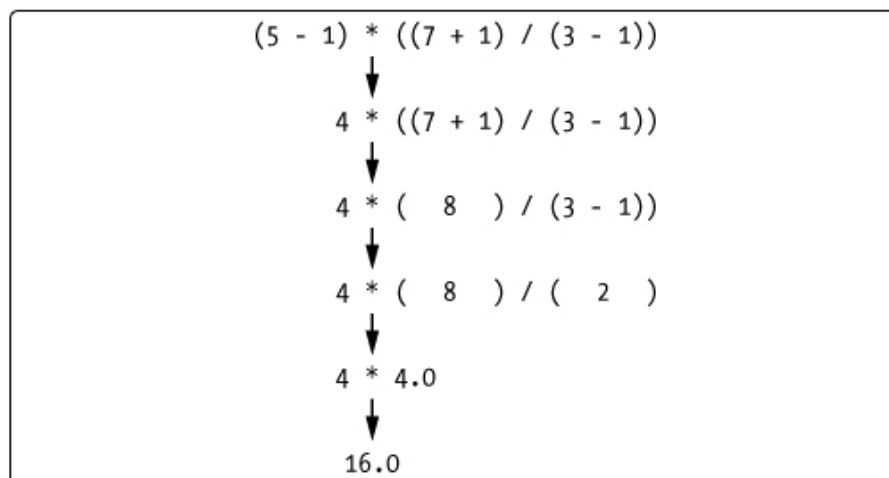
-	Subtração	5-2	3
+	Adição	2+2	4

É possível, de forma muito similar a matemática, executar expressões que contenham parênteses. Por exemplo:

```
(5 - 1) * ((7 + 1) / (3 - 1))
```

Este comando segue as mesmas normas de resolução da álgebra tradicional:

1. Resolve-se os parênteses
2. Resolve-se as multiplicações e divisões da esquerda para a direita
3. Resolve-se as somas e subtrações da esquerda para a direita



VARIÁVEIS

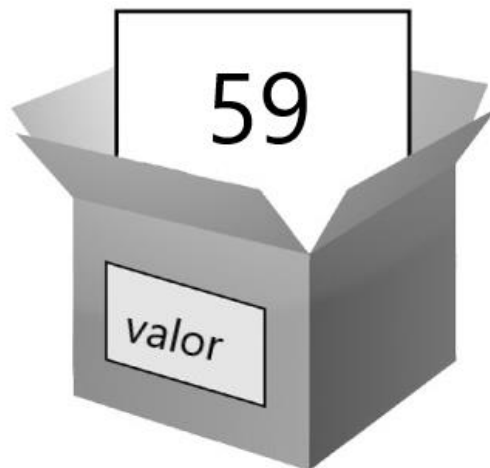
Na maioria dos programas é necessário armazenar o resultado de um cálculo em um lugar. Para isso se usa uma variável.

Uma variável é como se fosse uma caixa em que se coloca um objeto, ou no caso de um programa um valor.

Por exemplo, para adicionar o valor 59 dentro de uma variável chamada "valor" o código abaixo é praticável:

```
valor= 59  
print(valor)
```

Neste código, o valor 59 foi atribuído através do operador "=" a uma variável chamada "valor".



É factível usar operações em variáveis também, como por exemplo:

```
resultado=2+5  
print(resultado)
```

Desta forma, o cálculo 2+5 é armazenado na variável "**resultado**" e então é exibido usando a função **print**.

A variável resultado tem agora o valor de 7. Uma variável pode receber o valor de outra e ser usada em outro cálculo como por exemplo:

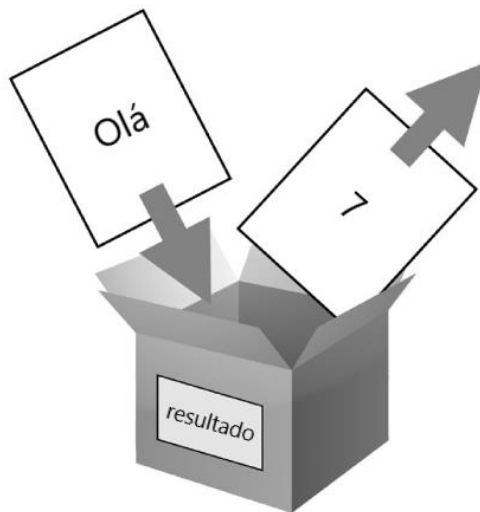

```
resultado=2+5  
resultado_final=resultado+3  
print(resultado_final)
```

Desta forma, a “**resultado_final**” terá o valor de “10”.

Uma variável pode ainda ter seu valor alterado, tanto de valor quando de tipo. Por exemplo:

```
resultado=2+5  
print(resultado)  
resultado="Olá"  
print(resultado)
```

Este exemplo se torna mais fácil se pensar que armazenamos um objeto em uma caixa e depois tiramos ele e colocamos outro:



As variáveis são fantásticas, porém, existem algumas regras para a nomeação das mesmas:

- Só pode usar uma palavra
- Pode usar letras, números e o caracter sublinhado “_”, ou como a maioria dos programadores chama “**underscore**”
- Não pode começar com um número

A tabela abaixo mostra alguns exemplos de nomeação:

Exemplo Válido	Exemplo Inválido
juros	juros-atual (Hífen não é um caractere válido)
poupancaTotal	poupanca total (Espaços não podem ser usados)
quarta_conta	4_conta (Não pode começar com números)
_valor	59 (Não pode ser um número)
SPAM	\$oma_tota (Não pode conter \$)
conta_4	'conta' (Não pode conter aspas)

Uma outra parte fundamental de um programa é a entrada de dados pelo usuário. Esta tarefa é atribuída a função **"input()"**.

Seu uso é bem simples, e requer apenas a frase da pergunta que se deseja fazer. Por exemplo:

```
nome_do_usuario = input("Qual é o seu nome? ")
print(nome_do_usuario)
```

Desta forma, a primeira linha faz a captura de uma entrada do usuário e após a captura imprime este dado na tela.

FUNÇÃO LEN()

Uma outra função muito interessante é a função **"len()"** com ela é possível verificar o tamanho de uma String. Integrando ela com o algoritmo anterior:

```
nome_do_usuario = input("Qual é o seu nome? ")
tamanho_do_nome = len(nome_do_usuario)
print(tamanho_do_nome)
```

Desta forma, após entrar com um nome, o algoritmo conta o número de letras do nome digitado.

TIPOS DE DADOS:

Até o momento verifica-se que as variáveis podem ter diversos tipos como:

- Nomes
- Números Inteiros
- Números fracionários
- Isso é verdadeiro ou falso

Quando se está falando de um nome, de um endereço ou qualquer "**Cadeia de caracteres**" como "**Jonas**", "**Rua das Laranjeiras**" ou "**AWQ1203**" está se falando de um tipo de dado conhecido no Python como "**String**".

Números inteiros positivos e negativos como **1**, **2**, **-7** e **-3** são tratados no Python como números do tipo "**int**".

Já números fracionários positivos e negativos como **1.2**, **2.75**, **-7.23** e **-3.54** são tratados no Python como números do tipo "**float**", também conhecidos como números de ponto flutuante.

Valores como "**Isso é verdade**", "**Isso é falso**", ou seja, de caráter lógico, são tratados como valores booleanos e recebem os valores de "**True**" para verdadeiro e "**False**" para falso.

FUNÇÃO TYPE()

Sempre que se deseja saber o tipo de uma variável ou de um valor é indicado o uso de uma função chamada **type()**. Para usá-la, basta passar a variável ou o valor que se deseja saber o tipo. Por exemplo:

```
type("Um texto qualquer")
variavel = 10
type(variavel)
```

Saber estes tipos evita erros ao tentar manipular dois tipos diferentes na mesma variável.

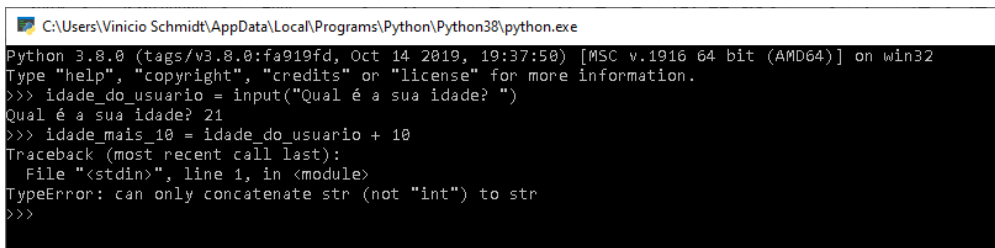
FUNÇÃO INT()

A função **input()** sempre retorna um dado do tipo **string**, às vezes se torna necessário converter este dado.

Por exemplo, se for desejado adicionar 10 anos na idade de em uma entrada do usuário, pode-se, equivocadamente usar um algoritmo parecido com esse:

```
idade_do_usuario = input("Qual é a sua idade? ")
idade_mais_10 = idade_do_usuario + 10
print(idade_mais_10)
```

Desta forma o Python irá apresentar um erro, pois o código está tentando somar uma **string** com um **número**.



```
C:\Users\Vinicio Schmidt\AppData\Local\Programs\Python\Python38\python.exe
Python 3.8.0 (tags/v3.8.0:fa019fd, Oct 14 2019, 19:37:50) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> idade_do_usuario = input("Qual é a sua idade? ")
Qual é a sua idade? 21
>>> idade_mais_10 = idade_do_usuario + 10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
>>>
```

Para fazer esta soma, é necessário converter a entrada do usuário para um número. Como a idade é um número inteiro a função **"int()"** pode ser usada. Implementando-a no código:

```
idade_do_usuario = input("Qual é a sua idade? ")
idade_mais_10 = int(idade_do_usuario) + 10
print(idade_mais_10)
```

Desta forma, o código converte a entrada para um número e então faz a soma com 10.

FUNÇÃO FLOAT():

Porém, e se fosse solicitado um dado de dinheiro? Como visto abaixo:

```
orcamento = input("Qual é o seu orçamento? ")
orcamento = float(orcamento) + 1500
print(orcamento)
```

Para estas conversões existe a função **float()**, ela faz a conversão de uma string em um número de ponto flutuante.

FUNÇÃO STR()

Em dado momento, pode ser necessário converter um número para texto, a função usada para esta tarefa é a **str()**, ela transforma números (decimais e inteiros) em **strings**:

```
orcamento = input("Qual é o seu orçamento? ")
orcamento = float(orcamento) + 1500
respota_usuario = "Seu orçamento é de: R$" + str(orcamento)
print(respota_usuario)
```

OPERADORES DE COMPARAÇÃO

Na maioria dos programas é necessário fazer checagens e expressões como **"isso é maior que isso?"**, **"O nome dele é igual a isso?"** ou **"o orçamento está acima de 1000?"** são comuns.

Para prover esta necessidade existem diversos comparadores no Python. A tabela abaixo traz os operadores de comparação:

Operador	Nome
==	Igual a
!=	Diferente de
<	Menor que
>	Maior que
<=	Menor ou igual a
>=	Maior ou igual a

Uma comparação sempre irá retornar um dado Booleano, ou seja, **"True"** (do inglês Verdadeiro) ou **"False"** (do inglês Falso). Por exemplo o nome **"Ana"** é igual ao nome **"Larissa"**?

```
"Ana" == "Larissa"
```

O número 2 é diferente do número 3?

```
2 != 3
```

O número 5 é maior que o número 4?

```
5 > 4
```

Todas estas comparações definem o fluxo que o programa vai seguir.

OPERADORES BOOLEANOS

Mas e se um programa precisar de mais de uma condição? Por exemplo, "O número 4 é maior que 2? Ele é menor que 5?". Para estes casos, existem os operadores Booleanos.

OPERADOR AND

Para o problema anterior existe a possibilidade do uso do **"and"** do inglês **"e"** como visto abaixo:

```
4>2 and 4<5
```

OPERADOR OR

```
4>2 or 4<5
```

É necessário supor também problemas como "A cor do carro é azul **ou** vermelha?".

O programa abaixo define uma variável contendo a cor de um carro e faz uma checagem para ver se ela é azul ou vermelha.

```
cor_do_carro = "azul"  
cor_do_carro=="azul" or cor_do_carro=="vermelha"
```

OPERADOR NOT

Um grande facilitador na tomada de decisões é o operador NOT. "O carro **não é** azul". O exemplo está negando uma comparação. Um algoritmo que exemplifica este caso está demonstrado abaixo:

```
cor_do_carro = "azul"  
not cor_do_carro=="azul"
```

CAPÍTULO 4

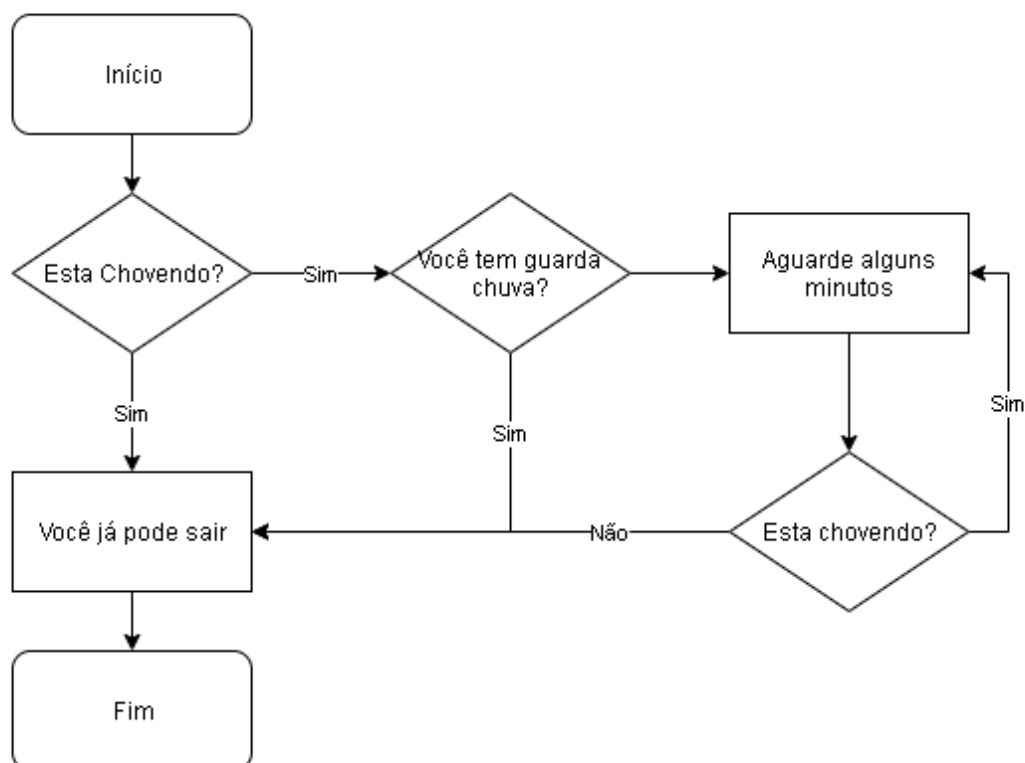
Estruras Condicionais

TOMADA DE DECISÕES

Uma outra tarefa extremamente importante de um algoritmo é a tomada de decisões. No dia a dia certamente já se ouviu “**se você fazer isso essa coisa vai ocorrer**” ou “**se isso não ocorrer faça aquilo**”.

Tomando um exemplo cotidiano em dias de chuva:

Este é o conceito de tomada de decisões, no Python, uma tomada de decisões é feita pelas estruturas “**if()**” (que vem do inglês **se**), “**else**” (do inglês **senão**) e “**elif()**” (acrônimo **de else if**, que significa “**senão se**”).



Uma estrutura, ou bloco é a forma com que o programa executa alguma função especial, em Python tudo que se deseja executar dentro de um bloco deve ser tabulado, ou seja, ter um espaço antes da declaração.

O BLOCO IF

Como anteriormente mencionado, o bloco **if** significa “**se**”. Intuitivamente usamos o “**se**” em nosso cotidiano como por exemplo “Ao ir no cinema, **se** o filme for ‘Malévola’ vamos assistir” ou “**se** estiver calor, ligue o ar condicionado”.

Intuitivamente, o bloco **if** faz uma comparação usando uma condição com operadores lógicos e booleanos. Abaixo segue um exemplo:

```
if(2>1):  
    print("Dois é realmente maior que um")
```

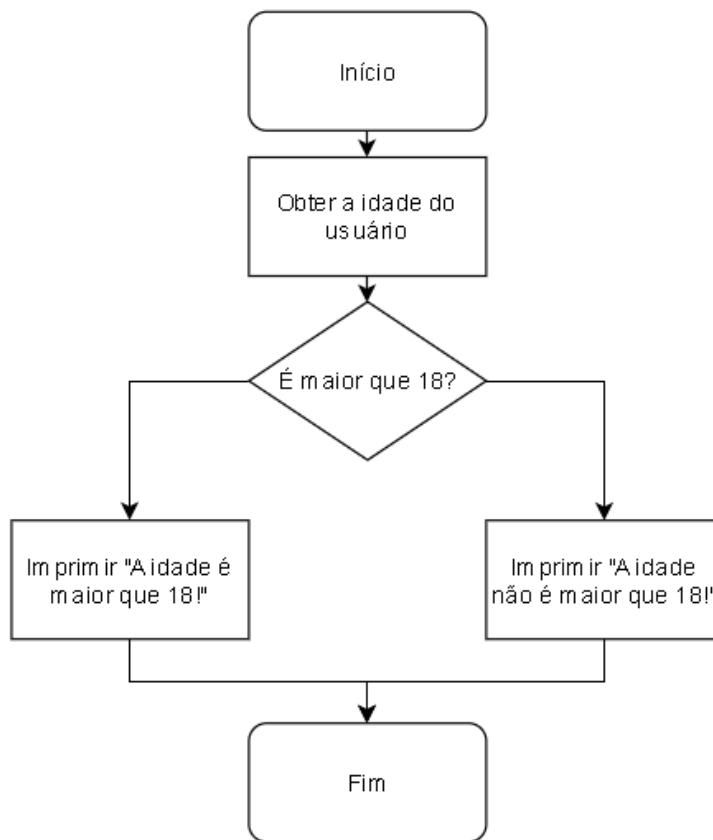
O BLOCO ELSE

As vezes é necessário que uma segunda ação seja tomada, “**se** tiver coca traga, **senão** traga pepsi”. O bloco “**else**” (**senão**) é usado para situações que é necessário um tratamento do fluxo caso a condição não seja atendida.

O código abaixo pega a idade do usuário e verifica se é maior que 18.

```
idade_do_usuario = input("Qual é a sua idade? ")  
if(int(idade_do_usuario)>18):  
    print("A idade é maior que 18!")  
else:  
    print("A idade não é maior que 18!")
```

Este programa pode ser visto em forma de um fluxograma:



Na segunda linha, o bloco **if()** fez a comparação da idade do usuário com o número 18 e **se** a idade for maior, ele irá imprimir "A idade é maior que 18!", **senão** o fluxo entra na instrução **else** e irá imprimir "A idade não é maior que 18!".

O BLOCO ELIF

Em alguns casos, é necessário inserir passos intermediários, ou seja, uma segunda condição caso a primeira não seja atendida. "**se** a bebida for suco de laranja, traga, **senão se** for suco de limão, traga, **senão** feche o restaurante". O Algoritmo abaixo resume este problema:

```
bebida = "suco de cevada"
if(bebida=="suco de laranja"):
    print("Trazer")
elif(bebida=="suco de limão"):
    print("Trazer")
```

```
else:  
    print("Fechar o restaurante")
```

Blocos condicionais podem usar-se de várias variáveis por exemplo. O script abaixo se chama "teste_se_sua_namorada_e_um_vampiro.py":

```
sai_de_dia = True #Verdadeiro  
tem_coloracao = True #Verdadeiro  
tem_coracao = False #Falso  
idade = 22  
  
if(sai_de_dia and tem_coloracao and idade<160 and not tem_coloracao):  
    print("Fique tranquilo, você está seguro")  
else:  
    print('Melhor estar preparado, sua namorada é um vampiro!')
```

EXERCÍCIO PROPOSTO

Com todos estes conceitos demonstrados, é hora de criar um programa prático e que pode dar auxílio com o leão da receita.

Desenvolver uma calculadora que solicite ao usuário o seu **nome**, seu **endereço**, a **empresa** que ela trabalha, seu **cargo** e o seu **salário bruto**.

Com estes dados, a calculadora deve fazer o cálculo de qual será o **salário líquido** após as deduções de **Imposto de Renda**.

Como saída, a calculadora deve exibir os **dados coletados** e o resultado do **salário líquido**, e a **porcentagem que foi descontada** seguindo a tabela abaixo:

Base de cálculo (R\$)	Alíquota (%)	Parcela a deduzir do IRPF (R\$)
Até 1.903,98	isento	isento
De 1.903,99 até 2.826,65	7,5%	R\$ 142,80

De 2.826,66 até 3.751,05	15%	R\$ 354,80
De 3.751,06 até 4.664,68	22,5%	R\$ 636,13
Acima de 4.664,68	27,5%	R\$ 869,36

EXERCÍCIOS COMPLEMENTARES

1. O que a variável bacon contém após a execução do código a seguir?

```
bacon=20  
bacon+1
```

2. Por que essa expressão causa um erro? Como pode-se consertar isso?

```
'Eu comi ' + 99 + ' burritos'
```

3. O que as seguintes expressões avaliam e qual é o resultado booleano delas?

Ex: (5 > 4) and (3 == 5)

Resposta: “A expressão avalia se 5 é maior que 4 e se 3==5, entrega o resultado False”

- a) not (5 > 4)
- b) (5 > 4) or (3 == 5)
- c) not ((5 > 4) or (3 == 5))
- d) (True and True) and (True == False)
- e) (not False) or (not True)

4. Identifique os blocos de código, verifique seu fluxo e como resposta diga quantas vezes a palavra “**salaminho**”, “**bacon**” e “**fritas**” foram impressas:

```
salaminho = 0  
  
if salaminho == 10:  
    print('fritas')  
  
    if salaminho > 5:  
        print('bacon')
```

```
else:

    print('fritas')

    print('salaminho')

print('salaminho')
```

CAPÍTULO 5

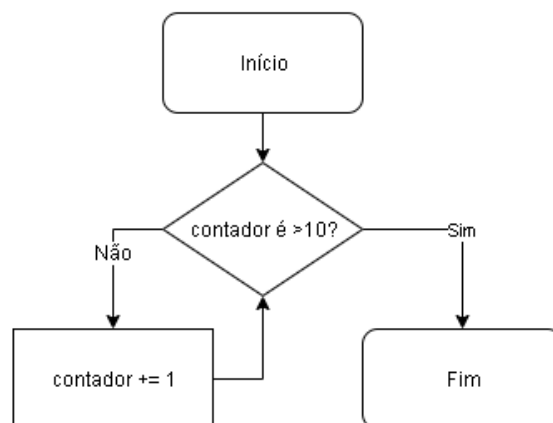
Estruturas de repetição

WHILE

A automação de tarefas repetitivas é sem dúvida o grande trunfo da programação. E em Python não é diferente.

No dia a dia é possível ouvir “Faça isso **enquanto** essa tarefa não estiver pronta”, “Coma **enquanto** a comida está quente” ou “Esse relógio funciona **enquanto** ele tem pilhas”.

Um dos blocos responsáveis por repetir comandos é a estrutura “**while**”, do inglês “**enquanto**”. Este bloco basicamente “faz algo **enquanto** outra coisa for verdade”. O gráfico abaixo demonstra o funcionamento básico desta estrutura:



A estrutura **while** exige apenas um parâmetro que é a condição que mantém o loop. Abaixo está representado o código que traduz o gráfico:

```
contador = 0

while contador<10:

    contador += 1

    print(contador)
```

O DEBUG

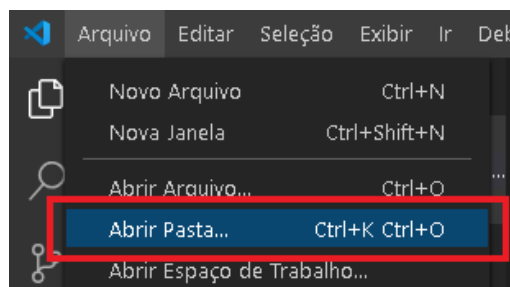
Bugs são erros que um código possui, não se deve enganar, eles são comuns, e com o tempo até o melhor dos programadores os comete. Com as estruturas de repetição alguns bugs são mais transparentes e difíceis de serem captados, e se faz necessário executar o código linha por linha.

Para esta tarefa o VSCode possui uma interface muito rica que auxilia o programador: “**O modo de Debug**”.

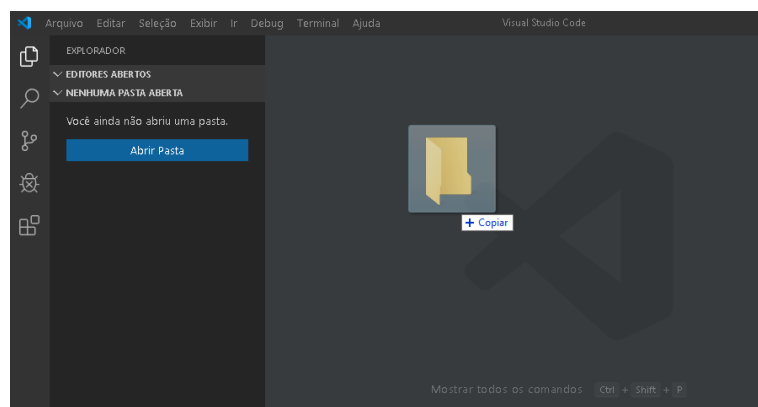
O modo de debug exige que uma pasta esteja aberta nele, caso contrário (se apenas um arquivo está aberto) ele não consegue entender onde devem ser criadas as configurações de debug.

Então, para tornar esta interface realizável, é necessário primeiramente abrir a pasta que o arquivo está. Isso vai fazer com que o VSCode salve o arquivo de configuração de debug nesta pasta.

Para abrir uma pasta no VSCode, pode-se usar o botão “**Abrir Pasta**”, usar o menu “**Arquivo**”>“**Abrir Pasta**” ou usando o comando “**Ctrl+O**”.



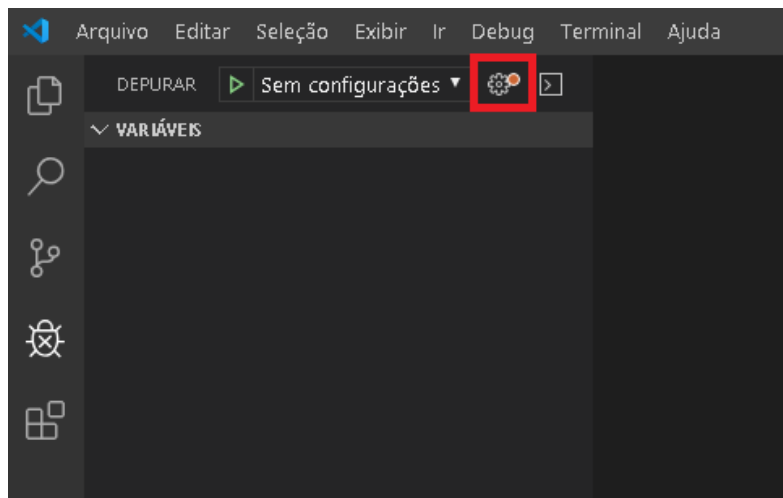
Também é possível simplesmente arrastar e soltar a pasta do arquivo para dentro do VSCode.



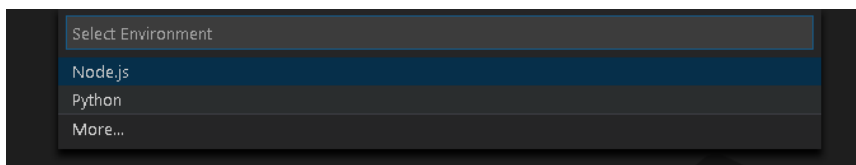
Após a pasta aberta, basta clicar no besouro no menu lateral esquerdo com o ícone:



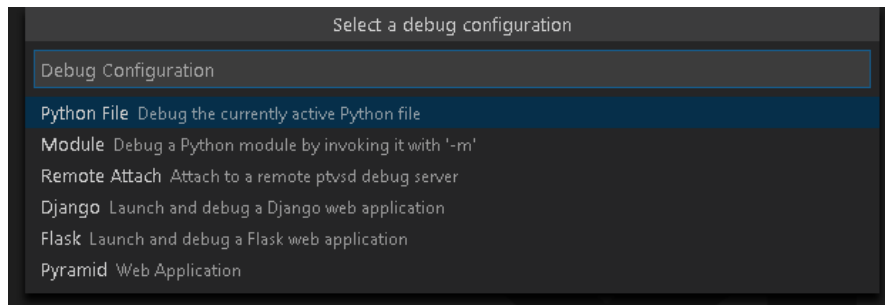
Neste momento, é necessário customizar como o modo Debug vai funcionar. De início, é necessário clicar na engrenagem, como visto abaixo:



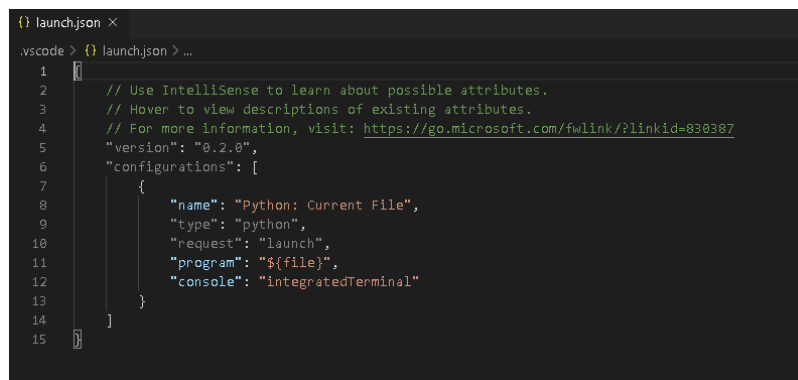
Os ambientes instalados no VSCode serão listados, basta selecionar o ambiente Python:



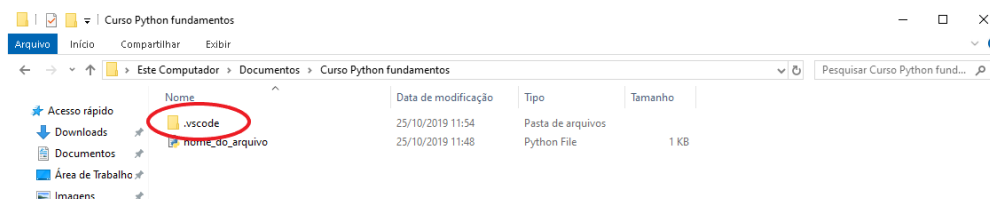
Por fim, selecione o debug para um arquivo Python chamado de **“Python File”**.



Pronto, o arquivo de debug foi gerado. Este arquivo só precisa ser gerado uma única vez por projeto e pode ser posteriormente copiado para outros.

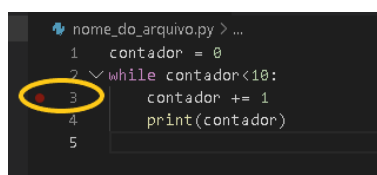


É possível ver que o VSCode criou uma pasta no mesmo diretório:



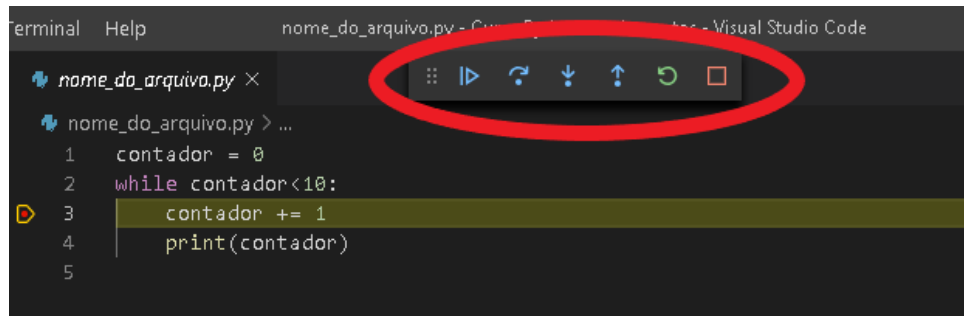
Agora é só fechar este arquivo criado e voltar ao arquivo de script. A partir deste momento é possível rodar um script passo a passo verificando e alterando o valor das variáveis.

Para criar um ponto de parada, ou **"breakpoint"**, como é conhecido, basta apenas clicar na ponta da linha do lado esquerdo do número. Um ponto vermelho será criado.

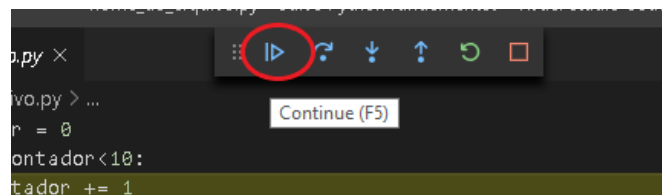


Agora, ao executar o código ele vai parar na linha em que foi criado um **breakpoint**.

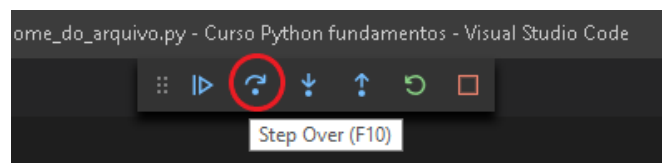
Enquanto o modo de Debug está ativo, uma barra com ferramentas fica disponível no topo.



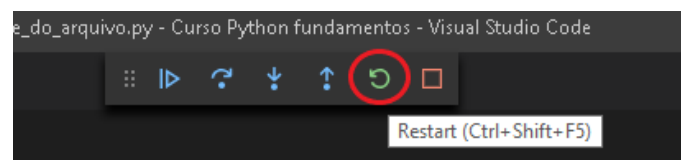
Nela, é possível “continuar”, ou seja, fazer o algoritmo caminhar até o próximo **breakpoint**. o atalho para este comando é o **F5**:



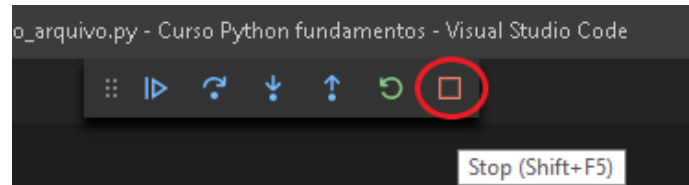
É possível avançar uma etapa de cada vez, ou seja, passo a passo no código. O atalho para este comando é o **F10**:



O VSCode possibilita fazer com que o algoritmo reinicie do começo com o botão “**Restart**”:



Por fim, se necessário é possível fazer com que o debug pare clicando no botão stop ou pressionando “**Shift+F5**”:



BREAK

Finalmente, o ambiente de debug está configurado e preparado. Com ele é mais fácil entender a utilização do **break**.

Loops While podem conter estruturas condicionais como o **if**. Em algum momento pode ser necessário parar se uma outra condição for atendida. Para estas situações existe o “**if**”. Ele faz com que seja possível sair de um loop sem que ele tenha acabado.

No exemplo abaixo, um usuário tenta acertar o número da mega sena. Caso ele acerte antes de seu saldo acabar ele vence o jogo. Nota-se que ao acertar o número o programa deve sair do loop, para isso, ele usa o “**break**”.

```
saldo_de_creditos=10

numero_usuario=0

mega_sena=5

while saldo_de_creditos>0:

    saldo_de_creditos -= 1 #Retira saldos

    numero_usuario = input("Qual é o número da mega
sena?") #Obtém um número de usuário

    if (mega_sena==int(numero_usuario)): #Testa se o
usuário ganhou
```

```
print("Acertou!")

break

print(saldo_de_credits)
```

ESCOPO DE VARIÁVEIS

Um conceito muito importante em programação é o escopo de variáveis. Uma variável não pode por exemplo ser usada antes de ser criada. O código abaixo retrata este erro quando a variável omelete é usada antes de ser criada:

```
bacon=2

ovos=1

print(bacon)

print(ovos)

print(omelete)

omelete=3
```

Existe ainda um problema quando uma variável é criada dentro de um bloco. O código abaixo reproduz sem problema algum pois a condição é atendida e o código entra no bloco "if":

```
var_primeiro_bloco = 0

if(2!=3):

    var_segundo_bloco=2

print(var_primeiro_bloco )

print(var_segundo_bloco)
```

Porém, ao supor uma condição que não seja atendida ao executar será exibido um erro pois para o programa aquela variável não existe.

```
var_primeiro_bloco = 0

if (2==3):

    var_segundo_bloco=2

print(var_primeiro_bloco )

print(var_segundo_bloco)
```

```
6 print(var_primeira_camada)
7 print(var_segunda_camada)
```

Exception has occurred: NameError
name 'var_segunda_camada' is not defined
File "C:\Users\Winicio Schmidt\Documents\Curso Python fundamentos\nome_do_arquivo.py", line 7, in <module>
 print(var_segunda_camada)

Para evitar este erro, sempre se busca deixar a variável no mesmo “nível hierárquico” que ela será usada. Para resolver o problema do código anterior basta declarar esta variável no mesmo “nível” que a função **print()** será usada, desta forma, a variável sempre irá existir.

```
var_primeira_camada = 0
var_segunda_camada = 0

if (2==3):

    var_segunda_camada=2

print(var_primeira_camada)
print(var_segunda_camada)
```

MÓDULOS

Imagine que é Natal. Compra-se um presente incrível, que faz mil peripécias, entretanto, ao abrir é constatado que ele está sem baterias. Muitas

linguagens são assim, existe muita coisa possível de ser feita, mas não existe nada pronto em primeira ocasião.

Python é diferente, ele possui diversas “baterias” prontas chamadas **módulos**. Existem, nativamente **módulos** para criação de datas (date e datetime), **números aleatórios** (random), **criação de telas gráficas** (tkinter), **criptação de dados** (crypt) e diversas outras.

Para importar um módulo usa-se a diretiva “**import**” e o nome do módulo que se deseja importar.

Como exemplo, para importar o módulo para a geração de números aleatórios o código abaixo pode ser usado:

```
import random
```

Após o módulo ser importado todas as funções contidas nele estarão disponíveis. O módulo **random** pode, por exemplo, gerar um número aleatório partindo de 1 até 10 com o código abaixo:

```
import random

numero_aleatorio = random.randint(1,10)
```

A função “**randint**” pertence ao módulo **random**, então, para usá-la basta escrever o nome do módulo seguido de um ponto e o nome da função.

Alguns módulos possuem nomes muito grandes que dificultam a leitura. Para sanar este problema é possível dar um apelido para o módulo com a diretiva “**as**”. No caso, anterior o código ficaria da seguinte forma:

```
import random as rd

numero_aleatorio = rd.randint(1,10)
```

Desta forma, um número aleatório foi gerado, exatamente como no exemplo anterior, porém agora o módulo **random** se chama **rd** no código.

Alguns módulos são excessivamente grandes e pesados, principalmente os de Machine Learning. É possível, nestes casos importar apenas uma única função. Tomando novamente o exemplo anterior, é possível importar apenas a função “**randint**” usando a diretiva “**from**” da seguinte forma:

```
from random import randint

numero_aleatorio = randint(1,10)
```

Nestes casos, também é possível atribuir um apelido para a função:

```
from random import randint as rd  
numero_aleatorio = rd(1,10)
```

EXERCÍCIO PROPOSTO

“Criar um jogo de adivinhação que faça o sorteio de um número de 1 até 60, solicite ao usuário um número de 1 até 60 e faça a checagem se o usuário acertou este número.

Implementar também, um sistema de créditos, ou número de tentativas semelhante ao visto na aula. **Se** o usuário acertar o número antes que o contador de créditos zerar imprimir a mensagem ‘Parabéns você ganhou :), **senão** imprimir ‘Infelizmente você não conseguiu :(.’

Como complemento perguntar ao usuário se ele deseja sair a cada tentativa.”

EXERCÍCIOS COMPLEMENTARES

1. Porque o código educativo abaixo não funciona? Como esse código pode ser reparado?

```
direcao = True  
  
alcool = True  
  
if(direcao and alcool):  
  
    morte = True  
  
else:  
  
    vida = True  
  
print(vida)
```

2. Um mercado quer fazer uma promoção. Foi decidido que será dado um desconto por categoria conforme a tabela abaixo:

Categoria	Desconto (%)
Comida	10%
Bebida	7%
Produtos de Limpeza	5%
DVD Galinha Pintadinha	75%
Outros	3%

Faça um algoritmo que pergunte ao usuário o **nome**, o **preço** e a **categoria** do produto. Após isso, faça o cálculo de desconto e mostre ao usuário o nome do produto, o valor antes e depois da promoção.

3. Faça um assistente que pergunte ao usuário uma **hora de início** e uma **hora de fim** e responda se ele tem compromissos agendados conforme a tabela abaixo:

Horário	Agenda
8:00 até 10:00	Reunião com a diretora
10:30 até 12:00	Visita ao cliente "Simpatia&Alegria"
12:00 até 13:00	Almoço
13:30 até 15:00	Reunião de debate com Arquiteto
18:00 até 23:00	Reunião técnica/prática para discussão do aproveitamento de Cevada no corpo humano

Mostre também, caso exista, todos os compromissos que existam neste período.

CAPÍTULO 6

Coleções Python

LISTAS

Organizar dados é fundamental e torna o trabalho de programar muito mais fácil. Imagine que exista uma lista de produtos como:

- Bacon
- Ovos
- Manteiga
- Pão

Guardar estes dados em variáveis se torna muito cansativo e oneroso como no exemplo abaixo:

```
bacon = "Bacon"

ovos = "Ovos"

manteiga = "Manteiga"

pao = "Pão"
```

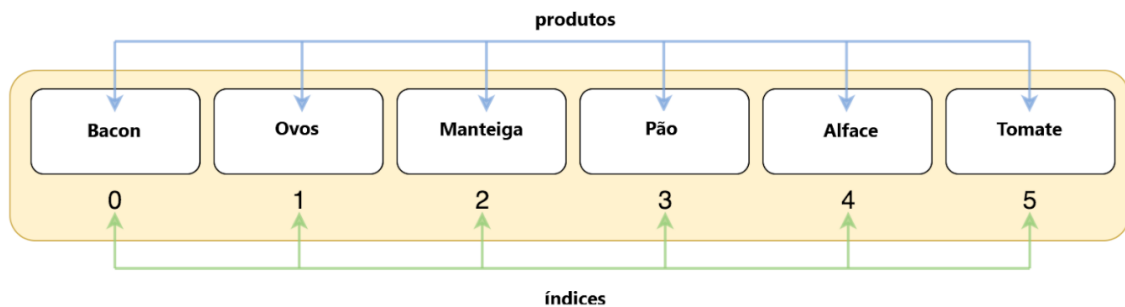
Imagine que uma lista chamada “**produtos**” pudesse ser criada contendo todos os valores. Em Python, existe um dado responsável por esta tarefa chamado “**list**”, do inglês **lista**. Abaixo é possível ver sua sintaxe:

```
produtos = ["Bacon", "Ovos", "Manteiga", "Pão",
            "Alface", "Tomate"]

print(produtos)
```

INDEXES

Desta maneira é criada uma lista contendo todos os valores. Para estes valores são atribuídos indexes como visto abaixo.



ACESSO DE ELEMENTOS

Depois de criada, uma lista pode ser acessada com a posição de seu index (partindo de 0). Por exemplo, para acessar o item “Manteiga”, que é o 3º elemento e ocupa ao index 2 basta usar:

```
produtos = ["Bacon", "Ovos", "Manteiga", "Pão",  
            "Alface", "Tomate"]  
  
print(produtos[2])
```

FATIAMENTO

É possível obter partes de uma lista. Supondo a seguinte lista de números:

```
numeros = [10,20,30,40,50,60,70,80,90]
```

Para obter os dois primeiros elementos basta usar o código abaixo:

```
print(numeros[:2])
```

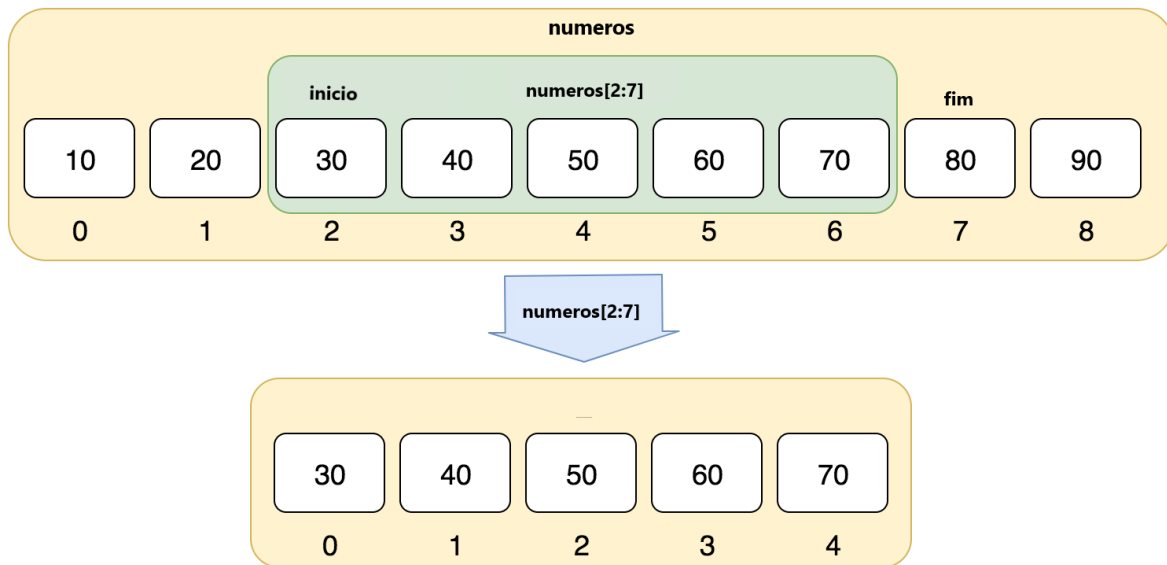
O mesmo pode ser usado para obter os elementos após o segundo elemento:

```
print(numeros[2:])
```

Ainda é possível obter trechos de uma lista:

```
print(numeros[1:3])
```

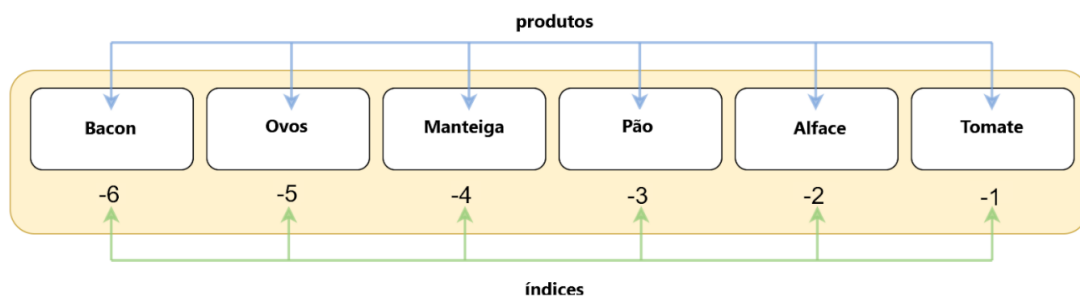
Em Python, essas propriedades são chamadas de “**Slicers**”, do inglês “**Fatiadores**” e seguem a regra vista na figura abaixo:



```
lista[intervalo_inferior:intervalo_superior]
```

É importante salientar que o intervalo inferior (**intervalo_inferior**) corresponde exatamente ao número de início do índice (intervalo fechado), já o intervalo superior (**intervalo_superior**) não é contido na lista (intervalo aberto).

Ainda é possível acessar os itens em sua forma inversa. Para isso é necessário imaginar a lista sendo percorrida de trás para frente:



Por exemplo, para acessar os dois últimos elementos de uma lista:

```
print(produtos[-2:])
```

Para obter uma lista com todos exceto os 3 últimos elementos basta usar:

```
nova_lista = produtos[:-3])
```

Desta forma é correto afirmar que existe uma correspondência de índices, na lista do exemplo anterior, esta relação é representada abaixo:

Index positivo	0	1	2	3	4	5
Correspondente Negativo	-1	-2	-3	-4	-5	-6

As listas facilitam a tarefa de obter todo item presente em um índice múltiplo de um número. Por exemplo, se for desejado obter todos os índices pares da lista, ou seja, todo múltiplo de 2:

```
numeros=[10, 20, 30, 40, 50, 60, 70, 80, 90]
numeros[::2]
```

É possível também usar um passo negativo para obter percorrendo a lista de trás para frente:

```
numeros=[10, 20, 30, 40, 50, 60, 70, 80, 90]
numeros[::-2]
```

VERIFICAR SE UM ITEM EXISTE

O tipo de dado "list" torna possível a busca por nome. Por exemplo, o código abaixo verifica se existe o nome **"Pão"** na lista:

```
ceu = ["Bacon", "Ovos", "Manteiga", "Pão", "Alface",
"Tomate"]

if "Pão" in ceu:

    print("No céu tem pão!")

else:

    print("No céu não tem pão.")
```

ENCONTRAR O ÍNDICE DE UM VALOR

Para encontrar o índice de um valor existe o método **index()**. Seu uso é contemplado abaixo:

```
produtos = ["Bacon", "Ovos", "Manteiga", "Pão",  
"Alface", "Tomate"]  
  
index_tomate = produtos.index("Tomate")  
  
print(index_tomate)
```

Caso existam elementos repetidos, apenas o primeiro índice será retornado:

```
produtos = ["Bacon", "Tomate", "Manteiga", "Pão",  
"Alface", "Tomate"]  
  
index_tomate = produtos.index("Tomate")  
  
print(index_tomate)
```

LISTAS DENTRO DE LISTAS

Uma lista pode conter além de um valor absoluto uma outra lista, ou seja, uma lista dentro de uma lista:

```
comidas = ["Bacon", "Ovos", "Manteiga", "Pão",  
"Alface", "Tomate"]  
  
musicas = ["Are you Ready",  
           "Highway To Hell",  
           "Better Than Sex",  
           "It's a Long Way to the Top",  
           "I Was Made For Lovin' You"]  
  
festa = [comidas, musicas]  
  
print(festa)
```

Para acessar um valor dentro de outro, pode-se usar a seguinte notação:

```
print(festa[0][0]) #Bacon  
  
print(festa[1][2]) #Better Than Sex
```

ALTERAÇÃO DE VALORES

A lista pode ter seus valores alterados. Por exemplo, o item “**Manteiga**” pode ser substituído por “**Margarina**” bastando saber qual é o seu índice:

```
produtos = ["Bacon", "Ovos", "Manteiga", "Pão",  
"Alface", "Tomate"]  
  
print(produtos[2])  
  
produtos[2] = "Margarina"  
  
print(produtos[2])
```

Substituir parte de uma lista é factível usando um fatiamento. Por exemplo, para substituir os dois últimos elementos de uma lista por outra lista pode-se usar:

```
numeros = [10, 20, 30, 40, 50, 60, 70, 80, 90]  
numeros[-2:] = [1,2,3,4]  
print(numeros)
```

INSERÇÃO DE DADOS EM LISTAS

Uma lista pode ainda ter seus elementos adicionados com o método **append()**:

```
produtos = ["Bacon", "Ovos", "Manteiga", "Pão",  
"Alface", "Tomate"]  
  
produtos.append("Burritos")  
  
print(produtos)
```

ENCONTRANDO O TAMANHO DE UMA LISTA

Outra função muito útil é função **len()** que pode ser usada para verificar o tamanho de uma lista:

```
produtos = ["Bacon", "Ovos", "Manteiga", "Pão",
            "Alface", "Tomate"]

tam_lista = len(produtos)

print("O tamanho da lista é: "+str(tam_lista))
```

REMOÇÃO DE ITENS EM LISTAS

Uma lista pode precisar ter itens removidos. Para isso existem, 4 principais formas:

- remove()
- pop()
- del()
- clear()

O método **remove()** faz com que um determinado valor seja removido da lista. Por exemplo, pode ser necessário remover **"Alface"**. Para esta tarefa pode-se usar:

```
produtos = ["Bacon", "Ovos", "Manteiga", "Pão",
            "Alface", "Tomate"]

produtos.remove("Alface")

print(produtos)
```

Importante notar que se o valor estiver repetido na lista, todos os duplicados serão removidos.

Para remover apenas o último elemento é interessante o uso da função **pop()**. No exemplo abaixo ela irá remover o item "Tomate":

```
produtos = ["Bacon", "Ovos", "Manteiga", "Pão",
            "Alface", "Tomate"]

produtos.pop()

print(produtos)
```

O método **pop()** também remove um índice específico bastando apenas passar o índice desejado como parâmetro:

```
produtos = ["Bacon", "Ovos", "Manteiga", "Pão",  
"Alface", "Tomate"]  
  
produtos.pop(3)  
  
print(produtos)
```

Para deletar uma lista ou parte dela se faz uso da diretiva **del**:

```
produtos = ["Bacon", "Ovos", "Manteiga", "Pão",  
"Alface", "Tomate"]  
  
del produtos[0] #Remove o item Bacon  
  
print(produtos)  
  
del produtos #Deleta completamente a lista  
  
print(produtos) #A lista não existe mais e irá gerar um  
erro
```

No exemplo anterior a lista foi completamente eliminada, causando um erro pois não estava declarada. Se for desejado apenas limpar a lista pode-se fazer uso do método **clear()**:

```
produtos = ["Bacon", "Ovos", "Manteiga", "Pão",  
"Alface", "Tomate"]  
  
produtos.clear()  
  
print(produtos)
```

UNIÃO DE LISTAS:

Duas listas podem ser unidas bastando apenas utilizar o operador +:

```
produtos_carol = ["Bacon", "Ovos", "Pão"]  
  
produtos_ana = ["Alface", "Tomate"]  
  
produtos = produtos_carol + produtos_ana
```

```
print(produtos)
```

Caso seja necessário unir dados de uma segunda lista na primeira o método **extend()** se faz útil, No exemplo a lista de **Carol** recebe os itens da lista de **Ana**:

```
produtos_carol = ["Bacon", "Ovos", "Pão"]

produtos_ana= ["Alface", "Tomate"]

produtos_carol.extend(produtos_ana)

print(produtos_carol)
```

ATRIBUIÇÕES DE LISTAS EM VARIÁVEIS:

É possível, claramente, atribuir os valores de uma lista para variáveis, como por exemplo:

```
gato = ['Gordo', 'Caramelo', 'Preguiçoso']

porte = gato[0]

cor = gato[1]

disposicao = gato[2]
```

A linguagem Python pode simplificar esta tarefa usando a sintaxe:

```
gato = ['Gordo', 'Caramelo', 'Preguiçoso']

porte, cor, disposicao = gato
```

ORDENAR UMA LISTA

Uma lista Python possui a função de ordenação chamada **sorted()**. Com ela é possível organizar itens por **ordem alfabética**:

```
produtos = ["Bacon", "Ovos", "Manteiga", "Pão", "Alface", "Tomate"]

produtos_alfabeticos = sorted(produtos)

print(produtos_alfabeticos)
```

Esta função permite ainda classificar em ordem inversa usando o parâmetro "reverse=True", fazendo a ordem ser invertida:


```
produtos = ["Bacon", "Ovos", "Manteiga", "Pão", "Alface", "Tomate"]

produtos_alfabeticos = sorted(produtos, reverse=True)

print(produtos_alfabeticos)
```

Para fixar o conceito de posições em listas é interessante usar o exemplo abaixo descrito como **"Sorteador de frase de bom dia"**. Neles, é sorteado um número que servirá de correspondência ao índice.

```
import random

mensagens = [

    'Não importa a cor do céu. Quem faz o dia bonito é você.',

    'Bom dia! Que seu dia seja igual a vontade de Deus: bom, perfeito e agradável.',

    'Que o dia comece bem e termine ainda melhor.',

    'A cada nova manhã, nasce junto uma nova chance. Bom dia!',

    'Bom dia! Comece o dia sorrindo. Afinal, coisa boa atrai coisa boa.',

    'Hoje eu acordei tão linda que quando fui bocejar, miei.',

    'O seu sorriso pode mudar o dia de alguém.',

    'Hoje você tem duas opções: ser feliz ou ser mais feliz ainda. Bom dia!',

    'Acorda! O melhor ainda está por vir. Bom dia!']

intervalo_inferior = 0

intervalo_superior = len(mensagens) - 1

index_sorteado = random.randint(intervalo_inferior, intervalo_superior)
```

```
print(mensagens[index_sorteado])
```

VALORES MÍNIMOS, MÁXIMOS, SOMA E CONTAGEM

O Python oferece ainda as funções `min()`, `max()` e `sum()`, através das quais é possível encontrar, respectivamente, o menor valor, o maior valor ou ainda realizar a soma de todos os elementos da lista. No seguinte exemplo é possível ver como utilizá-las.

```
numeros = [10,20,30,40,50,60,70,80,90]

print(max(numeros)) #Máximo
print(min(numeros)) #Mínimo
print(sum(numeros)) #Soma
```

Por fim, existe ainda o método **`count()`**, que retorna o número de ocorrências de determinado objeto, passado como parâmetro, em uma lista. No seguinte exemplo é possível ver seu funcionamento:

```
produtos = ["Bacon", "Ovos", "Tomate", "Pão", "Alface", "Tomate"]

print(produtos.count("Bacon")) #Retorna 1
print(produtos.count("Tomate")) #Retorna 2
```

TUPLAS

As Tuplas são, em suma, muito semelhantes às listas. Para declarar uma **tupla** basta colocar o seu conteúdo entre parênteses `()` de seguinte forma:

```
cartoes_credito = ("5403 3020 2826 3814", "4532 5253 7221 0240", "5033 5223 7004 4680")

print(cartoes_credito)
```

Os métodos de acesso a um elemento de um tupla são muito semelhantes aos métodos de acesso em uma lista:

```
cartoes_credito = ("5403 3020 2826 3814", "4532 5253 7221 0240", "5033 5223 7004 4680")

print(cartoes_credito[0]) #Primeiro elemento
```

```
print(cartoes_credito[1:2]) #Segundo elemento da tupla

print(cartoes_credito[-1]) #Último elemento da tupla
```

A principal diferença entre uma tupla e uma lista está no fato de que uma **tupla não pode ser alterada**, ou seja, uma vez criada não pode ser modificada:

```
cartoes_credito = ("5403 3020 2826 3814", "4532 5253 7221
0240", "5033 5223 7004 4680")

cartoes_credito[0] = "3755 513792 01790"
```

ALTERAÇÃO EM TUPLAS

Não é possível alterar uma tupla, porém, é possível transformá-la em uma lista, editá-la e então transformá-la em uma nova tupla:

Para transformar uma tupla em uma lista se usa da função **list()** e para tornar o dado em uma **tupla** novamente se usa da função **tuple()**:

```
tupla_cartoes_credito = ("5403 3020 2826 3814", "4532 5253
7221 0240", "5033 5223 7004 4680")

lista_cartoes_credito = list(tupla_cartoes_credito)

lista_cartoes_credito[0] = "5139 5350 8305 0813"

tupla_cartoes_credito = tuple(lista_cartoes_credito)

print(tupla_cartoes_credito )
```

O LAÇO DE REPETIÇÃO FOR

Tomando uma lista como vista abaixo é possível imprimir o elementos individualmente usando um loop **while** em conjunto com a função **len()** e um contador:

```
contador = 0

marcas_carros = ["Audi", "BMW", "Mercedes"]

while contador < (len(marcas_carros)):

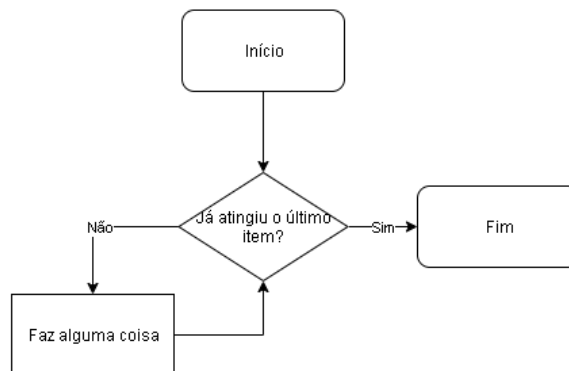
    print(marcas_carros[contador])

    contador += 1
```

Para isso 5 linhas e uma certa complexibilidade foram necessárias. Para estes casos (e na verdade, para a maioria deles), existe um bloco de função chamado “**for()**” do inglês “**Para**”.

Esta instrução funciona de uma forma semelhante a dizer “**Para** cada **item** da **lista** faça algo”.

O seu funcionamento é bem parecido com o loop **while** e pode ser visto no diagrama abaixo:



Tomando o exemplo anterior e substituindo o **while** pelo **for** o código fica da seguinte forma:

```
marcas_carros = ["Audi", "BMW", "Mercedes"]  
for item in marcas_carros:  
    print(item)
```

O primeiro parâmetro “**item**” é o que a função **for** recebe de uma **lista**, no caso os itens que a lista possui.

Caso uma determinada condição seja atendida é possível usar o **break** para encerrar a execução do laço. Por exemplo, se o item for igual a “BMW” é desejado que o programa pare:

```
marcas_carros = ["Audi", "BMW", "Mercedes"]  
for item in marcas_carros:  
    print(item)  
  
    if(item=="BMW"):  
        break
```

Uma função útil para trabalhar com laços for é a função **range()**. A função range gera um intervalo de valores. Por exemplo, para imprimir 5 números de 0 até 4 pode-se usar:

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

A função também permite gerar intervalos definidos passando um início e um fim. Por exemplo a geração de números de 5 até 14:

```
for numero in range(5, 15):  
  
    print(numero)
```

Ainda é possível gerar intervalos com um passo diferente de 1. Se for necessário gerar um intervalo de 4 até 16 com um passo de 2 pode-se usar:

```
for numero in range(4, 17, 2):  
  
    print(numero)
```

DICIONÁRIOS

Trabalhar apenas com índices torna-se muito difícil para estruturas de dados mais complexos. Por exemplo, como organizar o conjunto de dados abaixo sendo o nome e a quantidade de um produto?

- bacon = 2
- ovos= 3
- morangos = 1

Existe em Python um tipo de dado chamado dicionário, com ele é possível armazenar um conjunto de dados de forma estruturada em “**keys**” do inglês **chaves** e “**values**” do inglês **valores**, sendo assim um valor sempre será associado a uma chave. Para o exemplo anterior, pode-se dizer que “**bacon**” é a chave para o valor “**2**”.

CRIAÇÃO

Para criar um dicionário basta seguir a estrutura abaixo, que contém uma chave e um valor. Para definir múltiplos itens, a vírgula deve ser usada como espaçador:

```
dicionario = {
```

```
"chave1":valor1,  
  
"chave2":valor2,  
  
"chave2":valor2  
  
}
```

Alterando para a realidade do exemplo anterior obtém-se:

```
qtd_produtos = {  
  
    "bacon":2,  
  
    "ovos":3,  
  
    "morangos":1  
  
}
```

Desta forma os valores foram atribuídos a uma chave.

ACESSAR ITENS

Para acessar estes valores basta colocar a lista e entre colchetes a chave que se deseja obter de forma semelhante aos indexes da lista. Por exemplo, para saber a quantidade de morangos basta usar o código abaixo:

```
qtd_produtos = {  
    "bacon":2,  
    "ovos":3,  
    "morangos":1  
}  
  
print(qtd_produtos["morangos"])
```

O mesmo serve para todas as demais chaves.

Para obter todos os valores de um dicionário existe a função **values()**. Esta função devolve um dicionário contendo uma lista com os valores. Como por exemplo:

```
qtd_produtos = {  
    "bacon":2,  
    "ovos":3,
```

```
        "morangos":1
    }
    print(qtd_produtos.values())
```

O mesmo serve para obter as chaves:

```
qtd_produtos.keys()
```

É possível através de um laço for imprimir estes valores:

```
qtd_produtos = {

    "bacon":2,

    "ovos":3,

    "morangos":1

}

for valor in qtd_produtos.values():

    print(valor)
```

Para converter o **dicionário** em uma **lista** ou **tupla** pode ser fazer uso das funções **list()** e **tuple()**:

```
qtd_produtos = {

    "bacon":2,

    "ovos":3,

    "morangos":1

}

lista = list(qtd_produtos.values())

tupla = tuple(qtd_produtos.values())
```

```
print(lista)

print(tupla)
```

EDITAR ITENS

Para editar um item em um dicionário basta atribuir um novo valor em sua chave. Por exemplo, mudar o valor de “bacon” para 50

```
qtd_produtos = {

    "bacon":2,

    "ovos":3,

    "morangos":1

}

print(qtd_produtos["bacon"])

qtd_produtos["bacon"] = 50

print(qtd_produtos["bacon"])
```

REMOVER ITENS

Para remover itens existem basicamente três formas:

- pop()
- del()
- clear()

Com o método **pop()**, uma “**key**” deve ser passada e será removida do dicionário:

```
qtd_produtos.pop("bacon")
```

O método **del**, de forma semelhante às listas pode-se excluir um dicionário ou um valor de um dicionário:


```
qtd_produtos = {  
  
    "bacon":2,  
  
    "ovos":3,  
  
    "morangos":1  
  
}  
  
del qtd_produtos["morangos"] #Exclui morangos  
  
print(qtd_produtos)  
  
del qtd_produtos #Exclui a lista  
  
print(qtd_produtos) #Gera um erro
```

Já o método **clear()** faz com que o dicionário seja limpo e todas os **keys** e **values** sejam removidos.

```
qtd_produtos = {  
  
    "bacon":2,  
  
    "ovos":3,  
  
    "morangos":1  
  
}  
  
qtd_produtos.clear()  
  
print(qtd_produtos)
```

LISTAS, TUPLAS E DICIONÁRIOS DENTRO DE DICIONÁRIOS

A estrutura de dicionários permite colocar listas, tuplas e dicionários dentro de dicionários. Por exemplo, uma festa que tenha comida, convidados e já possua convidados confirmados pode ser representada da seguinte forma:

```
comida = {  
  
    "bacon":2,
```

```
        "ovos":3,

        "morangos":1

    }

    lista_convidados = [

        "Miguel",

        "Davi",

        "Arthur",

        "Pedro",

        "Gabriel",

        "Sophia",

        "Alice",

        "Julia",

        "Isabella",

        "Manuela"

    ]

    tupla_confirmados = (

        "Pedro",

        "Gabriel",

        "Julia",

        "Isabella",

        "Manuela"

    )
```

```
festa = {  
  
    "comida":comida,  
  
    "lista_convidados":lista_convidados,  
  
    "tupla_confirmados":tupla_confirmados  
  
}  
  
print(festa)
```

ERROS COMUNS AO TRABALHAR COM DADOS ESTRUTURADOS

ADIÇÃO EM ÍNDICE INEXISTENTE

Se houver uma tentativa de edição/adição de um índice que não existe, um erro será criado e o programa terá seu funcionamento comprometido. Como visto com o código abaixo:

```
produtos = ["Bacon", "Ovos", "Manteiga", "Pão",  
            "Alface", "Tomate"]  
  
produtos[10] = "Brigadeiro"
```

Existe, para estes casos o método **append**. Este método insere um item ao final da lista evitando o erro de índice fora de escopo.

REFERÊNCIA DE LISTAS

Em Python, listas são armazenadas em um lugar da memória. E memória é um recurso limitado. Em grandes listas a cópia torna-se um recurso impraticável. Para resolver isso quando você atribui uma lista a uma variável, na verdade você está atribuindo uma referência de lista à variável. Uma referência é um valor que aponta para um lugar na memória e uma referência de lista é um valor que aponta para uma lista.

Para ficar mais claro, o código abaixo será usado:

```
lista1 = [1,2,3,4]

lista2 = lista1

print(lista1)

print(lista2)

lista2[2] = 13

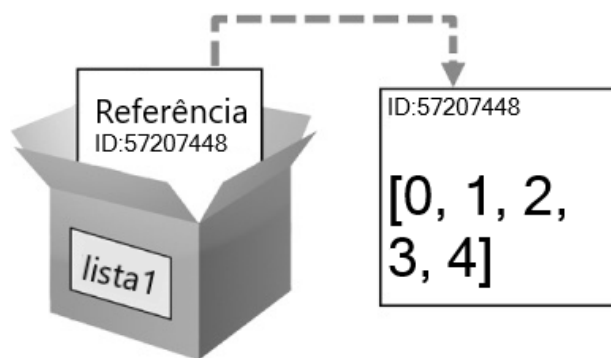
print(lista1)

print(lista2)
```

Ao reproduzir é possível ver o erro de referência que ele gera. Para entender o erro é preciso analisar o que aconteceu em cada etapa.

Na primeira linha `lista1` armazena a referência de uma lista (não seu valor):

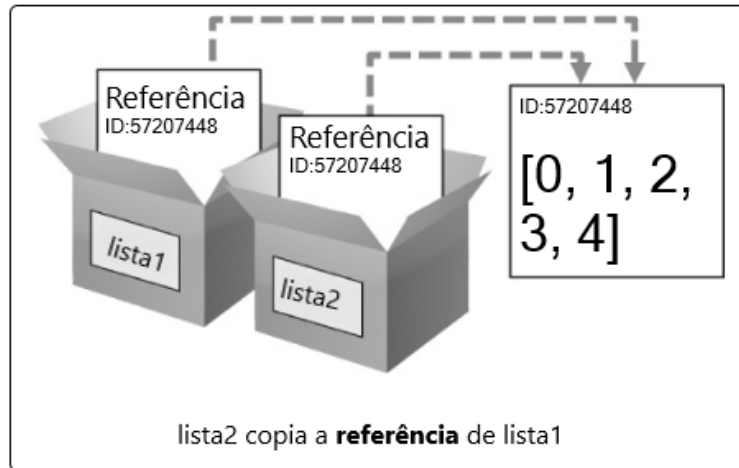
```
lista1 = [1,2,3,4]
```



`lista1 = [0, 1, 2, 3, 4, 5]` armazena a referência da lista e não seu valor

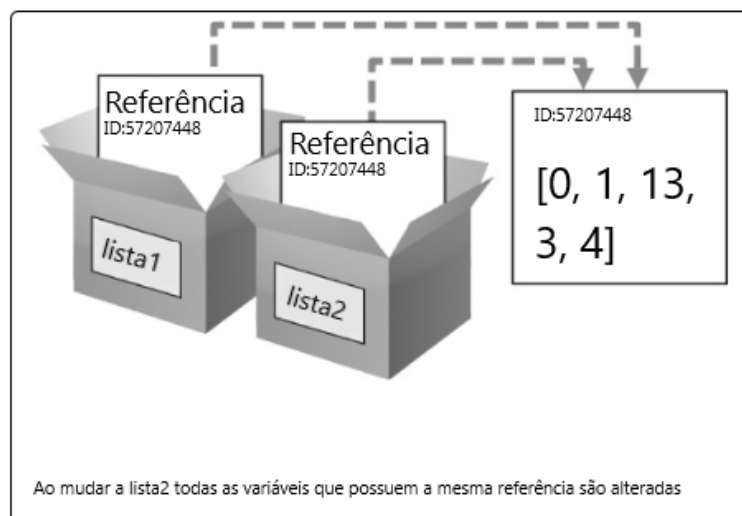
Na segunda linha, a variável **"lista2"** armazena a referência de **"lista1"**

```
lista2 = lista1
```



Ao mudar uma posição da variável lista2 todas as variáveis que possuem a mesma referência são alteradas:

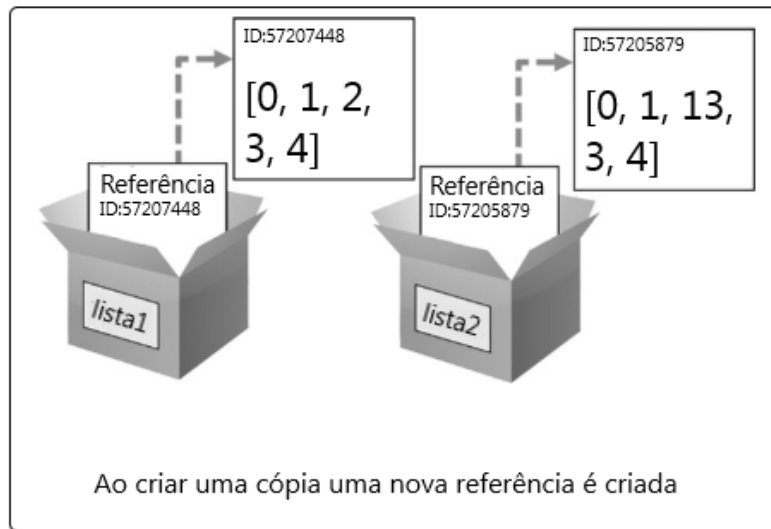
```
lista2[2] = 13
```



Para poder fazer efetivamente uma cópia basta usar o seguinte código:

```
lista1 = [1,2,3,4]  
lista2 = lista1[:]
```

Ao usar um fatiamento uma nova referência é criada e armazenada em "**lista2**".



EXERCÍCIO PROPOSTO

Escolher 10 Pokémons com base na lista [https://bulbapedia.bulbagarden.net/wiki/List_of_Pok%C3%A9mon_by_base_stats_\(Generation_VII-present\)](https://bulbapedia.bulbagarden.net/wiki/List_of_Pok%C3%A9mon_by_base_stats_(Generation_VII-present)) e criar uma espécie de **Pokédex**, ou seja, uma agenda em que o usuário pergunta o nome de um Pokémon e recebe como resposta as suas características.

Classifique ainda como 'Fraco' (Média abaixo de 40), 'Médio' (Média abaixo de 80) e "Forte" (Média acima de 80).

Se o Pokémon tiver um ataque maior que 150 imprima a mensagem "EXTREMAMENTE PERIGOSO".

EXERCÍCIOS COMPLEMENTARES

1. Em uma festa o DJ estava com problemas em um algoritmo que montava as músicas que seriam reproduzidas. Tomando como base o dicionário de músicas abaixo:

```
musicas_bailinho = {
    "funk": ["Só Love",
            "Rap da Felicidade",
            "Glamurosa"],
```

```
"rock":["Are you Ready",  
        "Highway To Hell",  
        "Better Than Sex",  
        "It's a Long Way to the Top",  
        "I Was Made For Lovin' You"],  
  
"eletrônica":["Akasha (Synthatic Remix)", "Hear Me  
Now", "FUEGO", "Your Body"],  
  
"tradicional russa":["Katyusha", "Tachanka", "Dark-  
Eyed Cossack Girl", "The Bogatyr"],  
  
anos 80:["Take On Me", "Never Gonna Give You Up",  
"Sweet Dreams", "Eye Of The Tiger"]  
  
}
```

- a) Como resolver o erro de reprodução do algoritmo?
- b) Como remover o gênero “funk” do dicionário?
- c) Como inserir uma nova lista com o gênero “rap”?

2. Monte um programa que contenha um **tupla** preenchida com os números de 0 até 10 por extenso. O programa deve ler uma entrada do usuário e mostrar o número por extenso.

3. Crie uma **tupla** contendo os 20 primeiros colocados do campeonato brasileiro de futebol. O usuário pode solicitar:

- 1. Os cinco primeiros colocados
- 2. Os quatro últimos
- 3. A posição do seu time
- 4. A lista completa com a respectiva posição
- 5. Qual é diferença de pontos do primeiro até o time que ele solicitar

4. Um programador se confundiu ao guardar dados de uma imagem. Faça um algoritmo que transforma os dados abaixo:

```
grid = [
    ['.', '.', '.', '.', '.', '.'],
    ['.', '0', '0', '.', '.', '.'],
    ['0', '0', '0', '0', '.', '.'],
    ['0', '0', '0', '0', '0', '.'],
    ['.', '0', '0', '0', '0', '0'],
    ['0', '0', '0', '0', '0', '.'],
    ['0', '0', '0', '0', '.', '.'],
    ['.', '0', '0', '.', '.', '.'],
    ['.', '.', '.', '.', '.', '.']
]
```

Nesta imagem:

```
..00.00..
.0000000.
.0000000.
..00000..
...000...
....0....
```

5. Desenvolver um algoritmo que recebe o valor total de um prêmio e o nome de participantes (quantos o usuário quiser!).

O algoritmo deve sortear o valor para todos os participantes sendo que o prêmio mínimo deve respeitar ser 10% do total dividido pelo número de usuários.

CAPÍTULO 7

Tratamento de Erros em Python

Em algoritmos, principalmente os que tem interface com o usuário é possível que erros sejam gerados. Muitos deles acabam destruindo o funcionamento do programa.

Quando uma aplicação está funcionando é necessário garantir que se o usuário errar algo o erro gerado não acabe interrompendo o funcionamento do programa.

TRY EXCEPT

Para ilustrar, imagine o exemplo a seguir:

```
idade = int(input("Qual é a sua idade? "))
print(idade)
```

Se o usuário entrar com um número decimal qualquer (1,2,3,4...) não existirá problemas e o código funcionará perfeitamente. Entretanto, caso o usuário entre com algo diferente disso como a palavra **"trinta"**, um erro será

```
PS C:\Users\Vinicio\Downloads\Apostilas> & C:/Users/Vinicio/AppData/Local/Programs/Python/Python39-64/Python.exe C:/Users/Vinicio/Downloads/Apostilas/remanufatura.py
Qual é a sua idade? trinta
Traceback (most recent call last):
  File "c:/Users/Vinicio/Downloads/Apostilas/remanufatura.py", line 3, in <module>
    idade = int(input("Qual é a sua idade? "))
ValueError: invalid literal for int() with base 10: 'trinta'
```

gerado:

Este erro ocorre porque não é possível converter uma palavra como **"trinta"** para um número.

Para dar tratamento neste erro, usamos o **try** (tentar) em conjunto com o **except** (exceção):

```
try: #Tentativa
    idade = int(input("Qual é a sua idade? "))
    print(idade)
except: #Em caso de erro
    print("Idade inválida!")
```

Desta forma sempre que um erro for gerado, o bloco de exceção será chamado.

ELSE NO TRATAMENTO DE EXCEÇÕES

É possível tentar executar alguma coisa e caso nenhum erro ocorra o bloco seja executado. Para fazer isso, usa-se a diretiva **else** da seguinte maneira:

```
try:
    idade = int(input("Qual é a sua idade? "))
    print(idade)
except:
    print("Idade inválida!")
else: #Caso nenhum erro ocorra
    print("Nenhum erro foi detectado")
```

Desta maneira, sempre que não forem detectados erros, o conteúdo do bloco **else** será executado.

FINALLY

Imagine, entretanto, que se deseja imprimir uma frase independentemente se houve erros ou não.

Para isso se usa a diretiva **Finally** como visto abaixo:

```
try:
    idade = int(input("Qual é a sua idade? "))
    print(idade)
except:
    print("Idade inválida!")
else:
    print("Nenhum erro foi detectado")
finally: #Executa de maneira independente
    print("Fim da execução do bloco")
```

Desta forma, o conteúdo sempre será executado independente da existência de erros ou não.

TRATAMENTO DE ERRO POR TIPO

Cada erro possui um tipo.

Por exemplo, se tentar dividir a string "2" por um inteiro 2 o erro **TypeError** (erro de tipo) será exibido:

```
resposta = "2"/2
```

```
resposta = "2"/2
TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

Caso tente-se dividir um número por 0, o erro **ZeroDivisionError** (Erro de divisão por Zero) será notado:

```
resposta = 5/0
```

```
resposta = 5/0
ZeroDivisionError: division by zero
```

Estes erros podem ser tratados em cada uma destas classes de erros usando o seu nome após o except:

```
try:
    resposta = 5/0
except ZeroDivisionError:
    print("Não se pode dividir por zero!")
```

Se for desejado, pode-se tratar diversos erros de forma adequada:

```
try:
    valor = int(input("Quanto espera gastar? ")) #Pode digitar um
    valor que não seja inteiro
    parcelas = int(input("Deseja dividir por quantos meses? ")) #Pod
    e digitar 0
    resposta = valor/parcelas #Pode gerar erro de divisão por 0
except ZeroDivisionError: #Tratamento de divisão por zero
    print("Não se pode dividir por zero parcelas!")
except ValueError: #Tratamento de erro na conversão
    print("Apenas números decimais como 1,2,3... são permitidos!")
```

Existem diversas classes de erros. A tabela abaixo demonstra os principais:

Erro	Descrição
ModuleNotFoundError	Quando um módulo não é encontrado
IndexError	Quando o index está fora da sequência buscada
KeyError	Quando a chave de um dicionário não é encontrada
SyntaxError	Quando um erro de sintaxe é encontrado
TypeError	Quando a operação, ou função não condiz com o tipo da variável
ValueError	Quando uma operação ou função recebe um argumento que tem o tipo certo, mas um valor inadequado
ZeroDivisionError	Quando se divide um valor por zero
FileExistsError	Quando se tenta criar um arquivo e ele já existe
FileNotFoundError	Quando um arquivo não é encontrado
PermissionError	Quando não se tem permissão para acesso
TimeoutError	Quando o tempo de execução da tarefa é esgotado

Caso duas exceções sejam geradas em uma linha, a seguinte hierarquia é seguida:

```
BaseException
+-- SystemExit
+-- KeyboardInterrupt
+-- GeneratorExit
+-- Exception
    +-- StopIteration
    +-- StopAsyncIteration
    +-- ArithmeticError
        |   +-- FloatingPointError
        |   +-- OverflowError
        |   +-- ZeroDivisionError
    +-- AssertionError
    +-- AttributeError
    +-- BufferError
```

```
    +-- EOFError
    +-- ImportError
    |    +-- ModuleNotFoundError
    +-- LookupError
    |    +-- IndexError
    |    +-- KeyError
    +-- MemoryError
    +-- NameError
    |    +-- UnboundLocalError
    +-- OSError
    |    +-- BlockingIOError
    |    +-- ChildProcessError
    |    +-- ConnectionError
    |    |    +-- BrokenPipeError
    |    |    +-- ConnectionAbortedError
    |    |    +-- ConnectionRefusedError
    |    |    +-- ConnectionResetError
    |    +-- FileExistsError
    |    +-- FileNotFoundError
    |    +-- InterruptedError
    |    +-- IsADirectoryError
    |    +-- NotADirectoryError
    |    +-- PermissionError
    |    +-- ProcessLookupError
    |    +-- TimeoutError
    +-- ReferenceError
    +-- RuntimeError
    |    +-- NotImplementedError
    |    +-- RecursionError
    +-- SyntaxError
    |    +-- IndentationError
    |    +-- TabError
```

```
+-- SystemError
+-- TypeError
+-- ValueError
|   +-- UnicodeError
|       +-- UnicodeDecodeError
|       +-- UnicodeEncodeError
|       +-- UnicodeTranslateError
+-- Warning
    +-- DeprecationWarning
    +-- PendingDeprecationWarning
    +-- RuntimeWarning
    +-- SyntaxWarning
    +-- UserWarning
    +-- FutureWarning
    +-- ImportWarning
    +-- UnicodeWarning
    +-- BytesWarning
    +-- ResourceWarning
```

CAPÍTULO 8

Funções

Algumas tarefas precisam ser executadas diversas vezes para cumprir uma tarefa. Da mesma forma, alguns blocos ou trechos de código, que executam uma tarefa precisam ser usados mais de uma vez. Com o tempo, isso pode gerar repetição de código ou gerar um código de difícil leitura. Para resolver este problema existem as funções!

DEFININDO FUNÇÕES

Neste material já foram usadas diversas funções como **len()**, **print()**, **max()** e **type()**. Todas elas são nativas do Python e **executam uma tarefa** sempre que chamadas.

Em algum ponto pode ser interessante transformar um código recorrente, e que é executado muitas vezes, em uma função. Por exemplo, a função abaixo imprime uma mensagem dizendo "Python é incrível!":

```
def funcao_imprime_a_verdade():  
  
    print("Python é incrível!")
```

Desta forma, sempre que chamada, a função irá imprimir a frase. Para chamar uma função basta apenas declarar seu nome em um código após a mesma ser definida:

```
funcao_imprime_a_verdade()
```

Esta função pode ser usada diversas vezes no código como visto abaixo:

```
def funcao_imprime_a_verdade():  
  
    print("Python é incrível!")  
  
print("Oi, tudo bem?")  
  
funcao_imprime_a_verdade()  
  
print("Você só sabe falar isso?")  
  
funcao_imprime_a_verdade()
```

```
print("Por favor, pare!")

funcao_imprime_a_verdade()
```

Como observado, para definir uma função deve-se usar a diretiva **def**, os parênteses junto aos dois pontos ":" no final.

PARÂMETROS EM FUNÇÕES

Ao invés de simplesmente imprimir uma frase, seria interessante que uma função **executasse** uma tarefa **a partir de uma variável**, como por exemplo, o cálculo de imposto de renda a partir de um salário. Para isso, a função precisa de parâmetros.

Para entender a passagem de parâmetros, pode-se observar o código abaixo que faz o cálculo de imposto de renda:

```
def imposto_de_renda(salario):

    if(salario<1903.98):

        print("Isento. Salário final: "+str(salario))

    elif(salario<2826.65):

        salario = salario * (1-0.075)

        print("7.5%. Salário final: "+str(salario))

    elif(salario<3751.05):

        salario = salario * (1-0.15)

        print("15%. Salário final: "+str(salario))

    elif(salario<4664.68):

        salario = salario * (1-0.225)

        print("22.5%. Salário final: "+str(salario))

    else:

        salario = salario * (1-0.275)
```



```
print("27.5%. Salário final: "+str(salario))

imposto_de_renda(1500.00)

imposto_de_renda(4500.00)

imposto_de_renda(8900.00)
```

Desta forma sempre que a função for acessada e um parâmetro for passado o cálculo será executado. O exemplo anterior seria mais interessante se retornasse o valor do cálculo para que possa ser armazenado em uma variável. Para isso as funções contam com a diretiva return. o código abaixo demonstra seu uso:

```
def imposto_de_renda(salario):

    if(salario<1903.98):

        return salario

    elif (salario<2826.65):

        salario = salario * (1-0.075)

        return salario

    elif (salario<3751.05):

        salario = salario * (1-0.15)

        return salario

    elif (salario<4664.68):

        salario = salario * (1-0.225)

        return salario

    else:

        salario = salario * (1-0.275)
```

```
        return salario

salario_liquido = imposto_de_renda(1500.00)

salario_liquido = imposto_de_renda(4500.00)

salario_liquido = imposto_de_renda(8900.00)
```

É possível também definir funções com mais de um parâmetro. Para isso basta apenas separá-los por vírgula. Como exemplo é possível supor que queira-se gerar uma mensagem de parabéns passando os parâmetros de idade e nome de um funcionário:

```
def frase_aniversario(funcionario, idade):

    mensagem = funcionario + " completa hoje " +
str(idade) + " anos!"

    return mensagem

frase = frase_aniversario("Jonas", 21)

print(frase)
```

Um problema surge quando existem muitos parâmetros, afinal, é difícil passar todos na ordem certa. Por exemplo deseja-se imprimir as características de um Tanque de Guerra em uma função:

```
def tanque_de_guerra(nome,

                    tipo,

                    local,

                    utilizadores,

                    fabricante,

                    custo_unitario,
```

```
        peso,  
  
        tripulacao,  
  
        blindagem,  
  
        armamento_principal,  
  
        armamento_secundario,  
  
        motor,  
  
        suspensao,  
  
        alcance_operacional,  
  
        velocidade  
  
    ):  
  
    print("Nome: "+str(nome))  
  
    print("Tipo: "+str(tipo))  
  
    print("Local de origem: "+str(local))  
  
    print("Utilizadores: "+str(utilizadores))  
  
    print("Fabricante: "+str(fabricante))  
  
    print("Custo unitário: "+str(custo_unitario))  
  
    print("Peso: "+str(peso))  
  
    print("Tripulação: "+str(tripulacao))  
  
    print("Blindagem do veículo: "+str(blindagem))  
  
    print("Armamento primário:  
"+str(armamento_principal))
```

```
print("Armamento secundário:
"+str(armamento_secundario))

print("Motor: "+str(motor))

print("Suspensão: "+str(suspensao))

print("Alcance Operacional:
"+str(alcance_operacional))

print("Velocidade: "+str(velocidade))
```

Desta forma basta chamar a função passando os parâmetros:

```
tanque_de_guerra("T-14 Armata",
                "Carro de combate principal",
                "Russia",
                "Forças armadas russas",
                "Uralvagonzavod",
                "7,4 milhões de dólares",
                "48 toneladas",
                3,
                "44S-sv-Sh",
                "1 canhão 2A82-1M de 125mm",
                "1 metralhadora Kord (6P49) e 1 metralha
dora PKT (6P7K)",
                "ChTZ 12H360 (A-85-
3A) (diesel) 1 500 h.p.",
                "Barra de torção",
                "500 km +",
                "80-90 km/h"
                )
```

Nota-se uma dificuldade na identificação e significado de cada parâmetro. Para estes casos, é interessante usar parâmetros nomeados. Parâmetros nomeados permitem que seja possível o envio fora da ordenação como visto no código abaixo:

```
tanque_de_guerra(nome="T-14 Armata",
                 tipo="Carro de combate principal",
                 local="Russia",
                 velocidade="80-90 km/h",
                 utilizadores="Forças armadas russas",
                 armamento_principal="1 canhão 2A82-
1M de 125mm",
                 blindagem="44S-sv-Sh",
                 fabricante="Uralvagonzavod",
                 custo_unitario="7,4 milhões de dólares",
                 peso="48 toneladas",
                 tripulacao=3,
                 suspensao="Barra de torção",
                 armamento_secundario="1 metralhadora Kor
d (6P49) e 1 metralhadora PKT (6P7K)",
                 motor="ChTZ 12H360 (A-85-
3A) (diesel) 1 500 h.p.",
                 alcance_operacional="500 km +"
                 )
```

Python ainda permite passar, como parâmetro, dados estruturados como listas, tuplas e dicionários. Suponha uma festa que tem convidados e deseja-se imprimir o nome destes convidados:

```
def imprime_lista(convidados):

    for convidado in convidados:

        print(convidado)

convidados = [

    "Marcos",

    "Júlio",
```

```
        "Natan",  
        "Pablo"  
    ]  
  
    imprime_lista(convidados)
```

O mesmo pode ser replicado para dicionários e tuplas.

Em alguns casos, existem valores que são sempre padrões e que possuem apenas exceções como por exemplo um carro. No geral têm 4 rodas e um motor mas diferem muito em transmissão e obviamente possuem nomes diferentes. Para evitar a necessidade de sempre ter que passar valores de número de rodas e motor, pode-se definir um valor padrão para eles como visto abaixo:

```
def caracteristicas_carro(nome, transmissao,  
    num_rodas=4, motor=1):  
  
    print(nome)  
  
    print(num_rodas)  
  
    print(motor)  
  
    print(transmissao)  
  
caracteristicas_carro(nome="Gol", transmissao="Manual")
```

É possível notar que os parâmetros de número de rodas e motor não foram definidos e mesmo assim foram impressos. Entretanto, quando existe um valor padrão e ele for definido como visto abaixo no parâmetro motor, o valor que é passado para a função é usado e o valor padrão é desprezado.

```
caracteristicas_carro(nome="Gol", transmissao="Manual",  
    motor=2)
```

EXERCÍCIO PROPOSTO

Lembrar do aniversário de todo mundo não é uma tarefa fácil. Porém, se tiver uma lista contendo o nome e a data de aniversário, é possível automatizar esta tarefa! Para reforçar os conceitos de função aprendidos desenvolva o seguinte algoritmo:

“Desenvolver um algoritmo que contenha uma lista com o nome e a data de nascimento dos funcionários de uma empresa. Cada vez que for executado, o algoritmo deve enviar um email com uma mensagem que pode ser padrão ou personalizada para todos que estão fazendo aniversário”

Complemento:

E-mails geralmente bloqueiam ações de algoritmos a fim de evitar spam. Para resolver este problema, basta permitir nas configurações do seu e-mail:

No gmail, deve-se acessar as Configurações de Contas do Google no link <https://myaccount.google.com/lesssecureapps?pli=1> e ativar a permissão:

← Acesso a app menos seguro

Alguns apps e dispositivos usam tecnologias de login menos seguras, o que deixa sua conta vulnerável. Você pode desativar o acesso desses apps, o que recomendamos, ou ativá-lo se optar por usá-los apesar dos riscos. O Google desativará essa configuração automaticamente se ela não estiver sendo usada. [Saiba mais](#)

Permitir aplicativos menos seguros: ATIVADA



O código para envio com gmail fica assim:

```
import smtplib

endereco_email = "seuemail@gmail.com"

senha_email = input("Digite sua senha para continuar:
")

with smtplib.SMTP('smtp.gmail.com', 587) as smtp:
```

```
smtp.ehlo()

smtp.starttls()

smtp.ehlo()

smtp.login(endereco_email, senha_email)

assunto = "Curso de Programação em Python"

conteudo = "Email enviado com Python <3"

mensagem = f"Subject: {assunto}\n\n{conteudo}"

smtp.sendmail(endereco_email, endereco_email,
mensagem)
```

No caso do Outlook, geralmente não é preciso fazer nenhuma espécie de configuração e o código fica como o descrito abaixo:

```
import smtplib

endereco_email = "seuemail@outlook.com"

senha_email = input("Digite sua senha para continuar:
")

with smtplib.SMTP('smtp-mail.outlook.com', 587) as
smtp:
```



```
smtp.ehlo()

smtp.starttls()

smtp.ehlo()

smtp.login(endereco_email, senha_email)


assunto = "Curso de Programação em Python"

conteudo = "Email enviado com Python <3"

mensagem = f"Subject: {assunto}\n\n{conteudo}"

smtp.sendmail(endereco_email, endereco_email,
mensagem)
```

EXERCÍCIOS COMPLEMENTARES

1. Uma empresa, preocupada com a saúde de seus funcionários decidiu fazer um acompanhamento médico de peso. Suponha a existência de uma lista de dicionários contendo as informações médicas dos funcionários como o modelo abaixo:

```
[
    {
        "nome": "José",
        "peso": 80,
        "altura": 1.89
    },
    {
        "nome": "Maria",
```

```

        "peso":55,
        "altura":1.75
    },
]

```

O cálculo do IMC respeita a seguinte fórmula:

$$\text{IMC} = \frac{\text{Peso em kg}}{\text{Altura}^2 \text{ (Metros)}}$$

O algoritmo deve, com as informações do peso e da altura, checar a situação de cada funcionário da lista e imprimir a situação conforme a tabela abaixo:

IMC	Estado Nutricional
< 10	Desnutrição Grau V
de 10 a 12,9	Desnutrição Grau IV
de 13 a 15,9	Desnutrição Grau III
de 16 a 16,9	Desnutrição Grau II
de 17 a 18,4	Desnutrição Grau I
de 18,5 a 24,9	Normal
de 25 a 29,9	Pré-obesidade
de 30 a 34,5	Obesidade Grau I
de 35 a 39,9	Obesidade Grau II
> 40	Obesidade Grau III

2. Use o conceito de funções para criar mensagens aleatórias motivacionais para uma lista de pessoas. O algoritmo deve conter:

- uma função que sorteia um número aleatório
- uma função que recebe o nome do funcionário e um número aleatório e imprima uma frase no seguinte formato:

```

"{nome do funcionário} lembre-se: {mensagem
motivacional}"

```

CAPÍTULO 9

Funções Lambda

Uma função lambda é usada para executar, geralmente, tarefas rápidas e que não tem necessidade de operações muito complexas. O uso é de gosto pessoal e de estilo do programador.

CRIANDO UMA FUNÇÃO LAMBDA

Para definir uma função lambda usa-se a seguinte sintaxe:

```
lambda (argumentos): (operação sobre os argumentos)
```

Como exemplo, imagine a um hotel que deseja proporcionar um desconto de 10% nas reservas antecipadas. Uma função poderia ser definida para isso:

```
def desconto_de_10(valor_entrada):  
    valor_retorno = valor_entrada * 0.9  
    return valor_retorno
```

Porém, com uma função lambda, a função ficaria em uma linha:

```
funcao_desconto = lambda valor_entrada:  
valor_entrada*0.9  
  
valor_entrada = 100  
  
valor_descontado = funcao_desconto(valor_entrada)  
  
print(valor_descontado)
```

Uma função lambda pode ter diversos parâmetros. Por exemplo, deseja-se pegar o primeiro e o último nome de uma pessoa, formatar e devolver o nome completo formatado. A função abaixo pode traduzir esta ação:

```
def nome_completo(primeiro_nome,segundo_nome):  
  
    primeiro_nome = primeiro_nome.strip().title()  
  
    segundo_nome = segundo_nome.strip().title()  
  
    nome_completo = primeiro_nome + " " + segundo_nome
```

```
return nome_completo
```

Com uma função lambda o processo pode ser reduzido para uma linha:

```
nome_completo = lambda primeiro_nome, segundo_nome:
primeiro_nome.strip().title() + " " +
segundo_nome.strip().title()
```

FUNÇÃO MAP

Neste material já foi visto que é possível usar a função **for()** para iterar, ou seja, percorrer todos os elementos de uma lista, uma tupla ou um dicionário. Para auxiliar e tornar o código mais limpo existe a função **map()**.

Para usá-la basta ter uma função que recebe um valor e uma lista.

Por exemplo, suponha uma corretora que precise converter valores de dólar para real. Para isso uma função como descrita abaixo pode ser criada:

```
def dolar_para_real(valor_em_dolar):

    taxa_cambial = 4

    valor_em_real = valor_em_dolar * taxa_cambial

    return valor_em_real
```

Ao invés de usar ela em um loop foi como visto abaixo:

```
lista_de_dolares = [10,20,34.55,99.99]

lista_de_reais = []

for dolar in lista_de_dolares:

    valor_real = dolar_para_real(dolar)

    lista_de_reais.append(valor_real)

print(lista_de_reais)
```

A lista pode ser usada em uma função **map()**:

```
lista_de_dolares = [10,20,34.55,99.99]

lista_de_reais = []

lista_de_reais = map(dolar_para_real, lista_de_dolares)

print(list(lista_de_reais))
```

Pode não parecer muito, mas a legibilidade que o código ganha com o passar do tempo torna a função **map()** muito apreciada por programadores mais experientes.

O algoritmo anterior pode ser mais reduzido ainda com o emprego das funções anônimas, ou como são conhecidas, funções lambda.

As funções lambda podem ser usadas em conjunto com a função **map()** como no algoritmo abaixo:

```
lista_de_dolares = [10,20,34.55,99.99]

lista_de_reais = map(lambda dolar: dolar*4,
lista_de_dolares)

print(list(lista_de_reais))
```

FILTER

Inúmeras vezes o emprego de filtros se torna necessário. No dia a dia é fácil ouvir “Quais são as contas **acima de R\$300?**”, “Ainda tem quais produtos do **tipo X?**” ou “O Palmeiras **tem mundial?**”.

Para responder estas perguntas, a função **filter()** pode ser usada. O primeiro argumento de uma função **filter()** é uma outra função. Esta função

deve retornar True ou False. Caso ela retorne **True**, o valor será **mantido na lista**, caso ele retorne **False**, o valor será **eliminado**.

Para resolver o problema das contas acima de R\$300 uma lista de contas é passada para uma função lambda que verifica se o valor é maior que 300. Como resposta, é possível receber uma lista filtrada:

```
contas = [100,50,300,30000,120000,35,45]

contas_acima = filter(lambda conta: conta>300, contas)

print(list(contas_acima))
```

Outro problema abordado foi a checagem para ver se ainda tem produtos do tipo X?:

```
produtos = [

    {"Nome":"Macaxeira", "Tipo": "Y", "Quantidade":50},

    {"Nome":"Acarajé", "Tipo": "X", "Quantidade":30},

    {"Nome":"Tapioca", "Tipo": "X", "Quantidade":340},

    {"Nome":"Berinjela", "Tipo": "Y", "Quantidade":50},

]

produtos_x = filter(lambda produto:
produto["Tipo"]=="X", produtos)

print(list(produtos_x))
```

Por fim, para verificar se o Palmeiras tem mundial o código abaixo pode ser usado:

```
titulos_palmeiras = [

    {"Título":"Campeão Mundial", "Competição": "Futebol de Mesa", "Tipo": "Mundial"},
```

```
        {"Título": "Campeão Mundial ANAC", "Competição":  
"Ginástica Aeróbica", "Tipo": "Mundial"},  
  
        {"Título": "Vice Campeão Mundial de Bocha  
Individual", "Competição": "Bocha", "Tipo": "Vice-Mundial"},  
  
    ]  
  
    mundiais = filter(lambda titulo:  
titulo["Tipo"]=="Mundial", titulos_palmeiras)  
  
    print(list(mundiais))
```

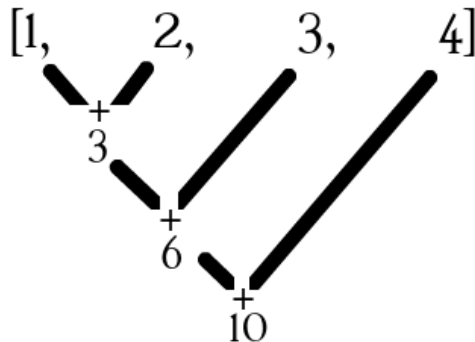
REDUCE

Uma função interessante, que tem um uso controverso e discutido por diversos programadores é a função **reduce()**. A discussão se baseia na filosofia explícita do Python, já que a função reduce se torna muito implícita.

Entretanto, ela executa uma operação par a par de uma lista o que torna diversas situações muito simples de serem resolvidas. Tomando como exemplo o código abaixo:

```
from functools import reduce  
  
numeros = [1,2,3,4]  
soma = lambda num1,num2: num1+num2  
resultado = reduce(soma,numeros)  
  
print(resultado)
```

Um loop da seguinte forma é executado:



FUNÇÕES ALL E ANY

Suponha que uma lista precisa ter uma condição checada. Por exemplo, se **todos** os seus elementos **sejam números pares**, ou **todos** os itens da loja tenham um valor **menor que R\$200** ou ainda se **todos** os produtos de uma loja estão com a **validade em dia**.

No primeiro problema, deseja-se saber se todos os elementos de uma lista são pares. Para testar se um número é par basta fazer a operação de módulo e checar se o seu resto é 0. Para aplicar isso à uma lista o código abaixo pode ser usado:

```
def sao_pares(lista):
    for item in lista:
        if(item%2!=0):
            return False
    return True
```

Para executar a mesma tarefa com a função **all()** pode-se usar:

```
lista = [2,4,6,8,10]

print(all((item%2==0) for item in lista))
```


De forma análoga à função **all()** a função **any()** retorna **True** ao atender uma condição, com a sutil diferença de fazer isso se pelo menos um item da lista for verdadeiro. Como exemplo:

```
lista = [1,3,4,5,7]

print(any((item%2==0) for item in lista))
```

Para saber todos os itens da loja tenham um valor menor que R\$200 pode-se usar:

```
lista_de_precos = [100,30,40,50,70]

print(all((preco<200) for preco in lista_de_precos))
```

Ainda é possível checar se existe pelo menos um produto abaixo de R\$200 usando a função **any()**:

```
lista_de_precos = [1000,399,480,199,99.90]

print(any((preco<200) for preco in lista_de_precos))
```

Para verificar se todos os produtos de uma loja estão com a validade em dia é necessário comparar datas usando o módulo **datetime** em conjunto com a função **any()** como visto abaixo:

```
from datetime import datetime as dt

lista_de_produtos = [

    {

        "Nome": "Vassoura de aço",

        "validade": dt.strptime("01/01/2999",

'%d/%m/%Y')
```

```
        },  
  
        {  
  
            "Nome": "Travesseiro de Pedra",  
  
            "validade": dt.strptime("01/01/9999",  
'%d/%m/%Y')  
  
        },  
  
        {  
  
            "Nome": "Havaiana de Pau",  
  
            "validade": dt.strptime("09/08/2019",  
'%d/%m/%Y')  
  
        },  
  
    ]  
  
  
    data_atual = dt.now()  
  
  
    tem_vencidos = any((produto["validade"] > data_atual) for  
produto in lista_de_produtos)  
  
  
    print(tem_vencidos)
```

CAPÍTULO 10

Trabalhando com Módulos

Quando se desenvolve em Python, é comum ficar maravilhado com o poder de seus módulos.

Além dos módulos padrão, é possível instalar uma gama gigante de módulos desenvolvidos por terceiros ou que não estejam disponíveis na “pilha padrão” do Python.

O GERENCIADOR DE PACOTES PIP

A forma mais fácil para este uso é o **pip**. O pip é um **gerenciador** de “**pacotes**”, ou seja, ele fará o gerenciamento dos módulos.

Como ele não é um programa padrão, é necessário instalá-lo. Para isso, primeiramente deve-se baixar o arquivo disponível em **<https://pip.pypa.io/en/stable/installing/>** chamado **get-pip.py**. Para baixar basta clicar com o botão direito em cima do link “get-pip.py” e após selecionar a opção “Salvar link como...”

Installation

Do I need to install pip?

pip is already installed if you are using Python 2 >=2.7.9 or Python 3 >=3.4 downloaded from the official Python website or if you are working in a [Virtual Environment](#) created by [virtualenv](#) or [pyvenv](#). Just make sure you have [pip](#).

Installing with get-pip.py

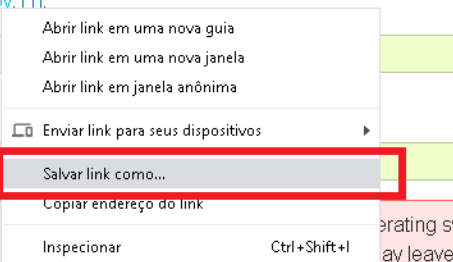
To install pip, securely download [get-pip.py](#):

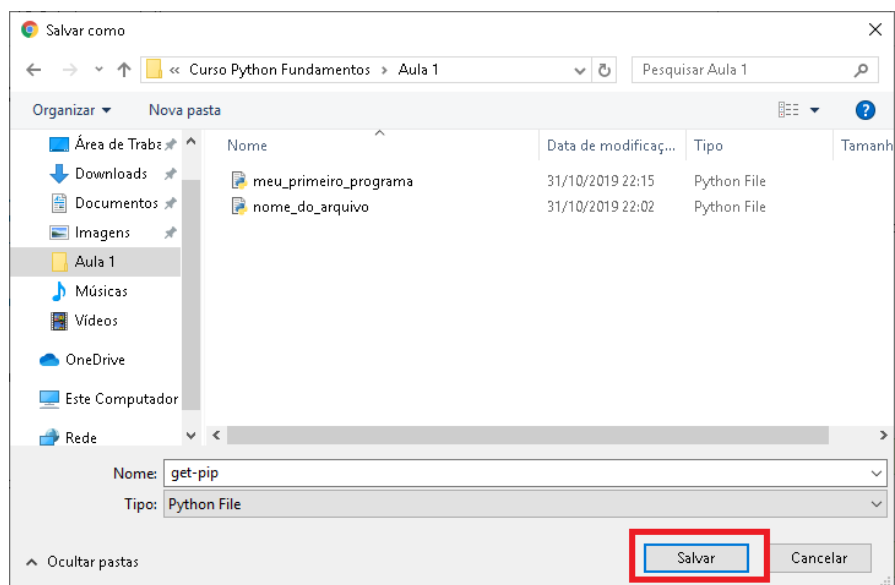
```
curl https://bootstrap.pypa.io/get-pip.py
```

Then run the following:

```
python get-pip.py
```

Warning: Be cautious if you are using another package manager. `get-pip.py` may leave your system in an inconsistent state.





Na janela de gravação clicar em salvar.

Este arquivo contém todos os comandos Python que farão a instalação correta do pip. Para instalar basta rodar o arquivo como um simples arquivo Python.

Para verificar se houve sucesso na instalação, basta abrir um terminal e digitar “pip --version”.

```
pip --version
```

Para instalar um pacote, é necessário apenas saber o nome e usar a seguinte sintaxe:

```
pip install {nome do pacote}
```

Como exemplo, é interessante instalar o módulo **matplotlib**. Este módulo permite a geração de gráficos avançados de uma forma bem simples.

Para instalar basta usar o comando:

```
pip install matplotlib
```

Para verificar quais são os pacotes instalados no sistema basta usar o utilitário freeze:

```
pip freeze
```

Na listagem é possível ver o pacote matplotlib disponível.

Agora já é possível usar as funcionalidades desta biblioteca. Por exemplo:

```
from matplotlib import pyplot as plt

plt.plot([1, 2, 3, 4])

plt.ylabel('Alguns números')

plt.show()
```

Da mesma forma, é possível instalar uma infinidade de outros pacotes.

Para desinstalar um pacote, é só usar a seguinte sintaxe:

```
pip uninstall {nome do pacote}
```

AMBIENTES VIRTUAIS

Existe, entretanto, uma famosa frase de um personagem de filme que diz *“Com grandes poderes, vem grandes responsabilidades”*.

Dessa forma, se alguns cuidados não forem tomados, o ambiente de desenvolvimento ficará lotado de bibliotecas.

Isso quando não são usadas apenas algumas vezes e depois nunca mais.

Não apenas isso! Imagine o seguinte cenário: está se desenvolvendo uma aplicação, chamada “AplicacaoLegal” que utiliza um módulo chamado “moduloporreta” em sua versão 1.0.

Daí, algum tempo se passa e se dá início no desenvolvimento de uma outra aplicação, chamada “AplicacaoDepoisDaUltima”, que por coincidência utiliza o mesmo módulo (moduloporreta), mas em sua versão 2.0.

De forma inocente se faz o upgrade (pip install --upgrade moduloporreta) dessa biblioteca para sua versão 2.0 para poder continuar seu desenvolvimento.

Então, neste cenário, o que acontecerá caso se tente retomar o desenvolvimento da aplicação AplicacaoLegal (juntamente com o desenvolvimento da aplicação AplicacaoDepoisDaUltima)?

Caso se tenha instalado todas as suas bibliotecas no diretório padrão, apenas a aplicação AplicacaoDepoisDaUltima irá funcionar.

Ou seja, quando se desenvolve Python “globalmente” e não se **isola cada ambiente** de desenvolvimento, é possível ter conflitos entre versões de bibliotecas.

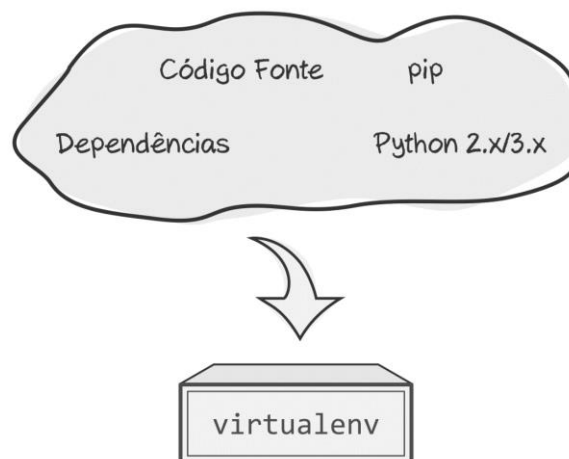
Outro ponto a se observar é na manutenção de um arquivo escrito em Python 2 e outro em Python 3.

A solução para todos os estes problemas chama-se **virtualenv**. Portanto, é necessário começar entendendo melhor do que se trata.

FUNCIONAMENTO DO VIRTUALENV

O funcionamento do virtualenv é realmente simples. Ele basicamente **cria uma cópia** de todos os diretórios necessários para que um programa Python seja executado, isto inclui:

- As bibliotecas comuns do Python (standard library);
- O gerenciador de pacotes pip;
- O próprio binário (interpretador) do Python (Python 2.x/3.x);
- As dependências que estiverem no diretório site-packages;
- Seu código fonte descrevendo sua aplicação.



Assim, ao instalar uma nova dependência dentro do ambiente criado pelo virtualenv, ele será colocado no diretório site-packages relativo à esse ambiente, e não mais globalmente. Assim, só esse ambiente enxergará essa dependência.

Para a instalação do **virtualenv** é necessário a utilização do **pip**.

```
pip install virtualenv
```

Para verificar se a instalação ocorreu bem usar o utilitário **freeze** do pip.

```
pip freeze
```

O virtualenv possui apenas um comando como visto:

```
virtualenv [opções] <nome_do_ambiente>
```

Esse comando cria um novo ambiente de desenvolvimento totalmente isolado.

Como exemplo, usaremos o nome “**venv**”. É importante comentar que o nome pode ser definido, mas por convenção se usa a nomenclatura “venv”.

```
virtualenv venv
```

Após criar o novo ambiente, deve-se ativá-lo. Para isso é necessário ir até à pasta que está o arquivo usando o comando **cd** como no código abaixo:

```
cd venv/Scripts
```

E então ativá-lo com o comando:

```
activate
```

O comando **activate** lê um arquivo e executa os comandos contidos ali.

Para desativar um ambiente virtual é só usar o comando **deactivate**:

```
deactivate
```

Para remover tudo que foi instalado, retornando ao estado anterior do sistema (sem o virtualenv) basta apagar a pasta criada.

Fazendo isso, o ambiente virtual é desfeito e tudo que foi copiado para este ambiente é apagado.

O uso do virtualenv traz muitas vantagens e facilidades quando se desenvolve em Python, principalmente na separação de ambientes de desenvolvimento.

Também ajuda muito na solução de conflitos entre versões de uma biblioteca e é essencial para o desenvolvimento de aplicações utilizando diferentes versões do Python em uma mesma máquina (sem ter que recorrer a máquinas virtuais, por exemplo).

CAPÍTULO 11

Trabalhando com Arquivos

Toda empresa precisa de arquivos. Rotineiramente é necessário abrir, criar ou editar uma planilha, um documento ou uma apresentação. Algumas destas tarefas são repetitivas e consomem muito tempo.

Para auxiliar nas tarefas monótonas ou repetitivas envolvendo estes arquivos existem módulos Python nativos e feitos por terceiros que dão um poder incrível para o gerenciamento destes arquivos.

GERENCIAMENTO DE ARQUIVOS CSV

Arquivos CSV (do inglês comma-separated values) são arquivos texto utilizados para armazenar dados. Cada linha é uma instância do conjunto de dados. Os campos são separados por vírgulas. Caso haja uma vírgula no conteúdo de um campo, o conteúdo inteiro é delimitado por aspas duplas. Geralmente, a primeira linha do arquivo CSV é composta pelo cabeçalho dos campos e, nas demais linhas, estão o conteúdo dos dados.

Abaixo temos um exemplo de arquivo CSV. A primeira linha é a do cabeçalho e as duas outras contém os dados do arquivo.

```
Ano,Fabricante,Modelo,Comprimento
```

```
1997,Ford,E350,2.34
```

```
2000,Mercury,Cougar,2.38
```

Este arquivo pode ser traduzido na seguinte tabela:

Ano	Fabricante	Modelo	Comprimento
1997	Ford	E350	2.34
2000	Mercury	Cougar	2.38

LENDO UM ARQUIVO CSV

Um arquivo CSV pode ser lido importando o módulo csv. Segue abaixo um exemplo de leitura de um arquivo CSV.


```
import csv

csv_file = open('dados.csv')

csv_reader = csv.reader(csv_file)

for linha in csv_reader:

    print(linha)
```

O Script acima lê e imprime todas as linhas do csv “dados.csv”.

Um arquivo CSV pode ser lido diretamente para um dicionário. O código abaixo mostra isto.

SALVANDO DADOS EM ARQUIVOS CSV

```
import csv

csv_arq = open('dados.csv', newline='')

csv_dict = csv.DictReader(csv_arq)

for linha in csv_dict:

    print(f'{linha["Ano"]} - {linha["Fabricante"]} {linha["Modelo"]}')

```

ESCREVENDO EM UM ARQUIVO CSV

Para escrever dados para um arquivo CSV, abre-se um arquivo para escrita (modo ‘w’, do inglês write que significa escrever).

O arquivo aberto é passado como parâmetro para o método csv.writer.

O método **writerow** escreve uma linha de dados no arquivo CSV de saída como visto no código abaixo:

```
import csv
```

```
csvfile = open('dados_escrita.csv', 'w', newline='')
#Arquivo que será criado

arquivo_criado = csv.writer(csvfile, delimiter=',')

arquivo_criado.writerow(['Ano', 'Fabricante', 'Modelo',
'Comprimento', 'Velocidade Máxima'])

arquivo_criado.writerow(['1970', 'Plymouth',
'Superbird', '5.613', '320'])

arquivo_criado.writerow(['2019', 'Chevrolet', 'Camaro
SS', '4.784', '248'])
```

É possível também escrever um arquivo CSV diretamente de um dicionário. O código abaixo mostra um exemplo usando o método **DictWriter**:

```
import csv

csv_file = open('dados_escrita2.csv', mode='w',
encoding='utf-8', newline='')

fieldnames = ['nome', 'departamento', 'mes_aniv']

arquivo_criado = csv.DictWriter(csv_file,
fieldnames=fieldnames)

arquivo_criado .writeheader()

arquivo_criado .writerow({'nome': 'João Silva',
'departamento': 'Contabilidade', 'mes_aniv': 'novembro'})

arquivo_criado .writerow({'departamento':
'Informática', 'nome': 'Catarina Andrade', 'mes_aniv':
'março'})
```

GERENCIAMENTO DE ARQUIVOS XLSX

Planilhas estão presentes no dia a dia na maioria das pessoas. Isso porque planilhas são a forma mais fácil de armazenar dados.

Com Python, é possível gerenciar planilhas do Excel de forma surpreendente.

LEITURA DE PLANILHAS

Para trabalhar com arquivos excel (ou qualquer outro .xls ou .xlsx) o módulo **openpyxl** pode ser usado. Abaixo segue a leitura de uma planilha e impressão das células "A2" e "B2" da aba "Dados dos Funcionários":

```
import openpyxl

planilha = openpyxl.load_workbook("Funcionários.xlsx")

aba = planilha["Dados dos Funcionários"]

print(aba["A1"].value)

print(aba["B1"].value)
```

EDIÇÃO DE VALORES

Para editar o valor de uma célula, de forma semelhante a uma variável, basta atribuir um novo valor e salvar as edições na planilha. No exemplo abaixo, a célula "A2" tem seu valor editado e então a planilha é salva "Funcionários.xlsx".

```
import openpyxl

planilha = openpyxl.load_workbook("Funcionários.xlsx")

aba = planilha["Dados dos Funcionários"]

print(aba["A2"].value)
```

```
aba["A2"] = "Jhonas Holtroup"

print(aba["A2"].value)

planilha.save("Funcionários.xlsx")
```

FILTRAGEM DE DADOS

Este módulo permite ainda salvar um arquivo com nome desejado. Por exemplo, deseja-se filtrar funcionários que recebam mais que R\$ 2.300,00. Para isso o código abaixo pode ser usado:

```
import openpyxl

planilha = openpyxl.load_workbook("Funcionários.xlsx")

aba = planilha["Dados dos Funcionários"]
aba_filtrada = planilha.create_sheet("Filtradas")

for row in aba.iter_rows(): #Filtra o salário
    if row[3].value != "Salário":
        if int(row[3].value) >= 2300:
            aba_filtrada.append((cell.value for cell in
row))

        else:
            aba_filtrada.append((cell.value for cell in
row))

planilha.remove(aba) #Remove a primeira Aba
```

```
planilha.save("Funcionários Filtrados.xlsx")
```

FORMATAÇÃO DE CÉLULAS

Existem diversos formatos que podem ser usados para estilizar uma célula da planilha.

As edições mais comuns em uma planilha são as de texto. Abaixo segue um código que demonstra as opções de tamanho, estilo e fonte que podem ser obtidos:

```
import openpyxl

planilha = openpyxl.load_workbook("Funcionários.xlsx")
aba = planilha["Dados dos Funcionários"]

for linha in aba["A1:D1"]:
    for celula in linha:
        fonte = openpyxl.styles.Font(name='Calibri',
                                      size=20,
                                      bold=True,
                                      italic=False,
                                      vertAlign=None,
                                      underline='none',
                                      strike=False,
                                      color='000000')

        celula.font = fonte

planilha.save("typologia.xlsx")
```

As edições de fundo da célula também se tornam interessantes:

```
import openpyxl

planilha = openpyxl.load_workbook("Funcionários.xlsx")
aba = planilha["Dados dos Funcionários"]
```

```
for linha in aba["A1:D1"]:  
    for celula in linha:  
        preenchimento = openpyxl.styles.PatternFill(  
            start_color='FFFF00',  
            end_color='FFFF00',  
            fill_type='solid')  
  
        celula.fill = preenchimento  
  
planilha.save("tipologia.xlsx")
```

Formatos de números e moedas também estão disponíveis:

```
import openpyxl  
  
planilha = openpyxl.load_workbook("Funcionários.xlsx")  
aba = planilha["Dados dos Funcionários"]  
  
for linha in aba["D2:D400"]:  
    for celula in linha:  
        number_format = "R$ 0.00"  
        celula.number_format = number_format  
  
planilha.save("tipologia.xlsx")
```

Para mais estilos e informações consultar a documentação no link <https://openpyxl.readthedocs.io/en/stable/index.html>

GERENCIAMENTO DE ARQUIVOS DOCX

Arquivos .docx são documentos do Microsoft Word. O formato é praticamente universal em empresas no gerenciamento de arquivos de texto.

Para manipular arquivos .doc e .docx a linguagem Python possui um módulo chamado docx. Com ele é possível ler e editar arquivos .doc com maestria.

LEITURA DE ARQUIVOS

O exemplo abaixo faz a leitura e imprime o conteúdo dos três primeiros parágrafos do documento;

```
import docx

doc = docx.Document('agradecimento.docx') #Abrir um documento

print(doc.paragraphs[0].text)

print(doc.paragraphs[1].text)

print(doc.paragraphs[3].text)
```

A primeira linha obtém o arquivo já presente no mesmo diretório do Script. Dentre o métodos disponíveis está o paragraphs. Com ele é possível obter via index todas as propriedades de um parágrafo.

EDIÇÃO DE ARQUIVOS

De forma similar, a edição de documentos é bem simples, muito similar a edição do valor de uma variável. No código abaixo um novo valor é atribuído ao segundo parágrafo.

```
import docx

doc = docx.Document('agradecimento.docx') #Abrir um documento

print(doc.paragraphs[1].text) #Paragrafo antes da edição

doc.paragraphs[1].text = "Prezado Sr. José das Couves"

print(doc.paragraphs[1].text) #Paragrafo após a edição

doc.save("agradecimento.docx") #Salvar edições
```

É possível adicionar parágrafos com o método `add_paragraph` como no exemplo abaixo:

```
from docx import Document

doc = Document() #Cria um novo documento

primeiro_paragrafo = doc.add_paragraph("Primeiro Parágrafo")

segundo_paragrafo = doc.add_paragraph("Segundo Parágrafo")

doc.save("Inserção de Parágrafos.docx") #Salvar edições
```

LIMITAÇÕES DA BIBLIOTECA

Uma das limitações desta biblioteca é não poder escolher a posição de inserção do parágrafo. Para isso, é possível criar uma função e usar como no exemplo abaixo:

```
from docx import Document

from docx.text.paragraph import Paragraph

from docx.oxml.xmlchemy import OxmlElement

def insert_paragraph_depois(paragraph, text=None,
style=None):

    """Insere um paragrafo após um parágrafo específico"""

    new_p = OxmlElement("w:p")

    paragraph._p.addnext(new_p)
```



```
new_para = Paragraph(new_p, paragraph._parent)

if text:

    new_para.add_run(text)

if style is not None:

    new_para.style = style

return new_para


doc = Document() #Cria um novo documento


primeiro_paragrafo = doc.add_paragraph("Primeiro Parágrafo")


segundo_paragrafo = doc.add_paragraph("Segundo Parágrafo")


insert_paragraph_depois(primeiro_paragrafo, "Parágrafo Um e
meio")


doc.save("Inserção.docx") #Salvar edições
```

ESTILOS DE TEXTO

No Word, existem certos estilos de texto, estes estilos aplicam uma formatação pré-determinada no texto. Com este módulo os estilos têm o mesmo nome do word em inglês. Abaixo segue uma pequena tabela com alguns dos estilos e o que cada faz:

Estilo	Ação
Normal	Deixa o texto com as opções padrão do Word (Cambria, tamanho 11) com espaçamento de linha maior.
Body Text	Deixa o texto com as opções padrão do Word (Cambria, tamanho 11)
List Bullet	Lista não numerada de itens
Heading 1, Heading 2...	Faz o que a função Cabeçalhos 1, Cabeçalhos 2 do word faz.

Para usá-los basta adicionar em um parágrafo da seguinte forma:

```
from docx import Document

doc = Document() #Cria um novo documento

p1 = doc.add_paragraph("Normal Parágrafo")

p1.style = 'Normal'

p2 = doc.add_paragraph("Body Text Parágrafo")

p2.style = 'Body Text'

p3 = doc.add_paragraph("ListBullet Parágrafo")

p3.style = 'ListBullet'

p4 = doc.add_paragraph("Heading 1 Parágrafo")

p4.style = 'Heading 1'

p5 = doc.add_paragraph("Heading 2 Parágrafo")
```

```
p5.style = 'Heading 2'

doc.save("Inserção de Parágrafos Estilosos.docx")
#Salvar edições
```

Ainda é possível gerenciar os estilos de texto como negrito, itálico e sublinhado. Para isso o código abaixo proporciona uma demonstração:

```
from docx import Document

doc = Document() #Cria um novo documento

p1 = doc.add_paragraph("Parágrafo ") #Cria o parágrafo
trecho1 = p1.add_run("Negrito") #Adiciona um trecho ao parágrafo
trecho1.bold = True #Define o trecho como negrito

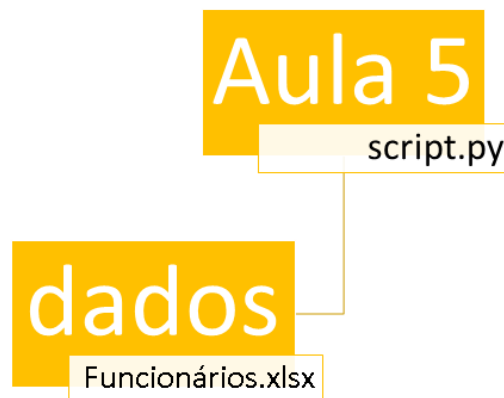
p2 = doc.add_paragraph("Parágrafo ")
trecho2 = p2.add_run("Itálico")
trecho2.italic = True

doc.save("Negrito e Itálico.docx") #Salvar edições
```

Para mais exemplos, formatações e informações consultar:
<https://python-docx.readthedocs.io/en/latest/user/styles-using.html>

NAVEGAÇÃO E MANIPULAÇÃO NO SISTEMA DE ARQUIVOS

Até o momento, os arquivos estavam na mesma pasta do Script. Mas e se elas estivessem em uma outra subpasta? Supondo a seguinte estrutura de pastas como o arquivo pode ser gerenciado?



Basta apenas mostrar o caminho do arquivo como visto abaixo na abertura de um arquivo Excel:

```
import openpyxl

planilha = openpyxl.load_workbook("dados/Funcionários.xlsx")
aba = planilha["Dados dos Funcionários"]

planilha.save("dados/Funcionários.xlsx")
```

GERENCIAMENTO DE VARIÁVEIS DE AMBIENTE E ARQUIVOS

Um módulo muito interessante, quase que imprescindível, quando se está editando arquivos é o módulo OS. Com ele todas as principais funcionalidades do sistema operacional como apagar arquivos, criar pastas e obter caminhos de arquivos estão disponíveis. É possível também obter informações como o nome do usuário, o nome da máquina e até mesmo a arquitetura do processador utilizado.

Abaixo seguem exemplos cotidianos do uso do módulo OS.

BUSCA DE PASTAS E ARQUIVOS

O módulo `os` permite ver todos os arquivos presentes em um determinado diretório, como por exemplo, o código abaixo que lista todos os arquivos e pastas presentes no "C:/":

```
import os

pasta = "C:/"
subpastas = os.listdir(pasta)

for arquivo in subpastas:
    print(arquivo)
```

VERIFICAR SE É UM ARQUIVO

É possível fazer a classificação em arquivos e diretórios com o método `isdir()`:

```
import os

for a in os.listdir("."):
    if os.path.isdir(a):
        print("Diretório: %s" % a)
    elif os.path.isfile(a):
        print("Arquivo: %s" % a)
```

VERIFICAR SE UM DIRETÓRIO EXISTE

Para verificar se um diretório existe o método **`exists()`** pode ser empregado:

```
import os

if os.path.exists("z"):
```

```
print("O diretório z existe.")

else:

    print("O diretório z não existe.")
```

CRIAÇÃO DE PASTAS

Para criar uma pasta se faz uso do método **makedirs()** como visto abaixo:

```
import os

diretorio = "Novos Arquivos"

diretorio_pai = "C:/"

pasta = os.path.join(diretorio_pai, diretorio)

os.makedirs(pasta)
```

REMOÇÃO DE ARQUIVOS

O método **remove()** remove um arquivo, como visto abaixo:

```
import os

os.remove("arquivo.docx")
```

Um outro empecilho que surge rotineiramente é a possibilidade de apagar arquivos com uma certa extensão ou condição. O algoritmo abaixo apaga todos os arquivos com uma lista de extensões desejada:

```
import os

diretorio_desejado = "C:/"

lista_arquivos = os.listdir(diretorio_desejado)

for arquivo in lista_arquivos:
    if arquivo.endswith((".zip", ".xlsx", ".docx")):
        os.remove(os.path.join(diretorio_desejado, arquivo))
```

EXERCÍCIO PROPOSTO

Partindo de um CSV que contém a seguinte estrutura:

Nome	CPF	Entrada na empresa	Mensagem do colaborador
Júlio	124.564.489-50	01/08/1995	"Gosto de trabalhar aqui!"
Ana	124.564.489-51	05/08/1999	"Lugar incrível!"

Criar:

- Um documento **.docx** (Word) contendo um convite para um evento especial que ocorrerá **30 dias após a geração do documento** para colaboradores com **4 anos ou mais de empresa**.
- Os colaboradores com menos de 4 anos devem ser registrados em uma planilha **.xlsx** (Excel).
- Os colaboradores que têm **menos de 1 ano de empresa**, devem ter a célula pintada de vermelho e o texto ficar branco.
- Os convites devem ser salvos em uma subpasta chamada "Convites" seguindo a seguinte regra de nomenclatura: "Nome do Funcionário.docx".
- Os demais devem ser salvos em uma subpasta chamada "Lista de Espera" em um arquivo chamado "Lista de Espera.xlsx".

EXERCÍCIOS COMPLEMENTARES

1. Como verificar se o arquivo "word.docx" existe na pasta atual (usar o módulo **os**)?
2. Qual a importância de se usar um ambiente virtual?
3. Crie um script Python que cada vez que for executado pega a data atual (**datetime**) e o nome do usuário (com o módulo **os** e o método **environ**) e insira numa planilha chamada **"logs.xlsx"**.
4. Instale o pacote python-pptx. Busque 10 imagens e coloque em uma pasta. Crie então uma apresentação de slides (**.pptx**) com Python que contenha todas elas. A ordem das imagens deve ser sorteada (**random**).

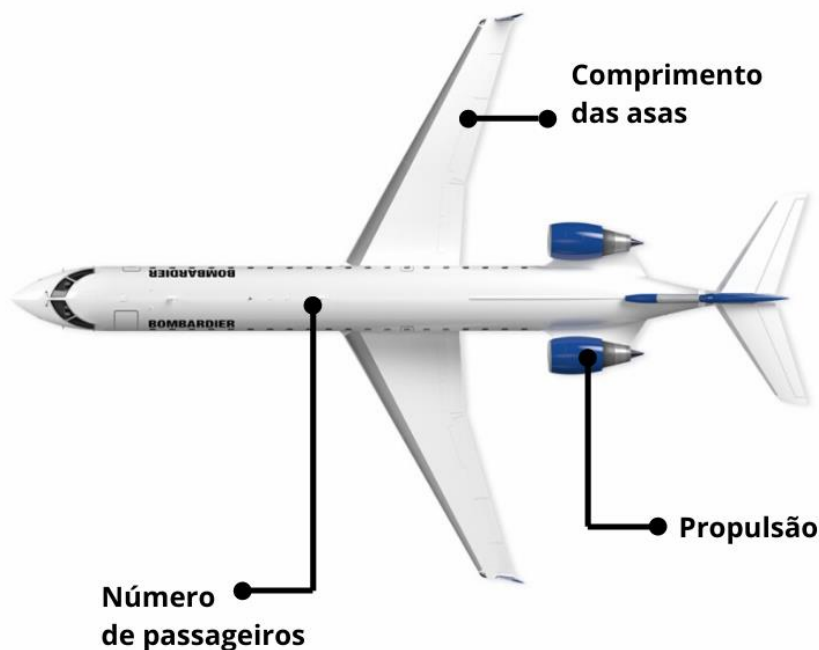
CAPÍTULO 12

Orientação a Objetos

Objetos são coisas tangíveis ao toque que executam algum papel no mundo físico, enfim eles existem. Desde os brinquedos de plástico de uma criança até um guindaste industrial possuem **características** como cor, tamanho ou peso.

Geralmente estes objetos são organizados para formar classes, ou seja, uma generalização de objetos com características semelhantes.

Em programação objetos não são tangíveis ao toque, mas da mesma forma possuem características, funções e interagem com o “ecossistema” de



alguma forma.

CLASSES

Em Python, para descrever um objeto, existe um tipo bloco chamado **class**.

Pensando em um **veículo**, pode-se atribuir características como **velocidade**, **autonomia** e **número de passageiros**.

Para descrever esta classe pode-se usar:

```
class Veiculo():  
  
    velocidade = 100  
  
    autonomia = 500  
  
    num_passageiros = 4
```

Para criar um objeto chamado carro que contenha essas características basta atribuir a ele a classe de forma similar à atribuição de uma variável:

```
carro = Veiculo()
```

De forma semelhante aos dicionários, é possível acessar a propriedades deste objeto criado:

```
print(carro.velocidade)
```

A forma com que o exemplo anterior foi construído mostra uma classe com diferentes atributos, todos fixos. Existem, entretanto, particularidades que objetos da mesma Classe possuem.

Por exemplo, um Bugatti Chiron é mais veloz que um Volkswagen Fusca comum. Classes permitem criar objetos **com características diferentes**.

Imagine agora que ao criar um objeto (iniciar ele) seja desejado passar as características deste objeto. Para esta necessidade, o método reservado **__init__** e o parâmetro **self**.

O método **__init__** faz com que os trechos de código contidos nele sejam executados assim que um objeto for criado a partir de uma classe. O exemplo abaixo mostra seu uso:

```
class Veiculo():  
    def __init__(self):  
        print("Objeto iniciado")  
  
carro = Veiculo()
```

O método **self** serve para referenciar o objeto dentro da classe. Ele é muito útil em diversas situações.

No exemplo dos carros, anteriormente explanado, é possível passar parâmetros de construção do objeto. Um Bugatti Chiron possui uma

velocidade máxima de 482 km/h. Já um Volkswagen Fusca 1600, ano 1994 possui uma **velocidade máxima** de 140 km/h. O código abaixo demonstra a criação destes dois objetos:

```
class Veiculo():  
    def __init__(self, nome, velo_maxima):  
        self.nome = nome  
        self.velo_maxima = velo_maxima  
  
    bugatti = Veiculo(nome="Bugatti Chiron", velo_maxima=482)  
    fusca = Veiculo(nome="Fusca 1600", velo_maxima=140)
```

MÉTODOS DE OBJETOS

Objetos podem ter métodos. Por exemplo, ao criar um objeto seria interessante ter um método que imprimisse as características do veículo quando assim fosse desejado.

O exemplo abaixo demonstra a criação de um método que imprime o nome, ano e velocidade de um carro:

```
class Veiculo():  
    def __init__(self, nome, ano, velo_maxima):  
        self.nome = nome  
        self.ano = ano  
        self.velo_maxima = velo_maxima  
  
    def imprime_caracteristicas(self):  
        print("Nome: "+self.nome)  
        print("Velocidade Máxima:  
"+str(self.velo_maxima)+"km/h")  
        print("Ano: "+str(self.ano))
```

```
superbird = Veiculo(nome="Plymouth Superbird",  
ano=1970, velo_maxima=320)  
  
superbird.imprime_caracteristicas()
```

É possível, de forma simples, atribuir valores padrão aos parâmetros das classes. Por exemplo, no ano, adicionar o valor "Indefinido" caso nenhum valor seja passado. O único cuidado que se deve ter é de colocar os parâmetros padrão após os parâmetros não padrão.

```
class Veiculo():  
  
    def __init__(self, nome, velo_maxima,  
ano="Indefinido"):  
  
        self.nome = nome  
  
        self.ano = ano  
  
        self.velo_maxima = velo_maxima  
  
bugatti = Veiculo(nome="Bugatti Chiron",  
velo_maxima=482)  
  
fusca = Veiculo(nome="Fusca 1600", velo_maxima=140)  
  
superbird = Veiculo(nome="Plymouth Superbird",  
ano=1970, velo_maxima=320)
```

ALTERAÇÃO DE PROPRIEDADES EM OBJETOS

É possível fazer a alteração dos atributos de um objeto. Para isso, basta atribuir à propriedade um novo valor, como por exemplo, alterar a velocidade máxima de um fusca para 150km/h como vista abaixo:

```
class Veiculo():  
  
    def __init__(self, nome, velo_maxima,  
ano="Indefinido"):
```

```
        self.nome = nome

        self.ano = ano

        self.velo_maxima = velo_maxima

    fusca = Veiculo(nome="Fusca 1600", ano=1994,
velo_maxima=140)

    print(fusca.velo_maxima)

    fusca.velo_maxima = 150

    print(fusca.velo_maxima)
```

É possível, de forma análoga a uma variável deletar as propriedades de um objeto usando a diretiva **del** como visto abaixo:

```
class Veiculo():

    def __init__(self, nome, velo_maxima,
ano="Indefinido"):

        self.nome = nome

        self.ano = ano

        self.velo_maxima = velo_maxima

    fusca = Veiculo(nome="Fusca 1600", ano=1994,
velo_maxima=140)

    print(fusca.velo_maxima)

    del fusca.velo_maxima

    print(fusca.velo_maxima)
```

Classes não podem ser vazias, mas, se deseja-se apenas defini-la e construir seus métodos depois é possível usar o método **pass**:

```
class Veiculo():  
  
    pass  
  
fusca = Veiculo()  
  
fusca.velo_maxima = 100  
  
print(fusca.velo_maxima)
```

HERANÇA

Veículos podem ter características como **nome**, uma **velocidade** e um **ano**. Um **tipo** de veículo é o **barco**. Um barco possui as características de um veículo, entretanto, possui **características próprias** como número de hélices e tipo de casco. O exemplo abaixo cria uma Classe Veículo e uma Classe Barco:

```
class Veiculo():  
  
    def __init__(self, nome, velo_maxima,  
ano="Indefinido"):  
  
        self.nome = nome  
  
        self.ano = ano  
  
        self.velo_maxima = velo_maxima  
  
class Barco():  
  
    def __init__(self, nome, velo_maxima, ano, casco,  
num_helices):  
  
        self.nome = nome  
  
        self.ano = ano  
  
        self.velo_maxima = velo_maxima  
  
        self.casco = casco
```

```
        self.num_helices = num_helices

cigarrete = Barco("50 Marauder GT S", 217, 2007, "Fibra
de Vidro", 12)
```

É fácil notar que existe duplicação de código, afinal, as classes possuem características semelhantes entre si. Imagine agora se uma classe de Carro e Avião for criada. Para evitar a duplicação de código e a complexibilidade de manutenção destes objetos, existe o conceito de Herança.

Digamos que a classe Barco herda as propriedades da classe Veículo, ou seja, traz para si todas as propriedades e métodos da classe Veículo.

Para fazer isso, basta passar a classe Veículo como parâmetros para a classe barco e chamar o método `__init__` da classe Veículo. O exemplo abaixo mostra como isso pode fazer:

```
class Veiculo():

    def __init__(self, nome, velo_maxima,
ano="Indefinido"):

        self.nome = nome

        self.ano = ano

        self.velo_maxima = velo_maxima

class Barco(Veiculo):

    def __init__(self, nome, velo_maxima, ano, casco,
num_helices):

        Veiculo.__init__(self, nome, velo_maxima, ano)

        self.casco = casco

        self.num_helices = num_helices
```

```
cigarrete = Barco("50 Marauder GT S", 217, 2007, "Fibra de Vidro", 12)
```

Para evitar a necessidade de usar o nome da classe mãe (Veiculo) com o método `__init__` bem como passar o parâmetro `self` existe o método **`super()`**. Com ele, automaticamente a classe de herança é atribuída como no exemplo abaixo:

```
class Veiculo():
    def __init__(self, nome, velo_maxima, ano="Indefinido"):
        self.nome = nome
        self.ano = ano
        self.velo_maxima = velo_maxima

class Barco(Veiculo):
    def __init__(self, nome, velo_maxima, ano, casco, num_helices):
        super().__init__(nome, velo_maxima, ano)
        self.casco = casco
        self.num_helices = num_helices

cigarrete = Barco("50 Marauder GT S", 217, 2007, "Fibra de Vidro", 12)
```

Para ser mais preciso, o **`super()`** é usado para fazer referência a superclasse, a classe mãe - no exemplo a classe **`Veiculo`**.

É possível, de forma análoga, estender a classe **`Barco`** para uma classe Lancha da seguinte forma:

```
class Veiculo():
    def __init__(self, nome, velo_maxima, ano="Indefinido"):
        self.nome = nome
        self.ano = ano
        self.velo_maxima = velo_maxima
```

```
class Barco(Veiculo):  
  
    def __init__(self, nome, velo_maxima, ano, casco,  
num_helices):  
  
        super().__init__(nome, velo_maxima, ano)  
  
        self.casco = casco  
  
        self.num_helices = num_helices  
  
class Lancha(Barco):  
  
    def __init__(self, nome, velo_maxima, ano, casco,  
num_helices):  
  
        super().__init__(nome, velo_maxima, ano, casco,  
num_helices)  
  
        self.tipo = "Lancha"  
  
cigarrete = Lancha("50 Marauder GT S", 217, 2007,  
"Fibra de Vidro", 12)
```

HERANÇA DE MÉTODOS

Ao estender uma classe, todos os métodos da classe mãe são automaticamente trazidos para a classe filha. Se na classe mãe existe um método chamado **imprime_caracteristicas** como no exemplo abaixo:

```
class Veiculo():  
  
    def __init__(self, nome, velo_maxima):  
  
        self.nome = nome  
  
        self.velo_maxima = velo_maxima
```



```
def imprime_caracteristicas(self):  
  
    print("Tipo: Veiculo")  
  
    print("Nome: "+self.nome)  
  
    print("Velocidade Máxima:  
"+str(self.velo_maxima))
```

Ao estender essa classe, o método é acessível em sua filha, como no exemplo abaixo:

```
class Barco(Veiculo):  
  
    def __init__(self, nome, velo_maxima, casco,  
num_helices):  
  
        super().__init__(nome, velo_maxima)  
  
        self.casco = casco  
  
        self.num_helices = num_helices  
  
    cigarrete = Barco("50 Marauder GT S", 217, "Fibra de  
Vidro", 12)  
  
    cigarrete.imprime_caracteristicas()
```

REESCRITA DE MÉTODOS

No exemplo anterior, as características de tipo de casco e número de hélices não foram impressas devido ao método não comportar isso.

É possível escrever um novo método chamado **imprime_caracteristicas_barco**, entretanto, no uso da classe como agora existem dois métodos (`imprime_caracteristicas` e `imprime_caracteristicas_barco`) fica difícil saber qual delas usar.

Nestes casos é possível fazer a sobrescrita de um método. Para isso, basta simplesmente reescrever novamente o método na classe filha, como visto em detalhes abaixo:

```
class Barco(Veiculo):
```

```
def __init__(self, nome, velo_maxima, casco,
num_helices):

    super().__init__(nome, velo_maxima)

    self.casco = casco

    self.num_helices = num_helices

def imprime_caracteristicas(self):

    print("Tipo: Barco")

    print("Nome: "+self.nome)

    print("Velocidade Máxima:
"+str(self.velo_maxima))

    print("Casco: "+str(self.casco))

    print("Hélices: "+str(self.num_helices))

cigarrete = Barco("50 Marauder GT S", 217, "Fibra de
Vidro", 12)

cigarrete.imprime_caracteristicas()
```

Quando se está trabalhando com objetos, a função **vars()** se torna muito interessante. Esta função transforma as propriedades de um objeto em um dicionário. Como por exemplo:

```
class Veiculo():

    def __init__(self, nome, velo_maxima, ano="Indefinido"):

        self.nome = nome

        self.ano = ano

        self.velo_maxima = velo_maxima
```

```
        fusca = Veiculo(nome="Fusca 1600", ano=1994, velo_maxima=140
    )

    print(vars(fusca))
```

OS MÉTODOS MÁGICOS

Existe uma função chamada **dir()** que mostra todos os métodos disponíveis em uma classe. Como no exemplo abaixo:

```
class Veiculo():

    def __init__(self, nome, velo_maxima,
ano="Indefinido"):

        self.nome = nome

        self.ano = ano

        self.velo_maxima = velo_maxima

    fusca = Veiculo(nome="Fusca 1600", ano=1994,
velo_maxima=140)

    print(dir(fusca))
```

Como saída:

```
['__class__', '__delattr__', '__dict__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__',
'__getattr__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__', '__module__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '__weakref__', 'ano',
'imprime_caracteristicas', 'nome', 'velo_maxima']
```

É possível notar que existe uma diversidade de métodos criados automaticamente. Estes métodos são herdados da classe **object**. Estes são os

métodos mágicos do Python, é possível notar que eles começam com dois traços sublinhados que são conhecidos como **dunders**.

Existe a possibilidade de sobrescrever todos eles. Como exemplo, o método `__str__`.

O funcionamento do método `__str__` se dá ao imprimir um objeto. Por exemplo ao imprimir:

```
print(fusca)
```

A saída será:

```
<__main__.Veiculo object at 0x00000290BD78B580>
```

Porém, se o método `__str__` for sobrescrito em uma classe, da seguinte forma:

```
class Veiculo():  
  
    def __init__(self, nome, velo_maxima,  
ano="Indefinido"):  
  
        self.nome = nome  
  
        self.ano = ano  
  
        self.velo_maxima = velo_maxima  
  
    def __str__(self):  
  
        return "Sou um Veículo chamado "+self.nome
```

Ao imprimir um objeto desta classe:

```
fusca = Veiculo(nome="Fusca 1600", ano=1994,  
velo_maxima=140)  
  
print(fusca)
```

A mensagem será:

Sou um Veículo chamado Fusca 1600

Existem outros métodos bem famosos como o método `__add__`. Com ele é possível dar a possibilidade de aplicação do operador “+” aos objetos.

Para ilustrar, suponha que a cada vez que somar um objeto fusca a uma variável, esta variável seja somada a velocidade do fusca. O resultado pode ser acompanhado abaixo:

```
class Veiculo():

    def __init__(self, nome, velo_maxima,
ano="Indefinido"):

        self.nome = nome

        self.ano = ano

        self.velo_maxima = velo_maxima

    def __add__(self, acrescimo):

        self.velo_maxima = 140 + 10

    fusca = Veiculo(nome="Fusca 1600", ano=1994,
velo_maxima=140)

    fusca + 10

    print(vars(fusca))
```

Para os outros operadores, também existem métodos mágicos:

- subtração(-): `__sub__`
- multiplicação(*): `__mul__`
- divisão(/): `__div__`
- módulo(%): `__mod__`
- potência(**): `__pow__`

Para dar a possibilidade de aplicação dos operadores de comparação aos objetos a lista de métodos abaixo pode ser usada:

- `__lt__(self, other)` : Operador de menor (<)
- `__le__(self, other)`: Operador de menor ou igual (<=)
- `__eq__(self, other)`: Operador de igualdade (==)
- `__ne__(self, other)`: Operador de desigualdade (!=)
- `__gt__(self, other)` : Operador de maior (>)
- `__ge__(self, other)` : Operador de maior ou igual (>=)

Como exemplo, supor que se deseja comparar dois fuscas. Entretanto, a única coisa que se deseja saber é se possuem a primeira palavra do nome igual. O código abaixo resume isso:

```
class Veiculo():

    def __init__(self, nome, velo_maxima,
ano="Indefinido"):

        self.nome = nome

        self.ano = ano

        self.velo_maxima = velo_maxima

    def __eq__(self, veiculo_2):

        veiculo_1 = self.nome.split(" ")

        veiculo_1 = veiculo_1[0] #Obtém o primeiro nome

        veiculo_2 = veiculo_2.nome.split(" ")[0] #Obtém
o primeiro nome

        resposta = veiculo_1 == veiculo_2

        return resposta

fusca_1600 = Veiculo(nome="Fusca 1600", ano=1994,
velo_maxima=140)

fusca_1300 = Veiculo(nome="Fusca 1300", ano=1969,
velo_maxima=105)
```

```
print(fusca_1300 == fusca_1600)
```

Para a lista completa de métodos mágicos consultar:
<https://docs.python.org/3/reference/datamodel.html#special-lookup>

MODULARIZAÇÃO E REUSO

Códigos podem conter muitas funcionalidades interessantes, sem dúvidas. Imagine um algoritmo que gerencie o funcionamento de uma turbina de avião como descrito abaixo chamado de **turbinas.py**

```
class Turbina():

    def __init__(self, empuxo_maximo):

        self.empuxo_maximo = empuxo_maximo

        self.empuxo = 0

    def acelera_empuxo(self,
empuxo_solicitado):

        if((self.empuxo +
empuxo_solicitado)<self.empuxo_maximo):

            self.empuxo = self.empuxo +
empuxo_solicitado

            return self.empuxo

        else:

            self.empuxo = self.empuxo_maximo

            return self.empuxo

rolls_royce_trent_900 = Turbina(90)
```

```
empuxo_atual = rolls_royce_trent_900.acelera_empuxo(50)

print(empuxo_atual)
```

Também existe um arquivo chamado **flaps.py** para o controle de ângulo dos flaps da asa de um avião:

```
class Flaps():

    def __init__(self, algulo_esquerdo_maximo,
algulo_direito_maximo):

        self.algulo_esquerdo_maximo =
algulo_esquerdo_maximo

        self.algulo_esquerdo = 0

        self.algulo_direito_maximo =
algulo_direito_maximo

        self.algulo_direito = 0

    def altera_angulo(self, lado, angulo):

        if(lado=="Esquerdo" and
((angulo+self.algulo_esquerdo)<self.algulo_esquerdo_maximo))
:

            self.algulo_esquerdo = angulo

            return self.algulo_esquerdo

        elif(lado=="Esquerdo"):

            self.algulo_esquerdo =
self.algulo_esquerdo_maximo

            return self.algulo_esquerdo

        if(lado=="Direito" and
((angulo+self.algulo_direito)<self.algulo_direito_maximo)):

```



```
        self.angulo_direito = angulo

        return self.angulo_direito

    elif(lado=="Direito"):

        self.angulo_direito =
self.angulo_direito_maximo

        return self.angulo_direito

asas = Flaps(60, 60)

angulo = asas.altera_angulo("Esquerdo", 70)

print(angulo)

angulo = asas.altera_angulo("Direito", 30)

print(angulo)
```

Em certo ponto, está se desenvolvendo um algoritmo para gerenciar o funcionamento do avião. Nele, é necessário o uso de um módulo para gerenciar as turbinas e outro para gerenciar os flaps. Coincidentemente ambos estão escritos em arquivos chamados **turbinas.py** e **flaps.py**.

Para usar funções, métodos e classes de outros arquivos, basta importá-los com a diretiva **import** de forma idêntica à importação de módulo python comum:

```
import turbinas, flaps
```

Para criar uma turbina, basta referenciar o módulo da seguinte forma:

```
rolls_royce_trent_900 = turbinas.Turbina(90)
```

Com o objeto Turbina criado já é possível acessar todos os seus métodos e atributos:

```
rolls_royce_trent_900.acelera_empuxo(50) #Método  
rolls_royce_trent_900.empuxo #Atributo
```

Também é possível importar apenas uma Classe ou método específico deste módulo. Por exemplo, importar a Classe Flaps de flaps.py:

```
from flaps import Flaps
```

Desta forma, a criação de objetos Flaps fica mais elegante:

```
asas = Flaps(80,80)
```

Como é fácil notar, todas as importações respeitam as formas de importação do capítulo 16 pois neste ponto, os arquivos Python são evidentemente módulos.

CAPÍTULO 13

Testes em Python

À medida que códigos começam a ficar mais complexos, uma rotina de testes começa a se fazer necessária. Testes são uma forma de garantir que o código se comporte de maneira adequada e quando novas implementações surgirem elas não causem bugs em trechos de código que já estavam funcionando.

O conceito de um teste, consiste em verificar se uma ação teve o resultado esperado. Por exemplo, ao somar 2+2 se espera o resultado 2 como visto abaixo:

```
def soma(numero1, numero2):  
  
    soma = numero1 + numero2  
  
    return soma  
  
print(soma(2,2)) #Esperado 4
```

Para testar resultados, o Python possui um módulo chamado **unittest**, Dentro dele existe uma Classe chamada **TestCase** que proporciona um tratamento de todas as rotinas de teste.

Na função soma o seguinte exemplo proporciona um teste usando o método **assertEqual()** do inglês “afirmar igual”. Esta função, basicamente, confere se dois resultados passados são iguais.

Um arquivo chamado **tests.py** foi criado. Abaixo segue um teste verificando se a função soma está fazendo o que deve:

```
from unittest import TestCase  
  
def soma(numero1, numero2):  
  
    soma = numero1 + numero2  
  
    return soma
```

```
class Testes(TestCase):  
  
    def teste_soma(self):  
  
        soma_via_funcao = soma(2,2)  
  
        soma_esperada = 4  
  
        self.assertEqual(soma_via_funcao,soma_esperada)
```

Para executar um teste a seguinte sintaxe deve ser usada:

```
python -m unittest {nome_do_arquivo}
```

Para o exemplo anterior:

```
python -m unittest tests.py
```

Se o resultado da função retornada não for 4, o código irá exibir um erro ao reproduzir o teste.

Para o exemplo do avião, anteriormente abordado, um teste importante seria checar se a rotina de aumento de empuxo da turbina não está ultrapassando o limite máximo.

Então, basta importar o módulo turbinas que foi anteriormente criado. O código abaixo demonstra uma rotina que checa se a função funciona:

```
from unittest import TestCase  
  
from turbinas import Turbina  
  
class TestesTurbina(TestCase):  
  
    def teste_expuxo_maxio(self):  
  
        max_empuxo = 100  
  
        turbina = Turbina(max_empuxo) #criar objeto
```

```
        empuxo_da_funcao = turbina.acelera_empuxo(999)
#Forçar um valor

        self.assertEqual(empuxo_da_funcao, max_empuxo)
#Verificar se tudo ocorreu bem
```

Em desenvolvimento de sistemas profissionais, algumas regras de testes devem ser seguidas:

- Uma unidade de teste deve se concentrar em um pequeno número de funcionalidades e provar que tudo está correto.
- Cada unidade de teste deve ser totalmente independente, ou seja, cada teste deve ser capaz de ser executado sozinho ou dentro do conjunto de testes, independentemente da ordem em que são chamados. A implicação desta regra é que cada teste deve ser carregado com um novo conjunto de dados e talvez seja necessário fazer alguma limpeza depois. Isso geralmente é gerenciado pelos métodos setUp() e tearDown().
- Tente arduamente fazer testes que funcionem rapidamente. Se um único teste precisa de mais de alguns milissegundos para executar, o desenvolvimento será diminuído ou os testes não serão executados com a frequência desejável. Em alguns casos, os testes não podem ser rápidos porque eles precisam de uma estrutura de dados complexa para trabalhar, e esta estrutura de dados deve ser carregada toda vez que o teste é executado. Mantenha esses testes mais pesados em um conjunto de teste separado executado por alguma tarefa agendada e execute todos os outros testes sempre que necessário.
- Aprenda suas ferramentas e aprenda como executar uma única prova ou um caso de teste. Então, ao desenvolver uma função dentro de um módulo, execute os testes dessa função frequentemente, de forma ideal, automaticamente quando salvar o código.
- Use nomes longos e descritivos para testar funções. O guia de estilo aqui é ligeiramente diferente do código de execução, onde os nomes curtos são frequentemente preferidos. O motivo é que as funções de teste nunca são chamadas explicitamente. Esses nomes de função são exibidos quando um teste falhar e deverá ser o mais descritivo possível.

EXERCÍCIO PROPOSTO

Supor que uma empresa tenha sido contratada para criar o algoritmo que irá operar na usina **Angra 3**, uma importante usina nuclear em construção no Brasil. Como a Usina faz parte de um complexo (as quais **Angra 1** e **Angra 2** fazem parte) muito código pode ser reaproveitado.

Visto que existem tarefas executadas infinitamente (contador de watts gerados) e a usina precisa ter eficiência máxima de código, crie um gerenciador de planta que contenha:

Um objeto que represente o módulo gerador contendo:

- A pressão de vapor
- A pressão máxima de vapor
- A temperatura de vapor
- A temperatura máxima de vapor
- O número de pás da turbina
- A marca da turbina
- O modelo de rotor usado
- A velocidade máxima do rotor
- A velocidade máxima de giro da turbina
- A potência elétrica aparente produzida [VA]
- A potência elétrica real produzida [W]
- O fator de potência do grupo gerador

Um objeto que representa o reator nuclear contendo:

- Combustível nuclear
- Material Moderador
- Tipo de Refrigeração
- Fabricante
- Ano
- Local de fabricação

Uma função que gere dados aleatórios dos sensores (para fins de simulação) contendo:

- Temperatura da água (deve estar entre 30 e 45)
- Temperatura do Vapor (500° e 520°)
- A potência elétrica aparente produzida [VA]
- A potência elétrica real produzida [W]
- O fator de potência (potência real / potência aparente)
- Pressão de Vapor (90 bar e 105)

Uma rotina de testes é importante

- Fazer uma checagem dos sensores a cada 5 segundos

Um loop principal de comando com as operações:

- Mostrar sensores
- Extrair relatórios Excel
- Extrair relatórios Word

EXERCÍCIOS COMPLEMENTARES

1. Porque usar objetos?

2. O que o método “atira” do código abaixo faz?

```
class Tanque():  
  
    def __init__(self):  
  
        self.nome = "M1 Abrams"  
  
        self.motor = "Turbina Multi-combustível Honeywell  
AGT-1500C, Renk HSWL 354 1500 shp"  
  
        self.arma_primaria = "M1A2SEP: 120mm L44 M256"  
  
    def atira(self, localizacao):  
  
        print("Fogo em: "+localizacao)
```

3. O que diferencia um método de uma propriedade?

4. Porque o código abaixo não funciona? O que pode ser feito para resolver este problema?

```
class Hilux():  
  
    def __init__(self, distancia, esquerda, direita):  
  
        self.distancia = distancia
```

```
        self.esquerda = esquerda

        self.direita = direita

    def andar(self, distancia):

        self.distancia += distancia

    def virar(self, sentido):

        if (sentido=="esquerda"):

            self.esquerda += 1

        else:

            self.direita += 1

modelo_2017 = Hilux(0,0,0)

modelo_2017.controle_de_estabilidade
```

5. Qual é a diferença que existe entre objetos e classes em Python?

6. Qual é a importância dos testes em uma aplicação?

7. Como usar uma classe para descrever um São Bernardo, de 5 anos, Macho, 76 cm de altura, com coloração Brownish-yellow, nascido na Suíça que possui uma personalidade Vivaz, Amigável, Vigilante, Calmo e Gentil?

Desafio - Existe um estabelecimento que é especializado em cuidar de animais durante o período do dia (enquanto seus donos estão trabalhando). Este estabelecimento cobra seguindo a tabela abaixo:

Tipo de Animal	Valor
Cachorro	R\$ 100
Gato	R\$ 150
Passáros	R\$ 50
Iguana	R\$ 150
Tartaruga	R\$ 50

Um adicional de preço é usado de acordo com o porte do animal seguindo a seguinte tabela:

Porte	Taxa
Pequeno	0%
Médio	10%
Grande	15%

A empresa trabalha com um plano de fidelidade, feito com base em quantos pontos o dono do animal tem. O dono ganha pontos quando traz seu animal regularmente. A cada reputação, o dono ganha um desconto seguindo a seguinte tabela:

Reputação	Pontos	Desconto
Filhote	100 - 999	5%
Lobo Aprendiz	1000 - 4000	10%
Líder da Matilha	>4000	15%

No exercício proposto:

Quais são os objetos que devem ser criados em um software?

- Quais são os atributos de cada objeto?
- Quais são os métodos que cada objeto deve ter?
- Quais testes poderiam ser criados?

Com base nas respostas anteriores, projete um sistema que execute as seguintes operações:

- Cadastro de Donos
- Cadastro de Animais
- Cadastro de Estadia
- Cálculo de custo de estadia com base no dono e no animal.

Os dados podem ser armazenados em um arquivo xlsx ou csv.

CAPÍTULO 14

Generators

DEFINIÇÃO DE GENERATORS

Generators são funções que permitem declarar uma função com comportamento de iterador, ou seja, que pode ser usado dentro de um loop for ou com a função **next()** como listas e dicionários.

Generators são uma forma simples de se criar iteradores. Ele irá retornar um objeto (iterador) para que nós possamos iterar sobre este objeto (um valor de cada vez).

É muito simples criar uma função Generator, mas existem algumas peculiaridades. Por exemplo, é usado a declaração **yield** ao invés de **return**. Se a função contém ao menos uma declaração **yield** então ela se torna uma função Generator.

O exemplo abaixo imprime a vez que foi chamada:

```
def generator():  
  
    print("Execução do primeiro bloco")  
  
    yield True  
  
    print("Execução do segundo bloco")  
  
    yield True  
  
    print("Execução do terceiro bloco")  
  
    yield True  
  
funcao = generator()
```

```
next(funcao)

next(funcao)

next(funcao)
```

É como se uma função fosse quebrada em blocos, e cada bloco fosse executado uma vez. Desta forma, é possível executar um bloco de cada vez em uma função for como visto no exemplo abaixo:

```
def generator():

    yield("Terceiro lugar, um dia você chega lá")

    yield("Segundo lugar, quase lá!")

    yield("Primeiro lugar, parabéns!")

loop = generator()

for posicao in loop:

    print(posicao)
```

GENERATORS EXPRESSIONS

As Generators Expressions facilitam à criação de Generators. Assim como uma função lambda cria uma função anônima, uma Generator Expression cria uma função Generator anônima.

A sintaxe é semelhante à de uma função lambda dentro de uma tupla. Por exemplo, para criar um generator que faça a impressão do nome dos 4 filmes com maior bilheteria da história:

```
lista_filmes = [  
    "Star Wars: O Despertar da Força",  
    "Titanic",  
    "Avatar",  
    "Vingadores: Ultimato"  
]  
  
generator = (print(filme) for filme in lista_filmes)  
  
next(generator) #1ª Chamada  
next(generator) #2ª Chamada  
next(generator) #3ª Chamada  
next(generator) #4ª Chamada
```

USO DE GENERATORS

Existem várias razões que tornam os generators uma implementação atraente para ser adotada. A primeira delas é a facilidade de implementar.

Os geradores podem ser implementados de forma clara e concisa em comparação com a sua classe de métodos iteradores como o `__iter__` e o `__next__`.

A seguir, é apresentado um exemplo para implementar uma sequência de potência de 2 usando a classe de iterador.

```
class PotenciaDeDois:  
  
    def __init__(self, max = 0):  
  
        self.max = max  
  
        self.n = 0  
  
    def __next__(self):  
  
        if self.n > self.max:  
  
            raise StopIteration
```

```
        result = 2 ** self.n

        self.n += 1

        return result

potencia_dois = PotenciaDeDois(10)

resultado = next(potencia_dois)

print(resultado)

resultado = next(potencia_dois)

print(resultado)

resultado = next(potencia_dois)

print(resultado)
```

Isso foi demorado. Agora usando uma função de gerador:

```
def PotDoisGenerator(max = 0):

    n = 0

    while n < max:

        yield 2 ** n

        n += 1

potencia_dois = PotDoisGenerator(9)
```

```
resultado = next(potencia_dois)

print(resultado)

resultado = next(potencia_dois)

print(resultado)

resultado = next(potencia_dois)

print(resultado)
```

Como os geradores controlam os detalhes automaticamente, ele se torna muito mais conciso e limpo na implementação.

Outro fato importante é otimização de memória. Uma função normal para retornar uma sequência criará a sequência inteira na memória antes de retornar o resultado.

Não há problemas com isso, a menos que um grande número de dados sejam tratados no processo. Nestes casos, a implementação do gerador dessa sequência é compatível com a memória e é preferida, pois produz apenas um item por vez.

Em algumas aplicações, isso se torna crítico, como para representar fluxos infinitos. Os generators são excelentes meios para representar um fluxo infinito de dados.

Fluxos infinitos não podem ser armazenados na memória (devido ao seu tamanho) e, como os geradores produzem apenas um item por vez, ele pode representar um fluxo infinito de dados.

O exemplo a seguir pode gerar todos os números pares (em teoria) que existem.

```
def todos_pares():

    n = 0

    while True:

        yield n

        n += 2
```

CAPÍTULO 15

List Comprehensions

Em Python, como já abordado, colchetes são usados para criação de listas. Exemplo:

```
# Apenas números

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

# Letras e números

lista_alfanumerica = ['a', 'b', 'c', 1, 2, 3]
```

List Comprehension, ou compressões de lista são uma forma concisa de criar e manipular listas.

Sua sintaxe básica é:

```
[expressão for item in lista]
```

Em outras palavras: aplique a expressão em cada item da lista.

Como exemplo, dado o seguinte código:

```
for item in range(10):
    lista.append(x**2)
```

Pode-se reescrever, utilizando compressões de lista da seguinte forma:

```
lista = [item**2 for item in range(10)]
```

Ou seja: aplique a potência de 2 em todos os itens da lista.

Um outro exemplo é dado no seguinte código, que transforma os itens da lista em maiúsculos com o método **upper()**:

```
for item in lista:
    resultado.append(str(item).upper())
```

Pode-se reescrever da seguinte forma:

```
resultado = [str(item).upper() for item in lista]
```

LIST COMPREHENSIONS COM IF

Compressões de lista podem utilizar expressões condicionais para criar listas ou modificar listas existentes.

Sua sintaxe básica é:

```
[expressão for item in lista if condição]
```

Em outras palavras: “Aplique a expressão em cada item da lista caso a condição seja satisfeita”

Por exemplo, pode-se retirar os números ímpares de um conjunto de número da seguinte forma.

```
resultado = [numero for numero in range(20) if numero %  
2 == 0]
```

LIST COMPREHENSIONS COM VÁRIOS IF'S

Pode-se verificar condições em duas listas diferentes dentro da mesma list comprehension.

Por exemplo, para saber os Múltiplos Comuns de 5 e 6.

Utilizando múltiplos if's e compressões de lista, pode-se criar o seguinte código:

```
resultado = [numero for numero in range(100) if numero  
% 5 == 0 if numero % 6 == 0]
```

Ou seja, o número só será passado para lista de resultado caso sua divisão por 5 E por 6 seja igual à zero.

LIST COMPREHENSIONS COM IF + ELSE

Outra forma de se utilizar expressões condicionais e compressões de lista é usar o conjunto if + else.

A sintaxe básica para essa construção é:

```
[resultado_if if expressão else resultado_else for item  
in lista]
```

Em outras palavras: para cada item da lista, aplique o resultado resultado_if se a expressão for verdadeira, caso contrário, aplique resultado_else.

Por exemplo, para criar uma lista que contenha "1" quando determinado número for múltiplo de 5 e "0" caso contrário.

Pode-se codificá-lo da seguinte forma:

```
resultado = ['1' if numero % 5 == 0 else '0' for numero
in range(16)]
```

ANINHAMENTO DE COMPRESSÕES DE LISTA

Considerando a matriz:

```
matrix = [
    [1, 2, 3, 4],
    [5, 6, 7, 8],
    [9, 10, 11, 12]
]
```

O seguinte resultado é esperado:

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} = \begin{bmatrix} 1 & 5 & 9 \\ 2 & 6 & 10 \\ 3 & 7 & 11 \\ 4 & 8 & 12 \end{bmatrix}^T$$

Em Python, pode-se fazer isso da seguinte forma

```
transposta = []

matriz = [[1, 2, 3, 4], [4, 5, 6, 8], [9, 10, 11, 12]]

for i in range(len(matriz[0])):

    linha_transposta = []
```

```
for linha in matriz:

    linha_transposta.append(linha[i])

transposta.append(linha_transposta)
```

A matriz transposta conteria:

```
transposta = [[1, 4, 9], [2, 5, 10], [3, 6, 11], [4, 8, 12]]
```

Pode-se reescrever o código acima, de transposição de matrizes, da seguinte forma, utilizando compressão de listas:

```
matriz = [[1, 2, 3, 4], [4, 5, 6, 8], [9, 10, 11, 12]]

transposta = [[linha[i] for linha in matriz] for i in range(
len(matriz[0]))]
```

No código acima:

- No primeiro loop, *i* assume o valor de 0, portanto `[linha[0] for linha in matriz]` vai retornar o primeiro elemento de cada linha: `[1, 4, 9]`
- No segundo loop, *i* assume o valor de 1, portanto `[linha[1] for linha in matriz]` vai retornar o segundo elemento de cada linha: `[2, 5, 10]`
- No terceiro loop, *i* assume o valor de 2, portanto `[linha[2] for linha in matriz]` vai retornar o terceiro elemento de cada linha: `[3, 6, 11]`
- No quarto loop, *i* assume o valor de 3, portanto `[linha[3] for linha in matriz]` vai retornar o quarto elemento de cada linha: `[4, 8, 12]`

COMPRESSÕES DE DICIONÁRIOS (DICT COMPREHENSIONS)

A compreensão de listas é uma maneira prática e rápida de criar listas em apenas uma única linha de código.

A ideia de compreensão não é exclusiva das listas. Os dicionários, uma das estruturas que por sinal é comumente usada na ciência de dados, também podem ser comprimidas.

A ideia usada na compreensão da lista também é transportada na compreensão de um dicionário. A compreensão de um dicionário segue a seguinte sintaxe:

```
{key: valor para i na lista}
```

Por exemplo, para criar uma compreensão de dicionário a partir de uma lista de números sendo que a chave será uma string do seu resultado:

```
lista = [1,2,3,4]

dicionario = {str (i): i for i in [1,2,3,4,5]}

print(dicionario)
```

Um outro exemplo interessante é criar um dicionário contendo o tamanho da palavra de chave. Por exemplo, partindo de uma lista de frutas:

```
frutas = ['maçã', 'manga', 'banana', 'cereja']

dicionario = {f: len (f) for f in frutas}

print(dicionario)
```

Outro uso da compreensão é a de inverter chaves por valores em um dicionário existente. Às vezes, convém criar um novo dicionário a partir de um já existente, de modo que a chave vire valor e vice versa.

Usando a função enumerate é possível obter um dicionário com o item e o seu index:

```
frutas = ['maçã', 'manga', 'banana', 'cereja']

dicionario = {fruta: indice for indice, fruta in
enumerate(frutas)}

print(dicionario)
```

Caso seja desejado, é possível trocar o index pelo valor da seguinte forma:

```
dicionario = {fruta: indice for indice, fruta in
dicionario.items()}

print(dicionario)
```

Um outro uso recorrente de compressões é na exclusão de chaves determinadas chaves ou valores.

Digamos que exista um dicionário e queira criar um novo dicionário removendo as chaves a partir de um set de dados. Como por exemplo, remover “bananas” e “maçãs”, o exemplo abaixo retrata isso:

```
frutas = ['maçã', 'manga', 'banana', 'cereja']

frutas_dict = {fruta: fruta.capitalize() for fruta in
frutas}

print(frutas_dict)

remover = {"banana", "maçã"}

dicinario_filtrado = {chave:frutas_dict[chave] for
chave in frutas_dict.keys()-remover}

print(dicinario_filtrado)
```

CONDICIONAIS NA COMPRESSÃO DE DICIONÁRIOS

Assim como nas lista, é possível utilizar estruturas condicionais em compressões de dicionários.

Por exemplo, deseja-se adicionar o texto com a primeira letra maiúsculas para todas as frutas que não sejam maçã:

```
frutas = ['maçã', 'manga', 'banana', 'cereja']

frutas_dict = {fruta: fruta.capitalize() for fruta in
frutas if fruta!="maçã"}

print(frutas_dict)
```

Outro exemplo consiste em substituir o resultado da função caso seja “manga” por “Nops”:

```
frutas = ['maçã', 'manga', 'banana', 'cereja']
```

```
frutas_dict = {fruta: fruta.capitalize() if
fruta!="manga" else "Nops" for fruta in frutas}

print(frutas_dict)
```

COMPRESSÃO DE SETS (SET COMPRESSIONS)

O aspecto principal da compreensão de um dado do tipo set que o torna único é que os elementos no interior serão desordenados e não poderão conter duplicatas. O resto é praticamente o mesmo que uma compreensão de lista. A entrada pode ser qualquer coisa que contenha um grupo de elementos.

Por exemplo, se desejar retirar todas as palavras repetidas na forma de um conjunto de um texto.

Para isso é necessário o uso das funções **lower()** (Passar todo o texto para minúsculo), **replace()** (Para remover a vírgula e o ponto) e o **split()**, para transformar um texto em uma lista como visto no código abaixo:

```
texto = "Atirei o pau no gato, to, to mas o gato to,
to, to não morreu, reu, reu"

palavras = texto.lower().replace('.', '').replace(',', ' ')
.split()

palavras_unicas = {palavra for palavra in palavras}

print(palavras_unicas)
```

CONDICIONAIS EM SET COMPRESSIONS

Para expandir o exemplo anterior, imagine que se deseja filtrar todas as palavras únicas que possuam mais de 4 letras. Para isso o código abaixo traz uma solução:

```
palavras_unicas = {palavra for palavra in palavras if
len(palavra)>=4}
```

Deseja-se agora, retornar a palavra em maiúsculas, usando a função `capitalize()` atendendo a condição se a primeira letra seja "A", senão trazer a formatação normal:

```
palavras_unicas = {palavra.capitalize() if
palavra[0]=="a" else palavra for palavra in palavras}
```

Todos os exemplos anteriores podem ser unidos em um único set, trazendo um conjunto de palavras únicas que contenham apenas palavras maiores que 4 caracteres e que caso iniciem com a letra "a" tenham a primeira letra transformada em maiúscula:

```
palavras_unicas = {palavra.capitalize() if
palavra[0]=="a" else palavra for palavra in palavras if
len(palavra)>=4}
```

EXERCÍCIO PROPOSTO

Criar um Algoritmo que busque dados da ação 'Petróleo Brasileiro S.A. - Petrobras (PETR4.SA)' no endereço "https://br.financas.yahoo.com/quote/petr4.SA". Salvar estes dados em um banco csv cada vez que seja executado.

EXERCÍCIOS COMPLEMENTARES

1. Escreva as estruturas de repetição abaixo usando compressões:

a)

```
peessoas = []
for pessoa in range(10):
    pessoas.append(pessoa)
print(pessoas)
```

b)

```
peessoas = []
for pessoa in range(10):
    if(pessoa<5):
        pessoas.append("Permitido")
    else:
        pessoas.append("Esgotado")
print(pessoas)
```

c)

```
convidados = {"Ana", "Paula", "Joana", "Robson", "Ruan"}
convidados_R = set()

for convidado in convidados:
    if(convidado[0] == "R"):
        convidados_R.add(convidado)

print(convidados_R)
```

d)

```
from random import randint
num_aleatorios = {}
for aleatorio in range(10):
    numero = randint(0,10)
    num_aleatorios.update({aleatorio:numero})
print(num_aleatorios)
```

CAPÍTULO 16

Anaconda e Jupyter

INTRODUÇÃO AO ANACONDA E JUPYTER

O ramo de Data Science tem tido um crescimento e uma importância cada vez maiores na atualidade.

Python é a linguagem que ganha notória participação neste meio devido a sua facilidade de uso e suas poderosas bibliotecas. Os pacotes, por sinal, podem não ser fáceis de serem gerenciados e pode ser oneroso ficar configurando o ambiente quando a única coisa que se busca é a análise de dados. Para solucionar este e outros problemas existe o Anaconda.

O Anaconda é um projeto open-source que já contém a linguagem Python e centenas de bibliotecas “embutidas”. Com ele é possível usar python e outras bibliotecas quase que “obrigatórias” para fazer análise de dados.

Além das centenas de bibliotecas, o Anaconda é multiplataforma, ou seja, funciona em Windows, Linux e MacOS.

Porque eu devo Instalar Python com o Anaconda?

Como o Anaconda já contém centenas de bibliotecas é muito mais fácil e produtivo usar esse pacote do que instalar cada biblioteca manualmente, concorda?

Você instala o Anaconda e diversas bibliotecas como por exemplo: pandas, matplotlib, seaborn e numpy já vêm no bolo.

Diversas ferramentas e IDE's úteis para Data Science também estão disponíveis, como: Jupyter, Jupyter lab, Spyder, Orange, Microsoft Visual Code etc.

É por isso que eu recomendo sempre utilizar essa plataforma.

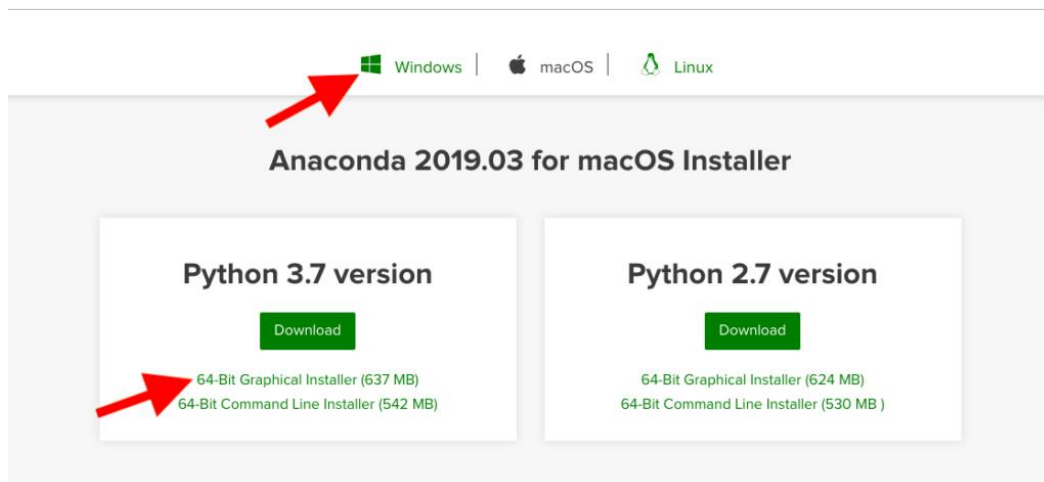
Além de tudo que foi falado acima, é sempre bom frisar: O Anaconda é gratuito.

INSTALAÇÃO DO ANACONDA

Como Instalar o Python no Windows com o Anaconda?

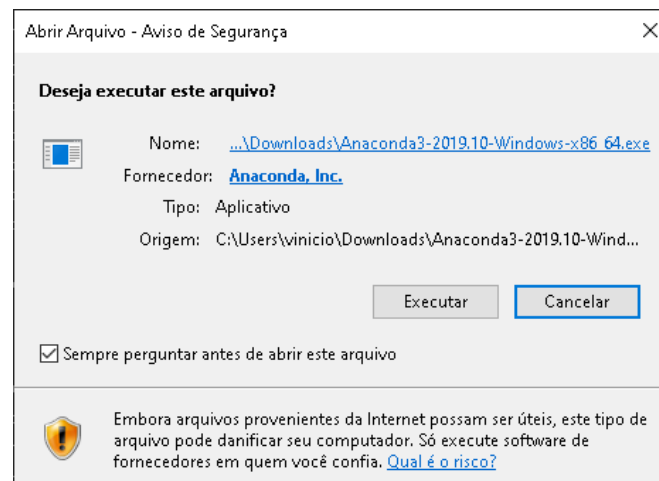
Primeiramente deve-se acessar o site oficial no link:
<https://www.anaconda.com/distribution/>.

Rolando um pouco para baixo, é possível ver as opções de download. Selecione a plataforma Windows e clique em download como retratado na imagem abaixo:

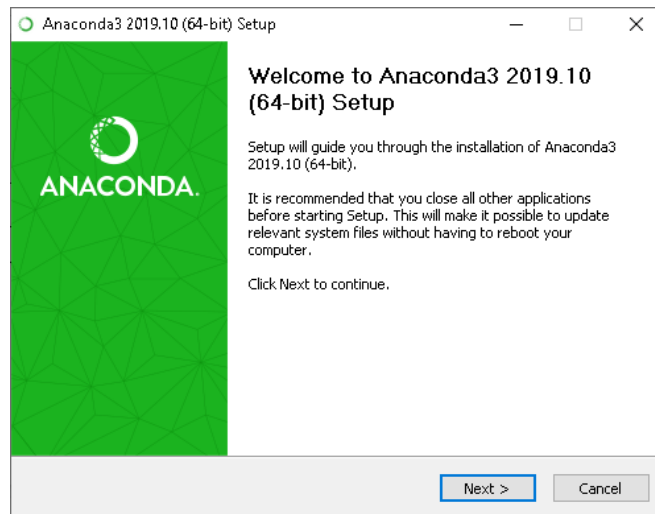


Após o download vamos iniciar a instalação do Anaconda.

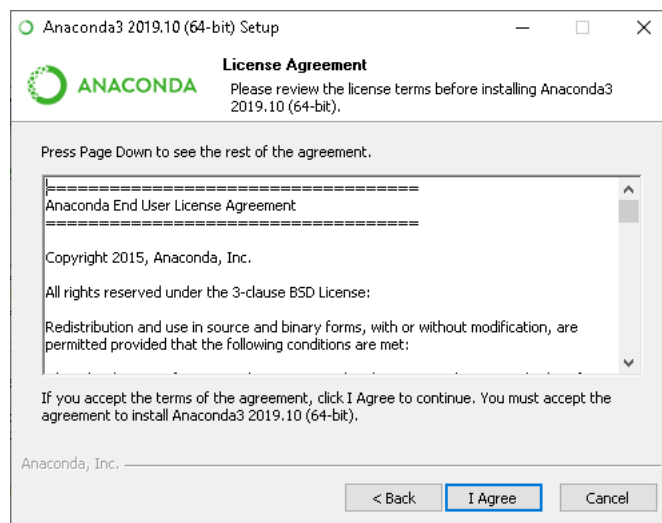
Ao executar o arquivo baixado, uma tela de permissão será exibida, clique em Executar:



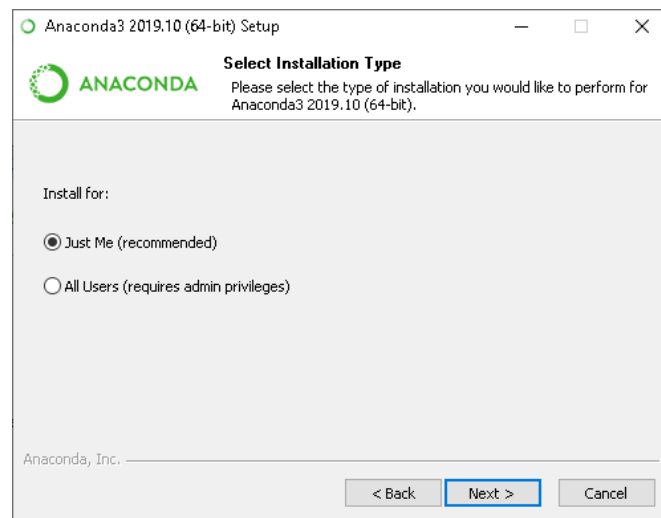
A primeira tela do instalador é apenas informativa, clique em Next para dar sequência:



A tela seguinte são os termos de contrato, clique em “I Agree” (Eu concordo) para dar sequência:

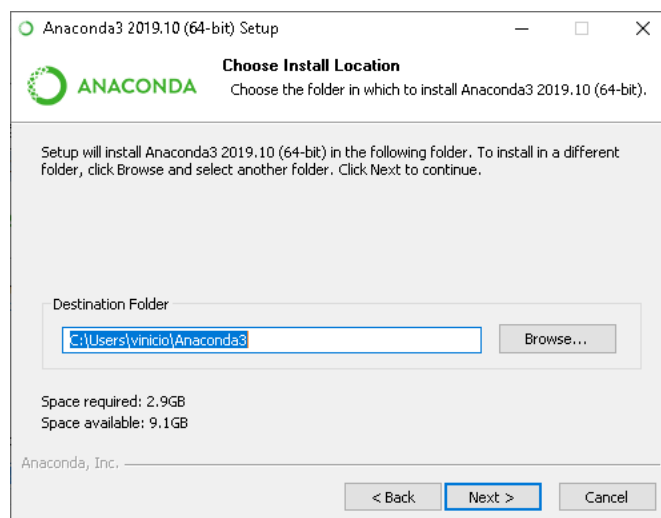


A próxima tela pergunta se deseja instalar só no usuário atual (Just Me) ou em todos (All Users). Selecione “Just Me” e clique em “Next”:

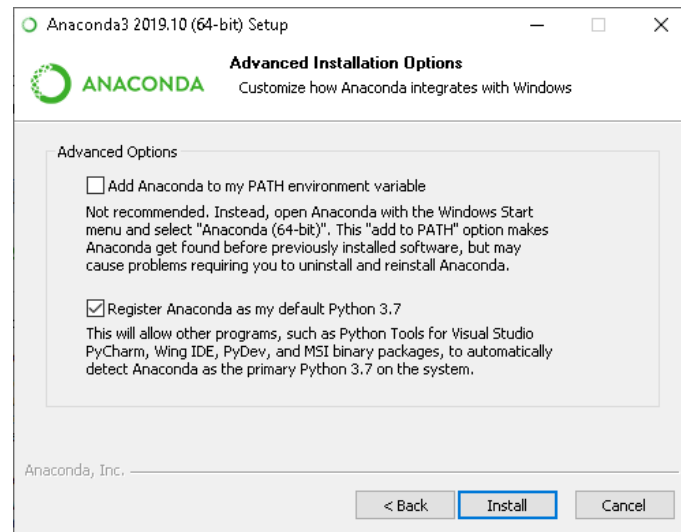


Nesse momento, o instalador sugere uma localização para instalação do Anaconda.

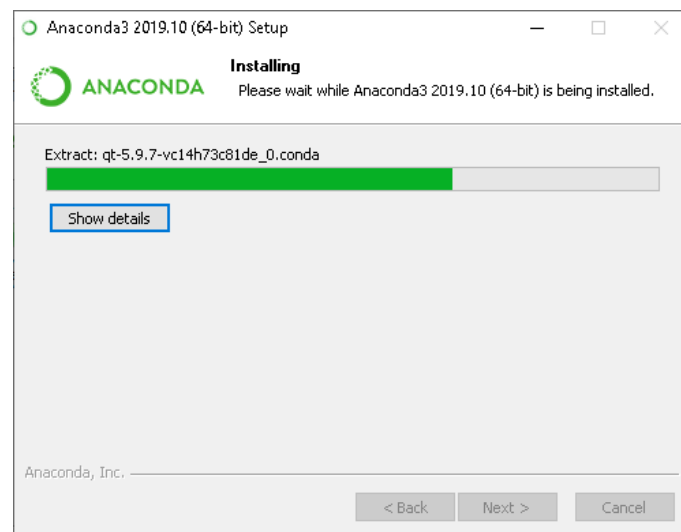
Aceite a opção padrão clicando em “Next”:



Como em Python e com o VSCode é possível adicionar o Anaconda na variáveis de ambiente, entretanto, não é recomendável. Desta forma, apenas de sequência clicando em “Install”:

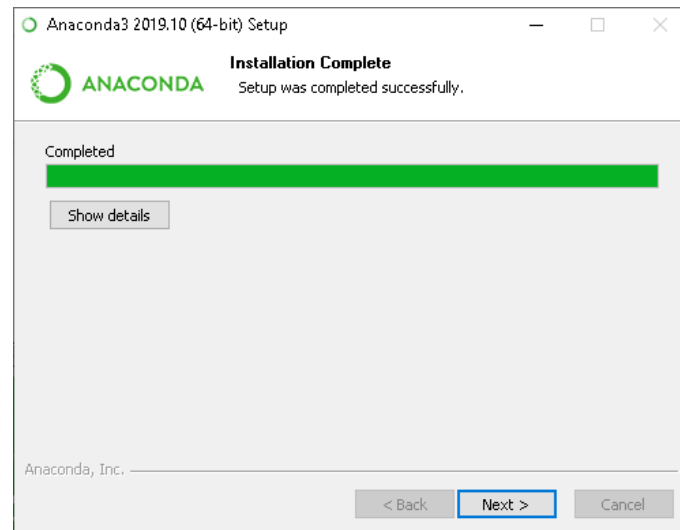


Uma tela de carregamento será iniciada mostrando o progresso da instalação:

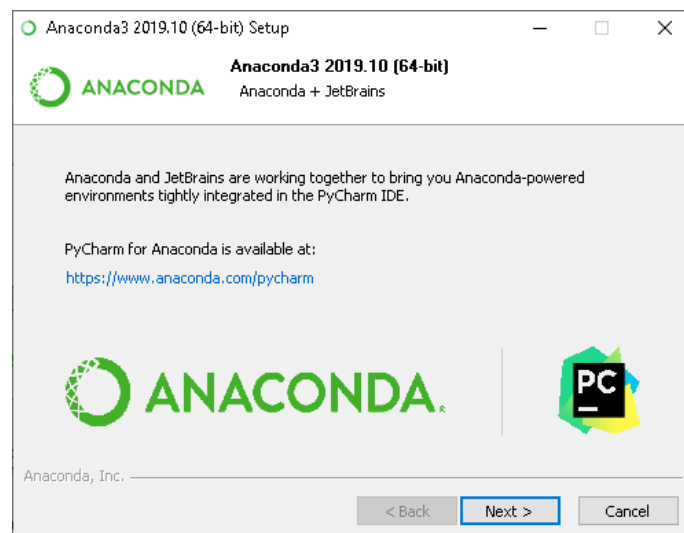


Ao finalizar uma tela de sucesso será exibida, basta clicar em Next:

Em seguida, uma tela de recomendação do uso da IDE PyCharm será exibida. Ignore clicando em “Next”:



Por fim uma tela de agradecimento é exibida, retira as marcações dos checkbox e clique em “Finish”:



Após a conclusão do instalador, o Anaconda está instalado e já contém o Python e diversas bibliotecas instaladas.

INICIANDO O JUPYTER NOTEBOOK

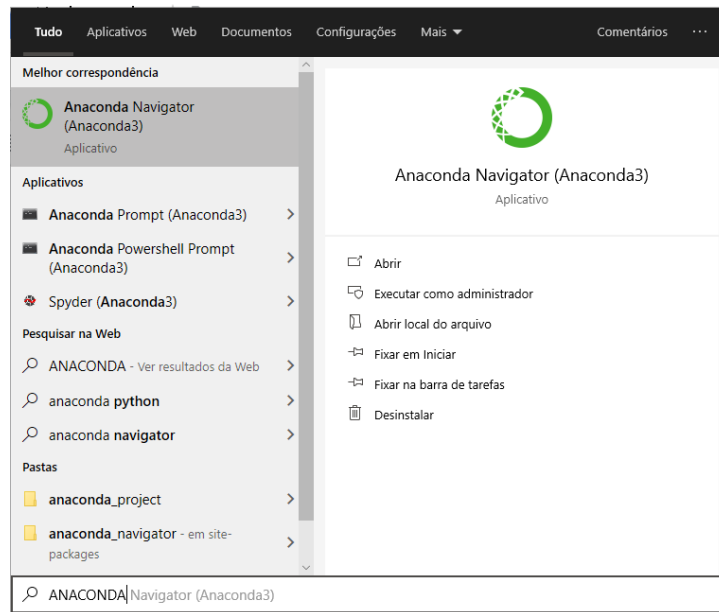
Com o Anaconda instalado temos também o Jupyter Notebook a disposição.

O Jupyter Notebook é uma ferramenta extremamente interessante, pois, permite que seja usando uma interface interativa e simples.

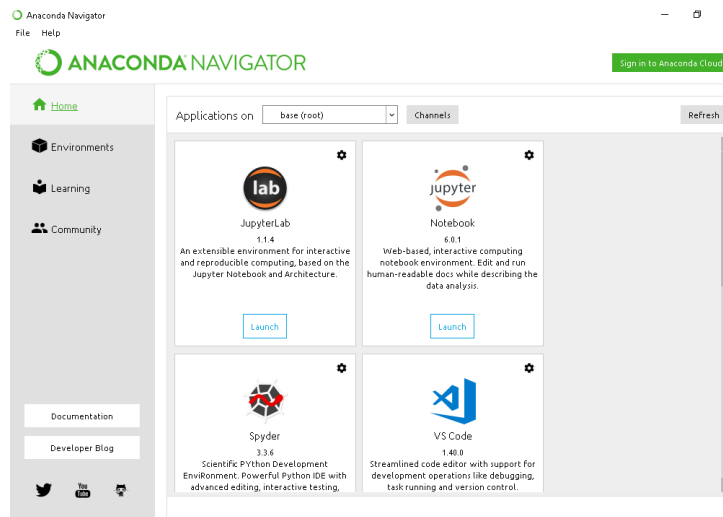
Com essa ferramenta tarefas como por exemplo plotar gráficos ou ainda conectar em bancos de dados se tornará muito mais simples..

Para abrir o jupyter notebook deve-se usar Anaconda Navigator.

Para isso clique no menu iniciar e digite Anaconda

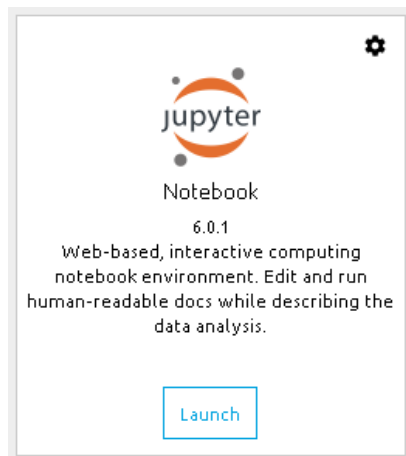


Ao clicar, o navigator é exibido como na imagem a seguir:



Veja que além do Jupyter e do VSCode é possível usar o Spyder, ou ainda o Orange... e mais algumas ferramentas que estão pré-instaladas.

Para iniciar o Jupyter basta clicar em “Launch”:




A opção de onde abrir será exibida. Selecione o navegador Chrome e

Como você deseja abrir este arquivo?


Continuar usando este aplicativo

 Microsoft Edge

Outras opções

 Google Chrome
Novo

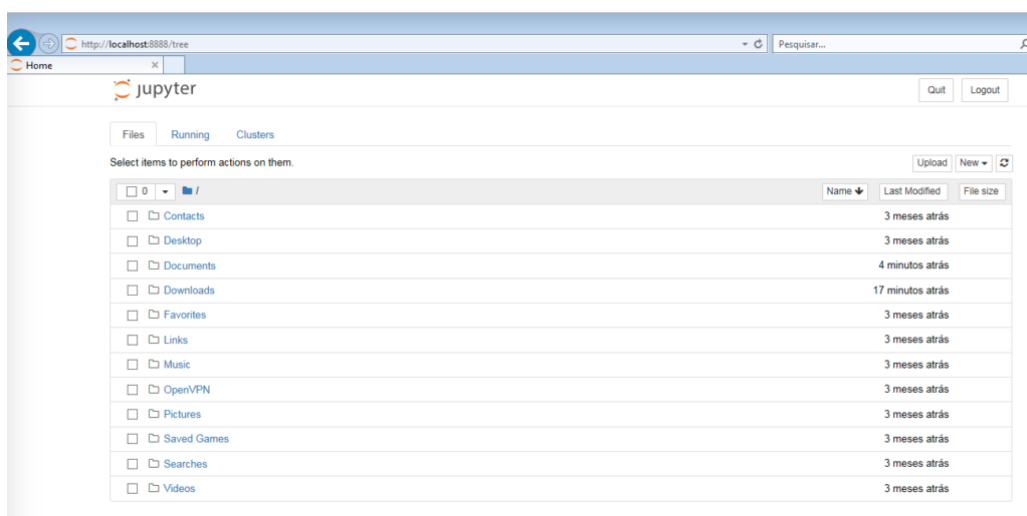
 Internet Explorer

 Procurar um app na Microsoft Store

Mais aplicativos ↓

☐ Sempre usar este aplicativo para abrir arquivos .html

OK



clique em Ok:

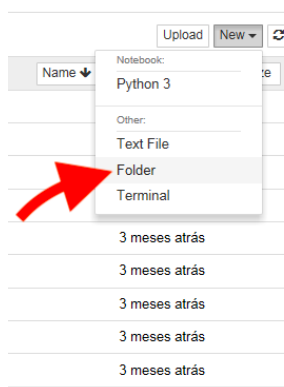
Ao clicar em Launch o Jupyter notebook irá carregar no navegador a interface que será usada.

É fácil notar que se está diante dos diretórios padrão do home do usuário no Windows.

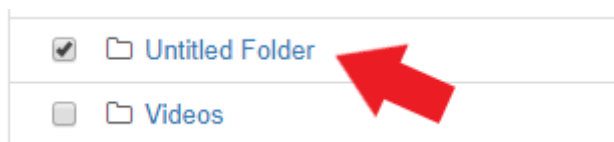
Pode-se criar notebooks e começar escrever códigos Python aqui.

Mas antes disso, por questão de organização, será criado um diretório chamado scripts.

Para isso clique em New, e em seguida em folder como na imagem abaixo:

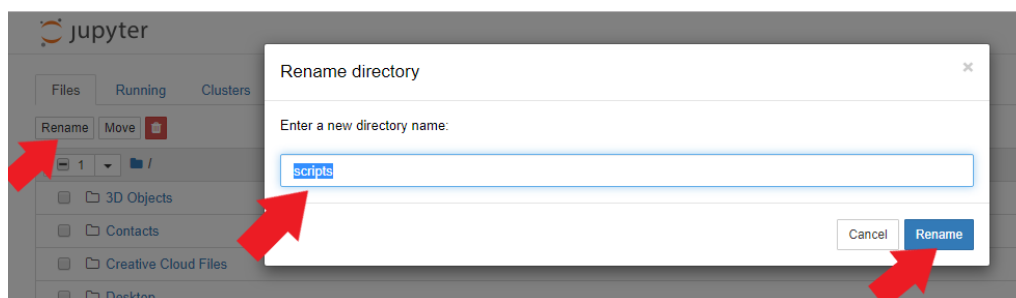


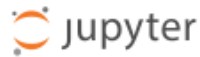
Ao clicar em Folder, uma pasta é criada com o nome "Untitled Folder".
Veja:



Clique no checkbox ao lado do nome para selecionar e em seguida clique no botão Rename acima para renomear essa pasta.

Após renomear o diretório, clique no nome do diretório. Desta forma será aberto um diretório vazio.





Files

Running

Clusters

Select items to perform actions on them.

0 / scripts

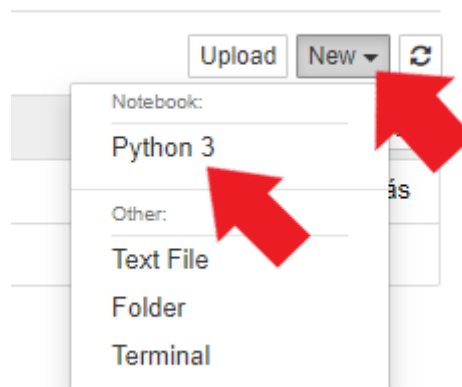
..

The notebook list is empty.

Agora é possível criar notebooks.

CRIANDO UM NOTEBOOK

Para fazer isso, clique em “New” e em seguida clique em “Python 3”.

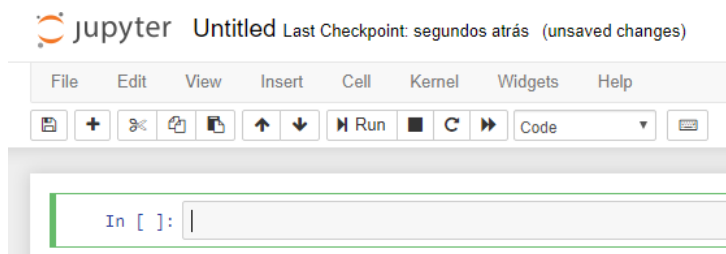


Veja:

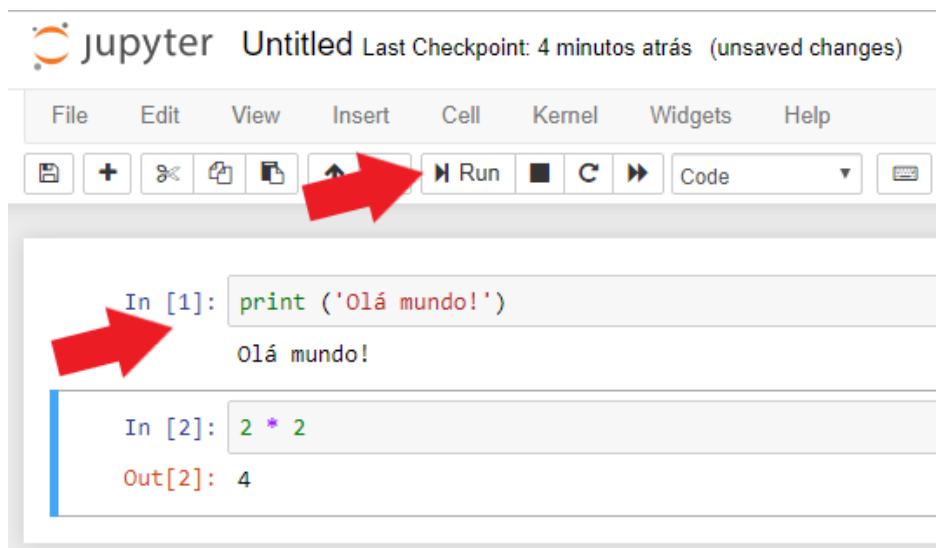
Ao clicar em Python 3 é aberta uma página com um notebook recém criado.

Nesse notebook é que será inserido os códigos python, da mesma forma que em uma IDE.

Veja a primeira célula vazia:



Na célula vazia pode-se inserir códigos Python, basta digitar e clicar em Run em seguida, veja:



CAPÍTULO 17

Conjuntos ou sets

Um conjunto (set) é um tipo de dados de coleção, suportando operadores como o in, a função len() além de ser iterável como uma lista. Conjuntos não possuem noção de ordem por isso seus elementos não podem ser acessados com colchetes [] nem podem ser fatiados.

Os conjuntos não aceitam valores repetidos ao tentar criar um conjunto com valores repetidos eles serão descartados só sobrando um valor do mesmo.

CRIANDO UM CONJUNTO

Conjuntos só aceitam tipos de dados imutáveis como inteiros, floats, tuplas e strings não aceitando listas, dicionários e outros conjuntos (set).

Para criar um conjunto basta colocar os valores entre chaves {} :

```
conjunto = {'Paulo', 'Claudio', 'Marcio', 'Erica'}
```

Pode-se ainda criar conjuntos a partir de uma lista ou tupla com set. Uma grande vantagem é por exemplo limpar os dados repetidos em uma lista:

```
lista = ['Maçã', 'Laranja', 'Uva', 'Abacaxi', 'Maçã',  
        'Abacate', 'Laranja']  
  
conjunto = set(lista)  
  
print(conjunto)
```

Repare no exemplo acima que set não se importou com a ordem dos itens na lista, e como conjuntos só podem ter itens únicos ele dispensou os valores duplicados de Maçã e Laranja.

Um conjunto vazio só pode ser criado através de set() sem argumentos:

```
conj_vazio = set()
```

Usar as chaves vazias para criar um conjunto vazio não é uma forma válida para se criar um conjunto vazio pois o Python vai interpretar como um dicionário vazio:

```
conjunto = {}
```

```
dicionario = {}  
  
print(type(conjunto))  
  
print(type(dicionario))
```

Adicionado um item ao conjunto com o método **add()**. A sintaxe para adicionar um valor à um conjunto é:

```
conjunto.add(valor)
```

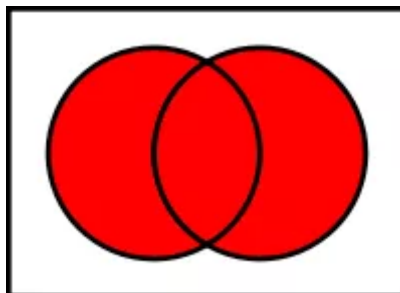
Para esclarecer o exemplo abaixo retrata o uso da propriedade adição:

```
conjunto = {'Maça', 'Uva', 'Abacate', 'Laranja',  
            'Abacaxi'}  
  
conjunto.add('Melancia')  
  
print(conjunto)
```

OPERAÇÕES COM CONJUNTOS

Conjuntos possuem uma série de operações muito interessantes e que facilitam a vida do programador. Abaixo segue a lista das principais operações.

UNIÃO |

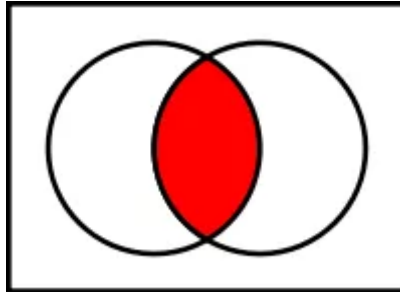


A união de dois conjuntos se dá com o uso do operador "|" e retorna um conjunto contendo todos os itens dos dois conjuntos:

```
conjunto_a = {'Paula', 'Erica', 'Marcio'}  
  
conjunto_b = {'Ana', 'João', 'Paula'}  
  
final = conjunto_a | conjunto_b
```

```
print(final)
```

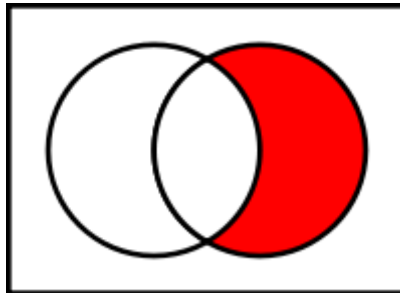
INTERSEÇÃO &



A interseção usa o operador "&" e retorna um conjunto contendo os itens comuns aos dois conjuntos:

```
conjunto_a = {'Paula', 'Erica', 'Marcio'}  
  
conjunto_b = {'Ana', 'João', 'Paula'}  
  
final = conjunto_a & conjunto_b  
  
print(final)
```

DIFERENÇA -



A diferença é usado com o operador "-" e tem por objetivo encontrar os itens que não fazem parte de um conjunto. Por exemplo `conjunto_a - conjunto_b` retorna somente os itens do primeiro conjunto `conjunto_a` que não contém itens iguais em `conjunto_b`:

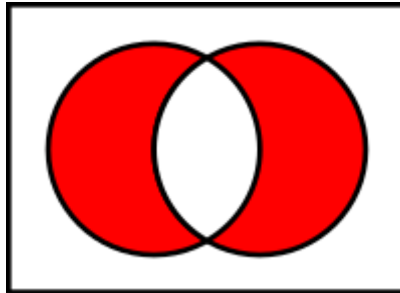
```
conjunto_a = {'Paula', 'Erica', 'Marcio'}  
  
conjunto_b = {'Ana', 'João', 'Paula'}  
  
a_b = conjunto_a - conjunto_b
```

```
print(a_b)

b_a = conjunto_b - conjunto_a

print(b_a)
```

DIFERENÇA SIMÉTRICA ^



A diferença simétrica usa o operador “^” e cria um conjunto contendo todos os itens que não são comuns aos dois conjuntos:

```
conjunto_a = {'Paula', 'Erica', 'Marcio'}
conjunto_b = {'Ana', 'João', 'Paula'}
final = conjunto_a ^ conjunto_b
print(final)
```

ALGUNS MÉTODOS PARA CONJUNTOS

Além das operações tradicionais de união, interseção e diferença, também estão disponíveis operações de verificação de pertinência, ou seja, se algo pertence a outra coisa.

Para, por exemplo, verificar se um determinado item pertence a um conjunto, pode-se usar o já conhecido operador de pertinência in:

```
a = {1, 2, 3, 4}

b = {3, 4, 5, 6}

print(1 in a)

print(5 in a)
```

Com conjuntos é possível verificar se um conjunto é um subconjunto de outro utilizando o método `issubset()`:

```
a = {1, 2, 3, 4}

c = {1, 2}

print(a.issubset(c)) #False

print(c.issubset(a)) #True
```

Além disso, de forma contrária, pode-se verificar se um conjunto é superconjunto de outro com o método `issuperset()`:

```
a = {1, 2, 3, 4}

c = {1, 2}

print(a.issuperset(c)) #True

print(c.issuperset(a)) #False
```

Outra relação que é importante checar é a disjunção entre dois conjuntos com o método `isdisjoint()`. Dois conjuntos são disjuntos se tiverem interseção nula, ou seja, não existem elementos comuns aos dois. Exemplo:

```
a = {1, 2, 3, 4}

b = {5, 6}

c = {1, 2}

print(a.isdisjoint(b)) #True

print(a.isdisjoint(c)) #False
```

CAPÍTULO 18

Geração de Mapas

A visualização geográfica é muito importante em diversas profissões. Para esta tarefa Python possui diversos módulos. Um dos mais expoentes é o Folium.

Este capítulo irá abordar sobre este magnífico módulo. Para usá-lo, é necessário instalá-lo com o gerenciador pip:

```
pip install folium
```

LATITUDE E LONGITUDE

Latitude e longitude são descrições da localização, ou coordenadas geográficas, de um determinado lugar na Terra. O Folium (assim como a maioria dos sistemas de georeferenciamento) se utiliza destas coordenadas.

Por exemplo, o palácio Rio Branco, no Acre, está localizado em -9.975130 de latitude e -67.809653 de longitude.

Para plotar esta localização com o folium, o código abaixo pode ser usado:

```
import folium

mapa = folium.Map(

    [-9.975130, -67.809653]

)

mapa.save("palacio.html")
```

Um arquivo html será gerado. Basta abri-lo em um navegador.

DIMENSÕES

As dimensões do mapa podem ser setadas usando os parâmetros width e height como visto abaixo:


```
mapa = folium.Map(  
  
    width='80%',  
  
    height='80%',  
  
    location=[-9.975130, -67.809653]  
  
)
```

ZOOM

É possível também setar o zoom inicial usando o parâmetro `zoom_start` como visto abaixo:

```
mapa = folium.Map(  
  
    zoom_start=18,  
  
    location=[-9.975130, -67.809653]  
  
)
```

TILES OU MAPAS

Para alternar o tipo de visualização do mapa existe o parâmetro `tiles`. Com ele é possível fazer visualizações como o de impressora de toner como visto abaixo:

```
mapa = folium.Map(  
  
    zoom_start=18,  
  
    location=[-9.975130, -67.809653],  
  
    tiles="Stamen Toner"  
  
)
```

Ou de terreno:

```
mapa = folium.Map(  
  
    zoom_start=18,
```

```
location=[-9.975130, -67.809653],  
  
tiles="Stamen Terrain"  
  
)
```

A lista de mapas usáveis sem o uso de API's está abaixo:

- OpenStreetMap
- Stamen Terrain
- Stamen Toner
- Stamen Watercolor
- CartoDB positron
- CartoDB dark_matter

MARCADORES

Existem diversos marcadores disponíveis no folium. Começando pelo mais simples, adicionando um marcador em cima do palácio Rio Branco:

```
import folium  
  
mapa = folium.Map(  
  
    [-9.975130, -67.809653]  
  
)  
  
tooltip = 'Palácio!'  
  
folium.Marker([-9.975130, -67.809653], popup='<i>Mt.  
Hood Meadows</i>', tooltip=tooltip).add_to(mapa)  
  
mapa.save("marcador.html")
```

O parâmetro popup, faz com que ao clicar sobre o marcador uma mensagem seja exibida.

```
folium.Marker(  
  
    location=[-9.975130, -67.809653],  
  
    popup='Palácio Rio Branco',  
  
).add_to(mapa)
```

Para construir marcadores mais elaborados pode-se fazer uso do folium.Icon(). Este método pode ser usado no o parâmetro icon como visto abaixo:

```
icone = folium.Icon(  
  
    color='red',  
  
    icon='glyphicon glyphicon-send',  
  
    icon_color='white',  
  
    angle=45  
  
)  
  
folium.Marker(  
  
    location=[-9.975130, -67.809653],  
  
    popup='Palácio Rio Branco',  
  
    icon=icone  
  
).add_to(mapa)
```

A galeria com os ícones disponíveis pode ser vista em <https://getbootstrap.com/docs/3.3/components/>.

Para um caso de uso prático, imagine que se tenha uma lista de compras no seguinte formato:

```
lista_compras = {"Hambúrguer", "Bacon", "Macaxeira"}
```

Possuísse também a localização de vários mercados, sendo que cada mercado possui um estoque de produtos.

```
mercados = [

    {

        "nome": "Zeca's",

        "localizacao": [-9.975000, -67.809655],

        "estoque": {"Macarrão", "Bacon", "Macaxeira"}

    },

    {

        "nome": "Dois Irmãos",

        "localizacao": [-9.976010, -67.809658],

        "estoque": {"Bacon", "Macaxeira", "Arroz",

"Trigo"}

    },

    {

        "nome": "Ramazon",

        "localizacao": [-9.977005, -67.809656],

        "estoque": {"Bacon", "Macaxeira", "Hanburguer",

"Salaminho"}

    },

]
```

Deseja-se que todos os mercados que tenham a lista de compras completa do mercado tenham marcadores em verde, os demais devem ter marcadores em vermelho.

O primeiro passo é criar o mapa:

```
import folium

mapa = folium.Map(

    [-9.975130, -67.809653]

)
```

O segundo passo é percorrer a lista de mercados e usando o método `issubset()` verificar quais mercados possuem todos os produtos e salvando a cor que se deseja no marcador:

```
for mercado in mercados:

    if (compras.issubset(mercado["estoque"])): #Tem todos os
produtos

        color = "green"

    else:

        color = "red"
```

Agora, basta adicionar os marcadores no laço de repetição:

```
icone = folium.Icon(

    color=color,

    icon='glyphicon glyphicon-thumbs-up',

    icon_color='white',

    angle=0,

)

folium.Marker(
```

```
location=mercado["localizacao"],  
  
popup=mercado["nome"],  
  
icon=icone  
  
) .add_to(mapa)
```

Por fim, é só salvar:

```
mapa.save("mercados.html")
```

CAMADAS

É possível fazer um controle de camadas utilizando os métodos `TileLayer()` e `LayerControl()`.

Por exemplo, para adicionar várias formas de visualização de mapas:

```
import folium  
  
mapa = folium.Map(  
    [-9.975130, -67.809653]  
)  
  
folium.TileLayer('openstreetmap').add_to(mapa)  
  
folium.TileLayer('Stamen Terrain').add_to(mapa)  
  
folium.LayerControl().add_to(mapa)  
  
mapa.save("palacio.html")
```

Ou, no exemplo anterior de compras, os mercados que possuem os produtos e os mercados que não possuem os produtos usando o `MarkerCluster()`:

```
import folium
```

```
from folium.plugins import MarkerCluster

mapa = folium.Map(

    zoom_start=18,

    location=[-9.975130, -67.809653],

)

tem_estoque = MarkerCluster(name='Tem
estoque').add_to(mapa)

nao_tem_estoque = MarkerCluster(name='Não tem
estoque').add_to(mapa)

compras = {"Hanburguer", "Bacon", "Macaxeira"}

mercados = [

    {

        "nome": "Zeca's",

        "localizacao": [-9.975000, -67.809655],

        "estoque": {"Macarrão", "Bacon", "Macaxeira"}

    },

    {

        "nome": "Dois Irmãos",

        "localizacao": [-9.976010, -67.809658],
```

```
        "estoque": {"Bacon", "Macaxeira", "Arroz",
"Trigo"}

    },

    {

        "nome": "Ramazon",

        "localizacao": [-9.977005, -67.809656],

        "estoque": {"Bacon", "Macaxeira", "Hanburguer",
"Salaminho"}

    },

]

for mercado in mercados:

    if(compras.issubset(mercado["estoque"])):#Tem todos
os produtos

        icone = folium.Icon(

            color="green",

            icon='glyphicon glyphicon-thumbs-up',

            icon_color='white',

            angle=0,

        )

        folium.Marker(

            location=mercado["localizacao"],

            popup=mercado["nome"],
```



```
        icon=icone

    ).add_to(tem_estoque)

else:

    icone = folium.Icon(

        color="red",

        icon='glyphicon glyphicon-thumbs-up',

        icon_color='white',

        angle=180,

    )

    folium.Marker(

        location=mercado["localizacao"],

        popup=mercado["nome"],

        icon=icone

    ).add_to(nao_tem_estoque)

folium.LayerControl().add_to(mapa)

mapa.save("MarkerCluster.html")
```

HEATMAP

Um outro mapa muito interessante é o HeatMap, do inglês Mapa de Calor.

Sua utilização mais simples se dá com uma lista de tuplas contendo latitude, longitude e o magnitude como visto abaixo:

```
import folium

from folium.plugins import HeatMap

mapa = folium.Map(

    location=[-9.975130, -67.809653],

)

dados = [

    (-9.975130, -67.809653, 10),

    (-9.976130, -67.809853, 10),

    (-9.973130, -67.807653, 10),

    (-9.978130, -67.806653, 10),

    (-9.979130, -67.807953, 10),

]

HeatMap(

    dados,

).add_to(mapa)

mapa.save("mapacalor.html")
```

EXERCÍCIO PROPOSTO

Uma planilha é entregue à um setor imobiliário contendo as seguintes informações:

CRIM	Taxa de criminalidade per capita por cidade
ZN	Proporção de terras residenciais divididas por lotes acima de 25.000 pés quadrados
INDUS	Proporção de negócios não-varejistas em acres por cidade
CHAS	Variável Charles River
NOX	Concentração de óxidos nítricos (partes por 10 milhões)
RM	Número médio de quartos por habitação
AGE	Proporção de unidades ocupadas pelos proprietários construídas antes de 2010
DIS	Distâncias ponderadas para cinco centros de emprego em Boston
RAD	Índice de acessibilidade às rodovias radiais
TAX	Taxa de imposto sobre a propriedade a cada US\$ 10.000
PTRATIO	Proporção aluno-professor por cidade
LSTAT	% de pessoas pobres
MEDV	Valor médio das casas ocupadas pelos proprietários em US\$ 1.000

O chefe do setor designou que um funcionário se encontra uma correlação entre os dados e a partir deles encontra-se o preço ideal para um imóvel situado em [42.332729, -71.070460] com as seguintes características:

```
"CRIM": [0.0067],  
  
"ZN": [18.0],  
  
"INDUS": [2.31],  
  
"CHAS": [0],  
  
"NOX": [0.87],  
  
"RM": [6.575],  
  
"AGE": [65.2],
```

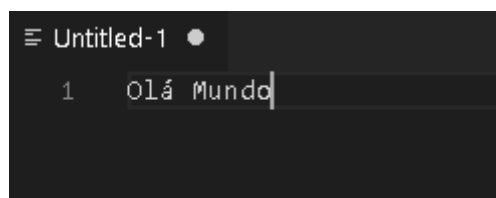
```
"DIS": [4.09],  
  
"RAD": [1],  
  
"TAX": [296],  
  
"PTRATIO": [15.3],  
  
"B": [396.9],  
  
"LSTAT": [4.98]
```

Ele deu ainda a tarefa de criar um mapa de calor com base na latitude e longitude contendo o preço por região e um marcador indicando o local que o imóvel se localiza.

Como os dados eram de análise muito complexa, o funcionário decidiu usar inteligência artificial para prever o preço do imóvel.

CONCEITOS BÁSICOS DE HTML

Todas as páginas da web são construídas em uma linguagem chamada HTML (Hypertext Markup Language ou em português Linguagem de Marcação de Hipertexto).



CRIAÇÃO DE UMA PÁGINA HTML

Para criar uma página HTML, é necessário, inicialmente criar um novo arquivo no VSCode (Arquivo>Novo Arquivo).

Basta escrever agora “Olá Mundo” como visto abaixo:

Por fim, basta salvar com a extensão “.html” como visto abaixo:



Com isso, um arquivo é gerado e já pode ser aberto por um navegador de internet:

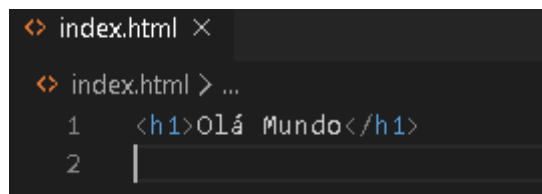
	Nome	Data de modificação	Tipo	Tamanho
Acesso rápido	index	21/11/2019 10:23	Chrome HTML Do...	1 KB
Área de Trabalho				
Downloads				

Ao abrir é possível ver o texto que foi digitado:



TAGS

Para identificar os diferentes componentes de uma página web, usamos as tags. Cada tag tem propriedades que especificam o formato do conteúdo que está dentro dela.



Por exemplo, a tag “**h1**” que faz com que o texto ganhe uma formatação de cabeçalho. Note que o texto “Olá mundo” deve estar entre `<h1>` e `</h1>`, que representam a abertura e o fechamento da tag.

Ao atualizar a página pressionando F5 é possível ver que o texto agora foi alterado:



Olá Mundo

Existem diversas outras tags como por exemplo:

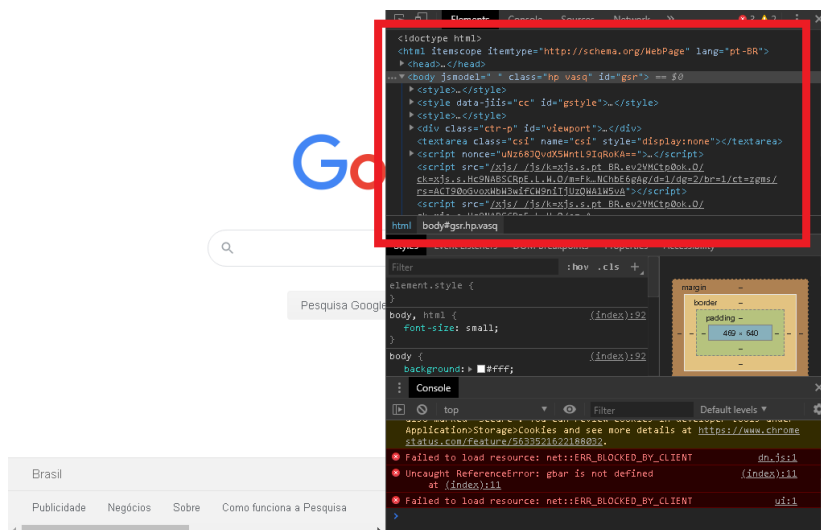
TAGS BÁSICOS	EFEITO
<html></html>	Inicia/termina um documento em HTML.
<head></head>	Define a área de cabeçalho, com elementos não visualizáveis na página.
<body></body>	Define a área visível do documento.
CABEÇALHO	EFEITO
<title></title>	Coloca o nome da página na barra de título da janela.
ATRIBUTOS DO CORPO	EFEITO
<body bgcolor=?>	Cor de fundo.
<body text=?>	Cor de texto.
<body link=?>	Cor dos links.
<body vlink=?>	Cor dos links visitados.
<body alink=?>	Cor do link ativo.
TEXTO	EFEITO
	Texto negrito.
<i></i>	Texto em itálico.
<tt></tt>	Texto estilo máquina de escrever, mono espaçamento.
	Tamanho das letras.
	Cor das letras.
	Define a fonte utilizada.

LINKS	EFEITO
<code></code>	Cria um hiperlink.
<code></code>	Cria um link para o envio de correio.
<code></code>	Cria um 'alvo' dentro de uma página.
<code></code>	Faz a ligação a um 'alvo' presente dentro da mesma página.
FORMATAÇÃO	EFEITO
<code><p></p></code>	Define a área de um parágrafo.
<code><p align=?></code>	Alinhamento de um parágrafo.
<code>
</code>	Insere uma quebra de linha.
<code><div align=?></code>	'Tag' genérico utilizado para formatar blocos de texto.
GRAFISMO	EFEITO
<code></code>	Insere uma imagem.
<code></code>	Alinha uma imagem em relação ao resto do texto.
<code></code>	Define a borda da imagem.
<code><hr></code>	Insere uma linha horizontal.
TABELAS	EFEITO
<code><table></table></code>	Cria uma tabela.
<code><tr></tr></code>	Linha de uma tabela
<code><td></td></code>	Célula individual numa linha.
ATRIBUTOS DAS TABELAS	EFEITO
<code><table border=#></code>	Borda à volta de cada célula.
<code><table cellspacing=#></code>	Espaço entre as células.
<code><table cellpadding=#></code>	Margem interior das células.
<code><table width=# or %></code>	Largura da tabela - em pixels ou percentagem.
<code><tr align=?> or <td align=?></code>	Alinhamento horizontal do conteúdo da célula: "left", "center" ou "right")

<code><tr valign=?></code> or <code><td valign=?></code>	Alinhamento vertical do conteúdo das células. ("top", "middle" ou "bottom")
<code><td colspan=#></code>	Numero de colunas 'percorridas' por uma célula.
<code><td rowspan=#></code>	Numero de linhas 'percorridas' por uma célula.

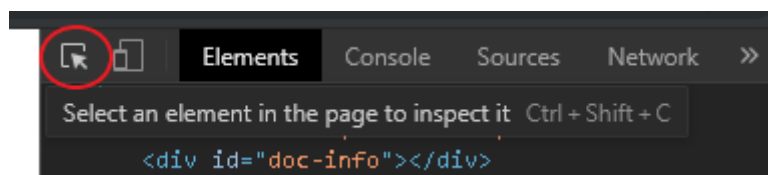
O INSPETOR

Para visualizar melhor isso, abra seu navegador, vá no site do google e pressione F12 para entrar no modo desenvolvedor. Uma tela semelhante à vista abaixo será exibida, com a construção HTML no canto superior direito:



Uma ferramenta interessante nesta interface é o inspetor. Como ele é possível clicar em cima de um elemento e verificar como ele está representado no HTML.

Para usar basta selecionar clicando nele (como visto abaixo) ou usando o atalho Ctrl+Shift+C:



Agora ao clicar sobre um elemento será exibida sua representação em HTML.

CAPÍTULO 19

Django – Python para Web

Django é um framework voltado para o desenvolvimento Web com Python.

Ele pode ser instalado pelo pip:

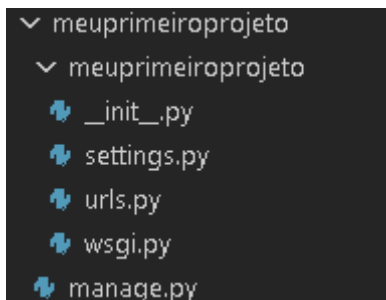
```
pip install django
```

CRIANDO UM PROJETO DJANGO

Para criar um projeto Django deve-se usar o comando:

```
django-admin startproject {nome_projeto}
```

Com isso, uma estrutura de pastas é criada:



INICIANDO O SERVIDOR

Para iniciar um servidor local, deve-se entrar na pasta que contém o arquivo “manage.py” no terminal:

```
cd .\meuprimeiroprojeto\
```

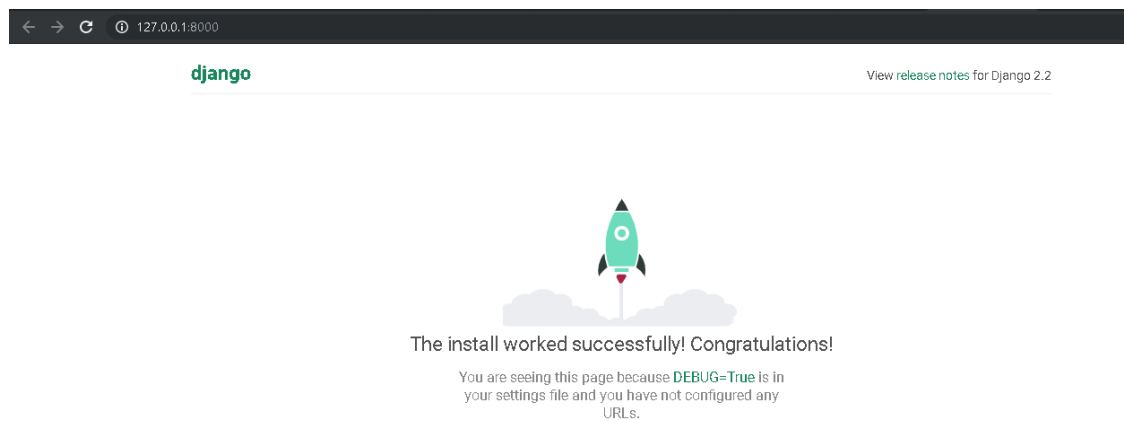
E usar o comando para executar o arquivo:

```
python manage.py runserver
```

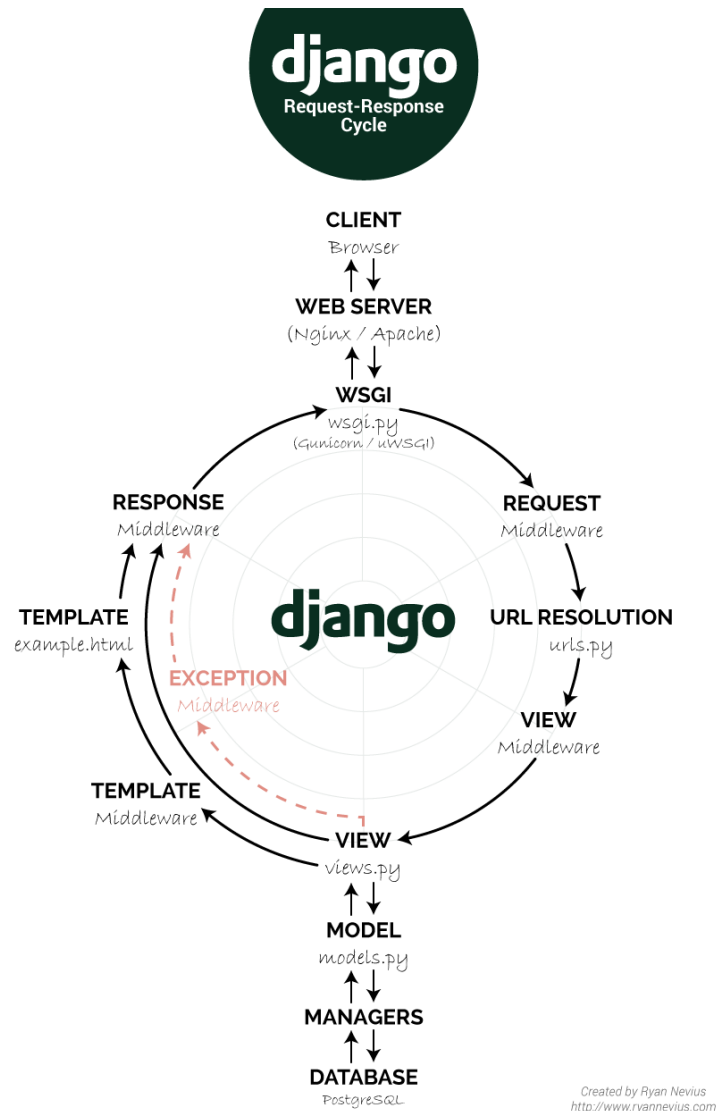
Com isso um servidor está inicializado como mostra a imagem abaixo:

```
System check identified no issues (0 silenced).  
  
You have 17 unapplied migration(s). Your project may not work properly  
contenttypes, sessions.  
Run 'python manage.py migrate' to apply them.  
November 21, 2019 - 11:12:10  
Django version 2.2.7, using settings 'meuprimeiroprojeto.settings'  
Starting development server at http://127.0.0.1:8000/  
Quit the server with CTRL-BREAK.  
□
```

Para acessar o servidor basta acessar o link local no navegador:



CICLO BÁSICO DO DJANGO



Django possui um ciclo como demonstrado na imagem acima.

Ele se resume à:

- O usuário faz uma requisição
- O django processa esta requisição
- O django devolve uma resposta

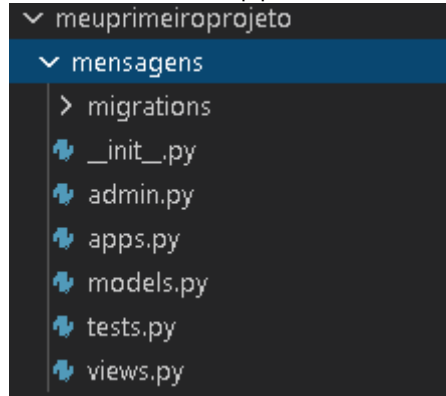
CRIANDO UMA APP

Na filosofia do Django, uma app é um conjunto de scripts que realiza alguma tarefa.

Para iniciar uma App é só usar o comando:

```
python manage.py startapp {nome}
```

Um diretório contendo a App é criado:

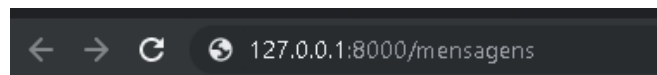


Após criada, deve-se instalar a App no arquivo **settings.py** como visto abaixo:

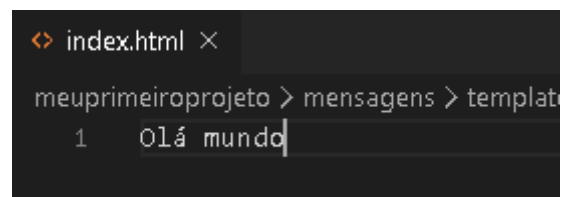
```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'mensagens'
]
```

CRIANDO UMA PÁGINA DE RESPOSTA

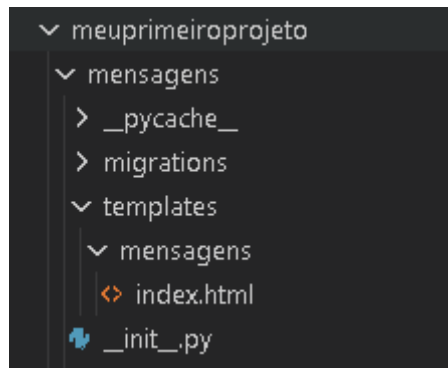
Suponha que o usuário digite a seguinte URL:



Como resposta, deseja-se que o Django devolva uma mensagem contendo “Olá Mundo!”. Para isso, inicialmente é necessário criar a página HTML que queremos mandar de resposta:



Este arquivo deve estar no seguinte diretório:



Após este processo, deve-se criar a chamada da página no arquivo view.py:

```
views.py ×
meuprimeiroprojeto > mensagens > views.py > ...
1  from django.shortcuts import render
2
3  # Create your views here.
4  def chama_pagina(request):
5      return render(request, "mensagens/index.html")
```

Por fim, é necessário dizer ao django que quando o usuário digitar aquela URL deve ser direcionado para esta página alterando o arquivo de URL:

```
from django.contrib import admin
from django.urls import path

from mensagens.views import chama_pagina

urlpatterns = [
    path('admin/', admin.site.urls),
    path('mensagens/', chama_pagina),
]
```

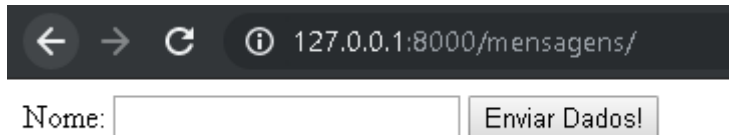
CRIANDO UM FORMULÁRIO

Geralmente em sistemas, o usuário entra com algumas variáveis e recebe uma saída.

Para isso se usam as tags de formulário do html. Um form simples pode ser visto abaixo:

```
<form method="post">
    Nome: <input type="text" name="nome">
    <button type="submit">Enviar Dados!</button>
</form>
```

Com isso o seguinte resultado é esperado:



RECEBENDO UMA RESPOSTA

Ao clicar em “Enviar Dados” é desejável que o django devolva a mensagem “Seu nome é {nome}”.

Para isso é necessário criar uma nova função:

```
def resposta(request):
    nome = request.POST.get("nome")
    return render(request, "mensagens/resposta.html", {"nome":nome})
```

Adicionar uma nova URL:

```
from django.contrib import admin
from django.urls import path

from mensagens.views import chama_pagina, resposta

urlpatterns = [
    path('admin/', admin.site.urls),
    path('mensagens/', chama_pagina),
    path('resposta/', resposta),
]
```

E editar o formulário para redirecionar para a url “resposta”:

```
<form method="POST" action="/resposta/">
    {% csrf_token %}
    Nome: <input type="text" name="nome">
    <input type="submit" value="Enviar Dados!">
</form>
```

EXERCÍCIO PROPOSTO

Criar um sistema que receba os parâmetros de um imóvel como descrito abaixo:

CRIM	Taxa de criminalidade per capita por cidade
ZN	Proporção de terras residenciais divididas por lotes acima de 25.000 pés quadrados
INDUS	Proporção de negócios não-varejistas em acres por cidade
CHAS	Variável Charles River
NOX	Concentração de óxido nítrico (partes por 10 milhões)
RM	Número médio de quartos por habitação
AGE	Proporção de unidades ocupadas pelos proprietários construídas antes de 2010
DIS	Distâncias ponderadas para cinco centros de emprego em Boston
RAD	Índice de acessibilidade às rodovias radiais
TAX	Taxa de imposto sobre a propriedade a cada US\$ 10.000
PTRATIO	Proporção aluno-professor por cidade
LSTAT	% de pessoas pobres

MEDV	Valor médio das casas ocupadas pelos proprietários em US\$ 1.000
------	--

Calcule o seu preço com base nos dados contidos na planilha e entregue ao usuário uma visualização geográfica do imóvel com um mapa de calor dos preços da região.

Conclusão

Uma linguagem é para um programador tal qual uma ferramenta para um marceneiro. Da mesma forma que os conceitos de como pregar e serrar servem para a construção de outras coisas, os conceitos criados neste curso servem como a **base** para a construção das suas aplicações.

Seja no ramo de Data Science trabalhando com Inteligência Artificial, seja no dia a dia para automatizar uma tarefa repetitiva ou até mesmo no desenvolvimento de sistemas: **Python só se torna eficaz quando praticado.**

Os próximos passos consistem em buscar formas de utilização e implementações para estes conceitos. Lembre-se, programação consiste no **emprego de uma linguagem para solucionar um problema.**

Agradeço sua companhia ao longo deste curso e espero você novamente aqui na **Clarify**.