PYTHON BASICS

INTRODUÇÃO

Introdução
O Python

- Criado em 1989 por Guido van Rossum.
- O nome Python vem programa de TV “Monty Python”.
- Python é sucessor da linguagem de programação ABC.
- Parte da sintaxe é derivada do C e Linguagem Haskell (Compreensão de listas, funções anônimas e função map derivadas)
- Leitura Complementar: Python Fluente – Luciano Ramalho

Introdução
Características

- Linguagem alto nível.
- Linguagem interpretada cuja filosofia enfatiza uma programação mais livre e flexível.
- Linguagem de programação multi-paradigma, pois suporta orientação de objeto, programação imperativa e, em menor escala, programação funcional.
- Usa Tipagem Dinâmica forte, significa que o próprio interpretador do Python infere tipo dos dados que uma variável recebe.
- Multiplataforma, roda em Windows, GNULinux, Unix e Mac.
- Ótima para prototipagem.
- Open Source.

Introdução
IDEs

- PyCharm:
  - Desenvolvido especificamente para trabalhar com Python;
  - Possui análise de código (seguindo o guia PEP 8), verificação dinâmica de erros;
  - Tem suporte a desenvolvimento em CoffeeScript, TypeScript, Cython, JavaScript, SQL, HTML/CSS, Angular, NodeJs e outras;
  - Integração com git, Docker, conda, virtualenv...;
  - Preenchimento inteligente de código.

Introdução
IDEs

Visual Studio Code

- VSCode:
  - Multi-plataforma, podendo ser executado no Windows, Linux e Mac;
  - Gratuito e de código aberto;
  - Oferece suporte para mais de 30 linguagens de programação, como JavaScript, C#, C++, Python, Java, PHP, HTML/CSS, R, etc;
  - Possui sistema de extensões para adicionar novas funcionalidades.

Introdução
IDEs

- Jupyter Notebook:
  - Suporta as linguagens Julia, Python e R;
  - Ambiente interativo para ciência de dados;
  - Muito utilizado na exploração de dados;
  - Execução e visualização da saída do código em cada célula;
  - Possibilidade de escrita de texto com markdown.

Introdução
IDEs

- Spyder:
  - Combinação de funcionalidades avançadas de edição, análise, depuração e criação;
  - Possui sistema de plug-ins;
  - Console interativo.

Introdução
Pip

- Gerenciador de pacotes do nativo do Python;
- Utilizada para instalar, atualizar e desinstalar pacotes;
- Principais comandos:
  - **pip install nome_pacote**: instala o pacote especificado;
  - **pip freeze**: lista todos os pacotes instalados e suas versões;
  - **pip install --upgrade nome_pacote**: atualiza o pacote especificado;
  - **pip uninstall nome_pacote**: desinstala o pacote especificado.
  - **pip –r install arquivo_de_requirements.txt**: instala arquivo contendo requirements do projeto.

Introdução
Virtualenv

```
# Instalar pacote virtualenv
!pip install virtualenv

# Criar um Ambiente Virtual
virtualenv python_venv

# Ativar Ambiente Virtual
python_venv\Scripts\activate

#Desativar Ambiente Virtual
deactivate
```

- Ambiente virtual para desenvolvimento;
- Útil para evitar conflito entre diferentes versões de pacotes;
- Necessário quando existem mais de uma versão diferente de python na mesma máquina.
- Existem ferramentas para compartilhamento de ambientes virtuais (Anaconda).
- Em Mac/Linux se usa “source python_venv/Scriptx/activate”

Introdução
Anaconda

- É uma Plataforma Open-Source de Data Science para Python e R.
- Possui Interface Gráfica
- Vem com vários pacotes e funcionalidades para trabalhar com Data Science.
- Possui:
  - Gerenciador de Pacotes
  - Gerenciador de Ambientes Virtuais
  - Jupyter
  - Spyder
- Pip vs Conda: Enquanto pip instala apenas pacotes python, o conda é um gerenciador de pacotes para várias linguagens.

Introdução
Anaconda e Ambientes virtuais

```
# Criando Ambiente com Conda
conda create -n venv python=x.x anaconda

# Ativando ambiente virtual
source activate venv

# Instalando pacotes adicionais ao seu ambiente
conda install -n venv [package]

# Desativar ambiente
source deactivate

# Exportando ambiente
conda env export > venv.yml --name venv

# Carregando ambiente
conda env create -f venv.yml
```

- Conda permite criar e distribuir ambientes virtuais, tanto por linha de comando como por GUI.

ELEMENTOS PYTHON

BOSCH

Elementos Python
Intro

```
a = [1, 2, 3, 4]
b = a

a += [5]

print(b)
```

```
[1, 2, 3, 4, 5]
```

```
print(id(a))
print(id(b))
```

```
2691055813440
2691055813440
```

```
print(a == b)

print(a is b)
```

```
True
True
```

```
a = {"a": 2}
b = {"a": 2}

print(b is a )

print(id(b))
print(id(a))

print(b == a)
```

```
False
2691056936384
2691056936704
True
```

```
a = None

print(id(a))
print(id(None))
```

```
140721338419328
140721338419328
```

- As variáveis em Python não são caixas, mas rótulos.
- Variáveis em python guardam a referência dos objetos.
- As variáveis são criadas depois dos objetos.
- O operador **is** compara identificação enquanto **==** compara valores
- Comparação com **None**, verifica se a variável referencia o objeto **None.**

Elementos Python
Intro

- Tipos simples
  - Tipos Numéricos (Int, float, complex) var = 1 ou 1.2 ou 3 + 4j .
  - Texto (String) var = “Texto” ou ‘Texto’.
- Tipos Containers:
  - Listas = [1,2,3,4]
  - Tuplas = (1,2,3,4)
  - Dicionários = {“a” : 1}
- Sets
- Métodos Mágicos (“*dunder*”)

Elementos Python
String

```
frase = "Olá! Tudo bem?"
print(frase)
print(type(frase))
print(frase[2])
print(frase[0:3])
```

```
Olá! Tudo bem?
<class 'str'>
á
Olá
```

```
print("tudo" in frase)
print("Tudo" in frase)
```

```
False
True
```

- String é o tipo de dado utilizado para caracteres.
- É uma estrutura semelhante a um array;
- Assim como em uma lista, também é possível utilizar o **in** para verificar se existe aquele elemento na string.

Elementos Python
String: Funções básicas

```
resp = "Estou bem!"
frase = frase +" "+ resp
print(frase)
```
```
Olá! Tudo bem? Estou bem!
```

```
print("Tamanho da frase: ", len(frase))
```
```
Tamanho da frase:  25
```

```
print(frase.isalpha())
print(frase[1].isalpha())
```
```
False
True
```

```
fruta = "    maçã    "
print("Eu adoro", fruta.strip(), "verde!")
```
```
Eu adoro maçã verde!
```

```
frase = "Olá! Tudo bem?"
frase = frase.replace("Olá", "Bom dia")
print(frase)
```
```
Bom dia! Tudo bem?
```

- Para concatenar basta utilizar o **+**;
- **len( )**: Retorna o tamanho da string;
- **isalpha( )**: retorna True se todos os valores na string são letras;
- **strip( )**: remove os espaços entre os caracteres;
- **replace( )**: localiza e substitui a string por outra.

Elementos Python
String: Funções básicas

```
frase = "LETRAS EM CAIXA ALTA"
frase = frase.lower()
print(frase)
```
```
letras em caixa alta
```

```
frase = "letras minúsculas"
frase = frase.upper()
print(frase)
```
```
LETRAS MINÚSCULAS
```

```
frase = "Dia 3 é meu aniversário"
print(frase.isnumeric())
print(frase[4].isnumeric())
```
```
False
True
```

```
lista_nome = ["Ana", "Jose", "Pedro"]
nomes = "#".join(lista_nome)
print(nomes)
```
```
Ana#Jose#Pedro
```

- **lower( )**: converte toda a string para minúsculas;
- **upper( )**: converte toda a string em maiúsculas;
- **isnumeric( )**: retorna True se todos os valores da string são números;
- **join( )**: pega todos os elementos iteráveis e os une como uma string, utilizando um separador (string especificada).

Elementos Python
String: Funções básicas

```
frase = "Oi, meu nome é Ana, tudo bem?"
frase_1 = frase.split()
frase_2 = frase.split(", ")
print(frase_1)
print(frase_2)
```

```
['Oi,', 'meu', 'nome', 'é', 'Ana,', 'tudo', 'bem?']
['Oi', 'meu nome é Ana', 'tudo bem?']
```

```
frase = "O preço  é {preco:.2f} reais!"
print(frase.format(preco = 23.99))
```

```
O preço  é 23.99 reais!
```

```
frase = "Valor 1: {v1:.1f} e valor 2: {v2:.3f}"
print(frase.format(v1=10, v2=2.12345))
```

```
Valor 1: 10.0 e valor 2: 2.123
```

- **split( )**: divide a string utilizando um separador e retorna uma lista. Se nenhum separador foi especificado, será utilizado o espaço;
- **format(*value1, value2, ...*)**: formata o valor, ou valores, especificado e o insere na string.

Elementos Python
Lista

```
lista_vazia = []
print(lista_vazia)
```

```
[]
```

```
lista = ['gato', 'cachorro', 'passaro', 'peixe']
print(lista)
```

```
['gato', 'cachorro', 'passaro', 'peixe']
```

- Lista em Python é uma sequência de elementos ordenada.
- É uma estrutura de dados mutável.
- Para criar listas em Python basta apenas declarar uma variável e atribuir os colchetes à ela;
- Para criar uma lista com conteúdo, basta colocar os valores dentro dos colchetes e separados por vírgulas;

Elementos Python
Lista

```
lista_variada = ['item1', 1, 1.5, [1, 2, 3, 4, 5], range(4), {"a" : 1}]

print(lista_variada)
for item in lista_variada:
    print(f" {type(item)} : {item} ")
```

```
['item1', 1, 1.5, [1, 2, 3, 4, 5], range(0, 4), {'a': 1}]
 <class 'str'> : item1
 <class 'int'> : 1
 <class 'float'> : 1.5
 <class 'list'> : [1, 2, 3, 4, 5]
 <class 'range'> : range(0, 4)
 <class 'dict'> : {'a': 1}
```

- Listas podem conter todo tipo de elemento.
- Listas são um tipo de *sequências container*, que armazenam a referência dos objetos que elas contêm.

Elementos Python
Lista: Funções básicas

```
lista.append("papagaio")
print(lista)
```

```
['gato', 'cachorro', 'passaro', 'peixe', 'papagaio']
```

```
lista.insert(3, "cobra")
print(lista)
```

```
['gato', 'cachorro', 'passaro', 'cobra', 'peixe', 'papagaio']
```

```
lista.pop()
print(lista)
```

```
['gato', 'cachorro', 'passaro', 'cobra', 'peixe']
```

```
lista.pop(3)
print(lista)
```

```
['gato', 'cachorro', 'passaro', 'peixe']
```

- Append: inserir no final da lista;
- Insert: insere na posição especificada;
- Pop: Remove o último elemento da lista. Se for passado o index, remove do index especificado;

Elementos Python
Lista: Funções básicas

```
lista.reverse()
print(lista)
```

```
['peixe', 'passaro', 'cachorro', 'gato']
```

```
lista.sort()
print(lista)
```

```
['cachorro', 'gato', 'passaro', 'peixe']
```

```
lista.count('gato')
```

```
1
```

- Reverse: Inverte a ordem da lista;
- Sort: Ordena a lista em ordem crescente;
- Count: Retorna a quantidade de valores daquele elemento

Elementos Python
Tupla

```
tupla_1 = (1, "a", "345", 2.5)
print("Elementos da tupla:")
print(tupla_1)
print("Tipo:", type(tupla_1))
print("a" in tupla_1)
```

```
Elementos da tupla:
(1, 'a', '345', 2.5)
Tipo: <class 'tuple'>
True
```

- A diferença da tupla para uma lista, é que a tupla é imutável. Ou seja, depois de criada, seus elementos não podem ser removidos ou adicionados novos;
- A tupla geralmente é mais utilizada quando estamos trabalhando com dados mais heterogêneos;
- Podem ser usadas como registros sem nomes de campos. Ex: (lat, long)

Elementos Python
Tupla Nomeada

```
from collections import namedtuple

City = namedtuple("City", "name country pop latlong")
tokyo = City("Tokyo", country="JP", pop=36.933, latlong=(35.689722, 139.691667))

print(tokyo)
print(tokyo.pop)
print(tokyo[-1])

print(tokyo._asdict())
```

```
City(name='Tokyo', country='JP', pop=36.933, latlong=(35.689722, 139.691667))
36.933
(35.689722, 139.691667)
{'name': 'Tokyo', 'country': 'JP', 'pop': 36.933, 'latlong': (35.689722, 139.691667)}
```

- collections.namedtuple é uma subclasse de Tuple melhorada com nomes de campos e um nome de classe.
- Ao serem convertidas para Dict, namedtuple gera um OrderedDict que é um Dicionário com ordenamento preservado.

Elementos Python
Dicionário

```
tels = {'Ana': '999-999', 'Pedro': '123-456',
              'Jose': '777-777', 'Sara': '876-876'}
print("Telefone do Jose:", tels['Jose'])
```

```
Telefone do Jose 777-777
```

```
contatos = [('Ana', '999-999'), ('Pedro', '123-456'),
            ('Jose', '777-777'), ('Sara', '876-876')]
cont_dict = dict(contatos)
print(type(cont_dict))
print(cont_dict)
```

```
<class 'dict'>
{'Ana': '999-999', 'Pedro': '123-456',
'Jose': '777-777', 'Sara': '876-876'}
```

- Estrutura formada por uma coleção de dados em padrão chave e valor;
- Em outras linguagens, estruturas semelhantes recebem o nome de hashmap, map ou array associativo;
- A estrutura básica é **dict = {chave: valor}**;
- É possível também transformar uma lista de valores em um dicionário.

Elementos Python
Dicionário: Funções Básicas

```
print(tels.get('Ana', 'Contato não encontrado'))
print(tels.get('Maria', 'Contato não encontrado'))
```

```
999-999
Contato não encontrado
```

```
tels['Maria'] = '555-123'
print(tels)
```

```
{'Ana': '999-999', 'Pedro': '123-456', 'Jose': '777-777',
  'Sara': '876-876', 'Maria': '555-123'}
```

```
print(tels.pop('Ana', 'Contato não encontrado'))
print(tels.pop('Ana', 'Contato não encontrado'))
```

```
999-999
Contato não encontrado
```

- **get( )**: Função usada para acessar o valor passando a chave como parâmetro, pode ser passada uma mensagem para aparecer caso não exista aquela chave no dicionário;
- Para adicionar um novo valor, basta criar a chave e associa-la a um valor;
- **pop( )**: Função para remover um elemento passando como parâmetro a chave.

Elementos Python
Dicionário: Variações

```python
from collections import OrderedDict, UserDict, defaultdict

# regular dict
d1 = dict({"a": 1, "b": 2, "c": 3})
d2 = dict({"c": 3, "b": 2,  "a": 1})
print(d1 == d2)

# OrderedDict
od1 = OrderedDict({"a": 1, "b": 2, "c": 3})
od2 = OrderedDict({"c": 3, "b": 2,  "a": 1})
print(od1 == od2)
```

```
True
False
```

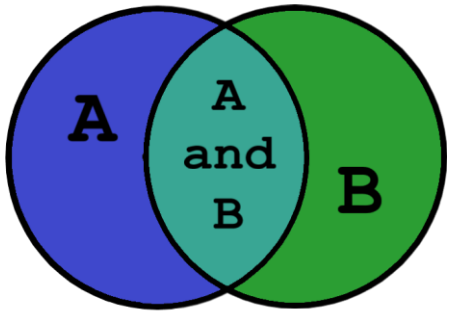
```python
# UserDict
"""
Caso o usuário queira sobrescrever a classe dict, é recomendado usar UserDict.
 "major caveat: the code of the built-ins (written in C) does not call special (magic)
  methods overridden by user-defined classes".
"""
class MyDict(UserDict):
    pass
```

```python
# defaultdict
"""
The main difference between defaultdict and dict is that when you try to access or modify a key
that's not present in the dictionary, a default value is automatically given to that key.
"""
d = dict({"a": 1}) # mesmo que {"a": 1}
dd = defaultdict(dict)
dd = defaultdict(list)
dd = defaultdict(tuple)
dd = defaultdict(int)
dd = defaultdict(bool)

try:
    print(d['b'])
except KeyError as e:
    print(e)

dd['2']
```

```
'b'

False
```

- Existem variações de dicionários em Python, **OrderedDict**, **UserDict** e **defaultdict**.

Elementos Python
Sets



- Sets é uma ferramenta muito útil em python.
- Sets podem ser modificados porém seus elementos são imutáveis.
- *Frozensets são conjuntos imutáveis*

```
l1 = [1, 2, 1, 1, 2, 3, 4, 5, 6, 6, 6, 6, 7, 7]
l2 = [1, 8, 8, 8, 9, 9, 2, 3]

set_l1 = set(l1)

print("Conjunto: ", set_l1)
print("Tamanho do Conjunto : ", len(set_l1))
print('l1 intersect l2 : ', set_l1.intersection(set(l2)))
print('l1 difference l2 : ', set_l1.difference(set(l2)))
```

```
Conjunto:  {1, 2, 3, 4, 5, 6, 7}
Tamanho do Conjunto :  7
l1 intersect l2 :  {1, 2, 3}
l1 difference l2 :  {4, 5, 6, 7}
```

Elementos Python
Métodos Mágicos (“*dunder*”)

```
l1 = [1, 2, 3, 4]
print(l1.__getitem__(0).__add__(2),  "==", l1[0] + 2)
print(l1.__len__())
print(l1.__repr__())
```

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

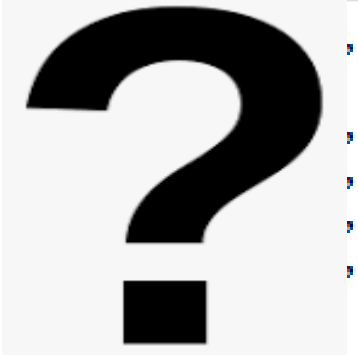
```
dir(list)
['__add__', '__class__', '__contains__', '__delattr__',
'__delitem__', '__dir__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__',
'__iadd__', '__imul__', '__init__', '__init_subclass__', '__iter__',
'__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__reversed__',
'__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__',
'__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend',
'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

- “Métodos mágicos” é uma gíria associada aos métodos especiais da linguagem python, Ex: __getitem__.
- Os métodos especiais são operações básicas em objetos chamados pelo Interpretador de Python.
- São chamados com dois underscores “*dunders*”.
- Permitem iterações com construções básicas da linguagem. Ex: Iteração, Coleção, Sobrecarga de operadores, Construtores e Destrutores de objetos.
- A invocação dos *dunders* é feita internamente pelo interpretador do Python.
- Utilizados na definição das classes para sobrecargas de função.

Elementos Python
Inplace Addition

```python
a = 2
b = 3
print(f"a : {a}")
print(f"b : {b}")
z = a + b
print(f"z : {z}")
print(f"a : {a}")
p = a
p += b
print(f"p : {p}")
print(f"a : {a}")
```

```python
a = [1, 2, 3, 4, 5]
b = [1, 2, 3]
print(f"a : {a}")
print(f"b : {b}")
z = a + b
print(f"z : {z}")
print(f"a : {a}")

p = a
p += b
print(f"p : {p}")
print(f"a : {a}")
```



```
, 5]

, 5, 1, 2, 3]
, 5]
, 5, 1, 2, 3]
, 5, 1, 2, 3]
```

Elementos Python
Inplace Addition

```python
def try_hash(x):
    try:
        print(hash(x))
    except TypeError:
        print(f"Unhashable type {type(x)}")

# Immutables
try_hash(5)
try_hash(1.1)
try_hash("oi")
try_hash(True)
try_hash((1, 2))
try_hash(frozenset([1, 2, 3]))

# Mutables
print(" ")
try_hash([1,2,3])
try_hash(set([1,2,3]))
try_hash({"a": 1})
```

```
5
230584300921369601
-5143190994707115136
1
-3550055125485641917
-272375401224217160

Unhashable type <class 'list'>
Unhashable type <class 'set'>
Unhashable type <class 'dict'>
```

- Existem dois tipos de operações “dunder” de adição em Python.
- O método __add__, adição simples. Toma 2 argumentos e retorna a soma entre eles, sem alterar nenhum dos argumentos. *Ex: a + b.*
- O método __iadd__, também toma 2 argumentos, porém ele faz uma transformação **inplace** no primeiro argumento, onde ele salva o resultado da soma entre os 2 argumentos. *Ex: a += b.*
- Este tipo de operação parece inofensiva mas pode causar problemas ao utilizar elementos mutáveis como argumento.

Elementos Python
Inplace Addition

```python
a = [1, 2, 3, 4, 5]
b = [1, 2, 3]
print(f"a : {a}")
print(f"b : {b}")
z = a + b
print(f"z : {z}")
print(f"a : {a}")

# cópia rasa
p = a.copy()
p += b
print(f"p : {p}")
print(f"a : {a}")
```

```
a : [1, 2, 3, 4, 5]
b : [1, 2, 3]
z : [1, 2, 3, 4, 5, 1, 2, 3]
a : [1, 2, 3, 4, 5]
p : [1, 2, 3, 4, 5, 1, 2, 3]
a : [1, 2, 3, 4, 5]
```

- Uma forma de se precaver de possíveis problemas com inplace addition em elementos mutáveis é utilizar cópia rasa **copy**.

CONTROLE DE FLUXO E ESTRUTURAS DE REPETIÇÃO

BOSCH

Controle de Fluxo e Estruturas de repetição.
Condicional

```
var = 42
if type(var) == str:
    print("Tipo string")
elif type(var) == int:
    print("Tipo inteiro")
else:
    print("Outro tipo")
```

```
Tipo inteiro
```

- **If**: Condicional padrão, onde é colocado a condição a ser testada. Ao final da condição é adicionado um **:**
- **Else**: O “se não” padrão, não precisa de condicional e só entrará no bloco de código se a condição anterior não for verdadeira.
- **Elif**: É a junção de um **else** com o **if**, necessita de uma condicional.

Controle de Fluxo e Estruturas de repetição.
Laço de repetição: For

```
lista = ['gato', 'cachorro', 'passaro', 'peixe']

for item in lista:
    print(item)
```

```
gato
cachorro
passaro
peixe
```

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

```
0
1
2
3
4
```

- Utilizado para percorrer sequencias previamente conhecidas, como: strings, listas, tuplas, buffers, etc;
- A cada interação, o laço passa por um item da lista, como no exemplo;
- Podemos utilizar o for com algumas funções, como por exemplo o **range**;

Controle de Fluxo e Estruturas de repetição.
Laço de Repetição: For

```
lista = ['gato', 'cachorro', 'passaro', 'peixe']

for key, value in enumerate(lista):
    print("Indicie:",key," Valor:", value)
```

```
Indicie: 0  Valor: gato
Indicie: 1  Valor: cachorro
Indicie: 2  Valor: passaro
Indicie: 3  Valor: peixe
```

```
for i in range(5):
    if i == 1:
        continue
    print(i)
    if i == 3:
        break
```

```
0
2
3
```

- Caso seja necessário também utilizar o indicie da lista, basta utilizar a função **enumerate**, onde será possível ter o indicie e o valor da lista naquela posição.
- **continue** serve para pular a iteração atual.
- **break.**

Controle de Fluxo e Estruturas de repetição.
Laço de Repetição: While

- O laço while que executa um conjunto de instruções enquanto a condição for verdadeira

```
contador = 0
while (contador < 5):
    print(contador)
    contador += 1
```

```
0
1
2
3
4
```

Controle de Fluxo e Estruturas de repetição.
Laço de Repetição

- Em Python é possível utilizar o comando **else** logo após o final de um laço (tanto no **for** quanto no **while**) para que execute determinado bloco de comandos.

```
contador = 0
while (contador < 5):
    print(contador)
    contador += 1
else:
      print("Loop while concluído")
```

```
0
1
2
3
4
Loop while concluído
```

```
lista = ['gato', 'cachorro', 'passaro', 'peixe']

for item in lista:
    print(item)
else:
    print("Laço concluído")
```

```
gato
cachorro
passaro
peixe
Laço concluído
```

EXCEÇÕES

Exceções
EAFP e Duck-Typing

É mais fácil pedir perdão do que pedir permissão

EAFP

Easier to ask for forgiveness than permission. This common Python coding style assumes the existence of valid keys or attributes and catches exceptions if the assumption proves false. This clean and fast style is characterized by the presence of many `try` and `except` statements. The technique contrasts with the LBYL style common to many other languages such as C.

Fonte: https://docs.python.org/3/glossary.html#term-eafp

duck-typing

A programming style which does not look at an object's type to determine if it has the right interface; instead, the method or attribute is simply called or used ("If it looks like a duck and quacks like a duck, it must be a duck.") By emphasizing interfaces rather than specific types, well-designed code improves its flexibility by allowing polymorphic substitution. Duck-typing avoids tests using `type()` or `isinstance()`. (Note, however, that duck-typing can be complemented with abstract base classes.) Instead, it typically employs `hasattr()` tests or EAFP programming.

Fonte: https://docs.python.org/3/glossary.html#term-duck-typing

Exceções
EAFP e Duck-Typing

Programação Defensiva

```
import os


arquivo = 'arquivo.csv'
if os.path.isfile(arquivo):
    with open(arquivo, "r") as arq:
        for line in arq.read():
            print(line)
else:
    print("Arquivo não encontrado!")
```

```
Arquivo não encontrado!
```

Jeito "Pythonico"

```
try:
    with open(arquivo, "r") as ard:
        for line in arq.read():
            print(line)
except FileNotFoundError as e:
    print(e)
```

```
[Errno 2] No such file or directory: 'arquivo.csv'
```

Exceções
Try - Except

```
try:
    # Código a ser testado
    pass
except Exception as e: # Não usar Exceptions Genéricas
    print(e)
```

```
try:
    pass
except KeyError as e:
    print(e)
except TypeError as e:
    print(e)
except FileNotFoundError as e:
    print(e)
except NotImplemented as e:
    print(e)
except RuntimeError as e:
    print(e)
```

```
try:
    if True:
        raise KeyError()
except KeyError:
    print("Ok!")
```

```
Ok!
```

- Exceções em Python são implementadas da seguinte forma.
- É recomendado não utilizar Exceções genéricas como “Exception”, o ideal utilizar a Exceção que o erro irá retornar.
- É possível ser feita implementação de Exceções aninhadas em Python.
- Caso necessário é possível levantar exceções específicas para o seu código

Exceções
Try - Except

```
class LowSalaryException(Exception):
    """Exception raised for errors in the input salary.

    Attributes:
        salary -- input salary which caused the error
        message -- explanation of the error
    """

    def __init__(self, salary, message="Salário esta baixo!! Ainda nao consigo comprar minha RTX 3090!"):
        self.salary = salary
        self.message = message
        super().__init__(self.message)

    def __str__(self):
        return f'{self.salary} -> {self.message}'

salary = int(input("Qual o seu salário? : "))
if salary < 1e6:
    raise LowSalaryException(salary)
```

```
Qual o seu salário? : 123456

---------------------------------------------------------------------------
LowSalaryException                        Traceback (most recent call last)
<ipython-input-216-bfb6b6422fa0> in <module>
     17 salary = int(input("Qual o seu salário? : "))
     18 if salary < 1e6:
---> 19     raise LowSalaryException(salary)

LowSalaryException: 123456 -> Salário esta baixo!! Ainda nao consigo comprar minha RTX 3090!
```

- É Possível criar suas próprias exceções para atender funcionalidades específicas de um programa.
- Custom Exceptions em python podem receber valores de variáveis deixando a mensagem de erro mais customizável.

BIBLIOTECAS

BOSCH

Bibliotecas
Módulos Nativos ao Python

- **OS**:
  - Fornece diversas funções para interagir com sistema operacional;
- **Shutil**:
  - Para gerenciamento de arquivos e diretórios;
- **Glob**:
  - Utilizado para retornar todos os nomes de caminho especificado;
  - Por exemplo **glob.glob('Imagens/*.gif')** irá retornar uma lista com o caminho de todos os arquivos da pasta Imagens que forem do tipo .gif.

Bibliotecas
Exemplo prático

```
import os
import sys
import glob
import shutil

# Retorna o diretório atual de trabaho
print("Diretório atual:", os.getcwd())
#Cria um novo diretório
os.mkdir('C:/Users/Public/Documents/Exercicio')
# Muda o diretório atual de trabalho
os.chdir('C:/Users/Public/Documents/Exercicio')
print("Diretório atual:", os.getcwd())

# Criando arquivos .txt
for i in range(0,10):
    n = i+1
    name = "File_" + str(n) + ".txt"
    file = open(name,"w+")
    file.close()
```

```
Diretório atual: C:\Users\.      \Documents\Exemplos
Diretório atual: C:\Users\Public\Documents\Exercicio
```

```
files = glob.glob('*.txt')
print("Arquivos .txt criados:")
for file in files:
    print(file)
```

```
Arquivos .txt criados:
File_1.txt
File_10.txt
File_2.txt
File_3.txt
File_4.txt
File_5.txt
File_6.txt
File_7.txt
File_8.txt
File_9.txt
```

Bibliotecas
Exemplo prático

```
# Mudando o formato dos arquivos
folder = "C:/Users/Public/Documents/Exercicio"
for filename in os.listdir(folder):
    infilename = os.path.join(folder,filename)
    if not os.path.isfile(infilename): continue
    oldbase = os.path.splitext(filename)
    newname = infilename.replace(".txt", ".csv")
    output = os.rename(infilename, newname)

# Movendo para outra pasta
files = glob.glob('*.csv')
os.mkdir('C:/Users/Public/Documents/Exercicio/new_files')
for file in files:
    shutil.move(file, 'new_files')
```

O glob pode ser utilizado como:

‘***.***’: retorna a lista de todos os arquivos.

‘**name.***’: retorna todos os arquivos com o nome especificado, independente do tipo.

‘***.txt**’: retorna todos os arquivos .txt independente do nome.

AVANÇADOS

BOSCH

Avançados
Function Annotations

- Function Annotations surgiu na versão 3.0 do Python.
- São utilizados como uma forma de documentação
- Melhoram o uso de IDEs e Linters.
- Ajudam a manter uma estrutura mais limpa

```
def foo(a:"int", b:"float"=5.0)  -> "int":
    return int(a+b)

print(foo(5, 2.1))

print(foo.__annotations__)

import inspect
print(inspect.getfullargspec(foo))
```

```
7
{'a': 'int', 'b': 'float', 'return': 'int'}
FullArgSpec(args=['a', 'b'], varargs=None, varkw=None, defaults=(5.0,), kwonlyargs=[], kwonlydefaults=None, annotations={'re
turn': 'int', 'a': 'int', 'b': 'float'})
```

Orientação Objeto
Decorator

```
# First Class Objects
def shout(text):
    return text.upper()
print(shout('Hello'))
yell = shout
print(yell('Hello'))

def max_or_min(func, list):
    return func(list)

M = max_or_min
print(M(max, [1,2,3,4]))
print(M(min, [1,2,3,4]))

def func_outter(a):
    def func_inner(a):
        return a + 3
    return func_inner(a)

print(func_outter(7))
```

```
HELLO
HELLO
4
1
10
```

- Em Python funções são Objetos de Primeira Classe, o que significa que podem ser utilizados como argumentos de uma função.
- Propriedades de um Objeto de Primeira Classe:
  - Uma função é uma instância do tipo Objeto:
  - Uma função pode ser salva em uma variável
  - Uma função pode ser passada como parâmetro para outra função
  - Você pode retornar uma função de uma função

Orientação Objeto
Decorator

```python
# definindo um decorator
def hello_decorator(func):

    # inner1 é a função Wrapper em que
    # é chamado o argumento

    # não é necessário passar a variável func para inner1
    # pois inner1 tem acesso ao escopo local de hello_decorator
    def inner1():
        print("Olá, isto acontece antes da execução da função")
        # chamando a função func denntro do wrapper
        func()
        print("Função executada")
    return inner1


# definindo função a ser chamada dentro do wrapper
def function_to_be_used():
    print("Função à ser executada no wrapper")


# passando função function_to_be_used dentro do decorator
function_to_be_used = hello_decorator(function_to_be_used)


# chamando a função
function_to_be_used()
```

```
Olá, isto acontece antes da execução da função
Função à ser executada no wrapper
Função executada
```

- Decorators em Python, permitem alterar o comportamento de uma função (ou classe) sem alterar a mesma.
- Em outras palavras, uma função A é utilizada como um argumento para outra função B, e chamada dentro de uma função Wrapper.

```python
# definindo função a ser chamada dentro do wrapper
@hello_decorator
def function_to_be_used():
    print("Função à ser executada no wrapper")


function_to_be_used()
```

```
Olá, isto acontece antes da execução da função
Função à ser executada no wrapper
Função executada
```

Orientação Objeto
Exemplo - Criando um Decorator

- Nesse exemplo, o nosso decorator é a função calcula_duracao, que recebe como parâmetro a função onde será aplicada o decorator.
- A função wrapper é que a que está envolvida pela função do decorator, e ela pode ser alterada de acordo com a necessidade.

Avançados
Compreensão de Listas

```
a = list(range(int(1e6)))

@calcula_duracao
def soma_for():
    x = a.copy()
    for i, val in enumerate(x):
        if val %2 == 0:
            x[i] = val+1
    return x
soma_for()


@calcula_duracao
def soma_lcomp():
    return [val+1 if val%2 == 0 else val for val in a]
soma_lcomp()
```

```
Nome da função: soma_for
Tempo total de execução: 0.10874366760253906
Nome da função: soma_lcomp
Tempo total de execução: 0.09877133369445801
```

- Uma forma mais performática de criar ou iterar sobre uma lista é por meio de Compreensão de Listas.
- Ao perder legibilidade temos um ganho substancial de velocidade.
- Por convecção não recomenda-se utilizar uma compreensão de listas que exceda 1 linha de Código.

Avançados
Compreensão de Dicionários

```
dict1 = {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}
double_dict1 = {k:v*2 for (k,v) in dict1.items()}
print(double_dict1)



# Initialize `fahrenheit` dictionary
fahrenheit = {'t1':-30, 't2':-20, 't3':-10, 't4':0}

#### Alternativa com For Loops + Lambda

#Get the corresponding `celsius` values
celsius = list(map(lambda x: (float(5)/9)*(x-32), fahrenheit.values()))
#Create the `celsius` dictionary
celsius_dict = dict(zip(fahrenheit.keys(), celsius))
print(celsius_dict)

#### Compreensão de dicionários
# Get the corresponding `celsius` values and create the new dictionary
celsius = {k:(float(5)/9)*(v-32) for (k,v) in fahrenheit.items()}
print(celsius_dict)
```

```
{'a': 2, 'b': 4, 'c': 6, 'd': 8, 'e': 10}
{'t1': -34.44444444444444, 't2': -28.88888888888889, 't3': -23.333333333333336, 't4': -17.77777777777778}
{'t1': -34.44444444444444, 't2': -28.88888888888889, 't3': -23.333333333333336, 't4': -17.77777777777778}
```

- Da mesma forma podemos criar dicionários utilizando a sintaxe de Compreensão de dicionários.
- Uma alternativa para evitar usar **for loop** e funções **lambda**

Avançados
Geradores

```
a = list(range(int(1e10)))
```

```
---------------------------------------------------------------------------
MemoryError                               Traceback (most recent call last)
<ipython-input-98-431faba3df50> in <module>
----> 1 a = list(range(int(1e10)))

MemoryError:
```

```
val = int(1e10)

def val_gen(x):
    for i in range(x):
        yield i

vg = val_gen(val)

print(vg)
print(type(vg))

# Iterando sobre o generator
soma = 0
for i in vg:
    soma+=i
    break
print(soma)
```

```
<generator object val_gen at 0x000002063EDE5350>
<class 'generator'>
0
```

- Funções geradoras, são úteis quando precisamos criar listas de elementos sem alocar espaço na memória.
- Geradores retornam os chamados “lazy iterator”, eles iteram sob uma lista. Porém, ao contrário de listas eles não armazém o conteúdo na memória.
- Úteis ao ler arquivos extensos. Gerar uma lista de imagens para um dataset de mais de 1.000.000 de imagens

Avançados
Compreensão de geradores

```
val = int(1e10)

vg = (i for i in range(val))

print(vg)
print(type(vg))
```

```
<generator object <genexpr> at 0x000001F398A5D270>
<class 'generator'>
```

- Geradores também possibilitam utilizar compreensão.

ORIENTAÇÃO OBJETO

BOSCH

Orientação Objeto
Criação da classe

```
class Filme():
    def __init__(self, titulo, ano, duracao):
        self.titulo = titulo
        self.ano = ano
        self.duracao = duracao
    def mostrar_filme(self):
        print("Titulo:", self.titulo)
        print("Ano:", self.ano)
        print("Ducação:", self.duracao, "minutos")
    def __del__(self):
        print("*****")
        print("Destruindo o objeto")
        print("*****")
```

```
filme = Filme("Titulo", 2033, 95)
filme.mostrar_filme()
del filme
```

```
Titulo: Titulo
Ano: 2033
Ducação: 95 minutos
*****
Destruindo o objeto
*****
```

- Nome da classe com primeira letra maiúscula;
- O **construtor** da classe é o **__init__**;
- O **self** representa a instância da classe, assemelhando-se ao **this** no Java e C++.
- O **destrutor** da classe é o **__del__**, ele é executado ao destruir a instância de uma classe;

Orientação Objeto
Criação da classe

```
import datetime

now = datetime.datetime.now()

print(str(now))
print(repr(now))
```

```
2021-04-15 09:14:00.310757
datetime.datetime(2021, 4, 15, 9, 14, 0, 310757)
```

```
class Person:

    def __init__(self, person_name, person_age):
        self.name = person_name
        self.age = person_age

    def __str__(self):
        return f'Person name is {self.name} and age is {self.age}'

    def __repr__(self):
        return f'Person(name={self.name}, age={self.age})'


p = Person("Marco", 28)

print(p.__str__())
print(p.__repr__())
```

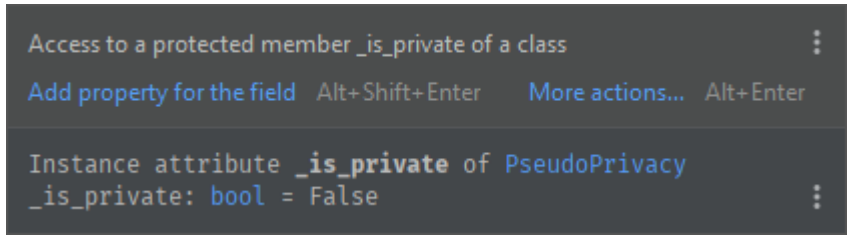
```
Person name is Marco and age is 28
Person(name=Marco, age=28)
```

- **__repr__** Retorna uma string que representa o objeto que o desenvolvedor quer ver.
- **__str__** Retorna uma string que representa o que o usuário quer ver (readable).

Orientação Objeto
Encapsulamento

```
class PseudoPrivacy:

    def __init__(self):
        self._is_private = False

p = PseudoPrivacy()
print(p._is_private)

p._is_private = True
print(p._is_private)
```

```
False
True
```



- A filosofia de pseudo-privacidade de Python pode ser descrita com a seguinte frase escrita em um livro de Pearl “ *You should stay out of the living room because you weren't invited, not because it is defended with a shotgun.*”
- Python não possui elementos privados.
- Você pode apenas sugerir que o elemento em si é privado, seguindo a convenção _. Mas ainda pode ser acessado de fora da Classe.
- Python assemelha-se a perl neste quesito.
- Algumas IDEs e Linters de Python geram warnings ao tentar acessar variáveis protegidas.

Orientação Objeto
Encapsulamento

- O método “Pythonico” de acessar elementos protegidos é utilizando o decorator **property**.

```
# Using @property decorator
class Celsius:
    def __init__(self, temperature=0):
        self._temperature = temperature

    def to_fahrenheit(self):
        return (self._temperature * 1.8) + 32

    @property
    def temperature(self):
        print("Getting value...")
        return self._temperature

    @temperature.setter
    def temperature(self, value):
        print("Setting value...")
        if value < -273.15:
            raise ValueError("Temperature below -273 is not possible")
        self._temperature = value
```

```
# create an object
pessoa = Celsius(37)
pessoa.temperature = 35
print("Temperatura")
print(pessoa.temperature)
print(pessoa.to_fahrenheit())
temp_negative = Celsius(-200)
print(temp_negative.temperature)
```

```
Setting value...
Temperatura
Getting value...
35
95.0
Getting value...
-200
```

Orientação Objeto
Herança

```
class Animal:
    def __init__(self, nome, peso):
        self.nome = nome
        self.peso = peso
    def mostrar(self):
        print("Nome:", self.nome)
        print("Peso:", self.peso)

class Gato(Animal):
    def __init__(self, nome, peso, cor):
        self.cor = cor
        super().__init__(nome, peso)
```

```
gato = Gato("Frajola", 6, "Preto e Branco")
```

```
gato.mostrar()
```

```
Nome: Frajola
Peso: 6
```

- Quando criado um construtor para classe filho, esse construtor irá substituir o construtor da classe pai. Para acessar o construtor da classe pai é necessário utilizar a função **super**;
- Super é utilizado para acessar métodos e o construtor da classe pai;
- Utilizando o método **mostrar** em gato, ele utilizará o da classe pai.

Orientação Objeto
Herança

```
class Animal:
    def __init__(self, nome, peso):
        self.nome = nome
        self.peso = peso
    def mostrar(self):
        print("Nome:", self.nome)
        print("Peso:", self.peso)

class Gato(Animal):
    def __init__(self, nome, peso, cor):
        self.cor = cor
        super().__init__(nome, peso)
    def mostrar(self):
        print("Gato")
        print("Nome:", self.nome)
        print("Peso:", self.peso)
        print("Cor:", self.cor)
```

```
gato = Gato("Frajola", 6, "Preto e Branco")
```

```
gato.mostrar()
```

```
Gato
Nome: Frajola
Peso: 6
Cor: Preto e Branco
```

- Se utilizarmos um método na classe filho com mesmo nome de uma método da classe pai, quando chamado pela classe filho, será chamado o método dela.
- No exemplo ao lado, método mostrar de classe Gato será invocado ao invés do método mostrar da classe Animal.

Orientação Objeto
Herança

```python
class Animal:
    def __init__(self, nome, peso):
        self.nome = nome
        self.peso = peso
    def mostrar(self):
        print("Nome:", self.nome)
        print("Peso:", self.peso)

class Gato(Animal):
    def __init__(self, nome, peso, cor):
        self.cor = cor
        super().__init__(nome, peso)
    def mostrar(self):
        print("Gato")
        super().mostrar()
        print("Cor:", self.cor)
```

```python
gato = Gato("Frajola", 6, "Preto e Branco")
```

```python
gato.mostrar()
```

```
Gato
Nome: Frajola
Peso: 6
Cor: Preto e Branco
```

- Mas caso seja necessário, também é possível chamar o método da classe pai dentro do método da classe filho. Isso evita que o mesmo código seja reescrito.

Orientação Objeto
staticmethod e classmethod

```
class Calcular:
    def __init__(self, base, altura):
        self.base = base
        self.altura = altura
    def calc_area(self):
        area = self.base*self.altura
        print("Area: ",area)
    @staticmethod
    def formula():
        print("A fórmula para calcular:")
        print("Area = base * altura")
```

```
Calcular.formula()
```

```
A fórmula para calcular:
Area = base * altura
```

- Para declarar um método como sendo estático basta utilizar o decorator **@statichmethod** antes da definição da função.
- Da mesma forma utilizar o decorator **@classmethod** para atribuir um método à classe.

Orientação Objeto
Classes Abstratas

```
from abc import ABC, abstractmethod

class Polygon(ABC):
    @abstractmethod
    def sides(self):
        pass

class Triangle(Polygon):
    def sides(self):
        print("I have 3 sides")

class Pentagon(Polygon):
    def sides(self):
        print("I have 5 sides")
```

```
p = Triangle()
p.sides()
p = Pentagon()
p.sides()
```

```
I have 3 sides
I have 5 sides
```

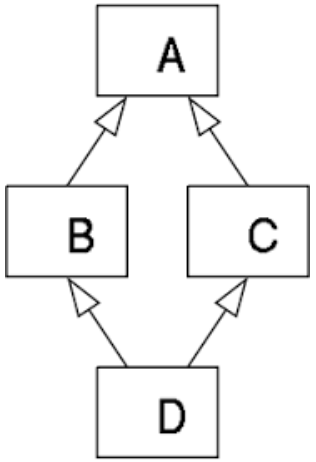
- Classes abstratas em Python utilizam o pacote **ABC**.
- Para definir métodos abstratos de uma classe, é necessário utilizar o decorator **@abstractmethod**.

Orientação Objeto
Exemplo Overloading built-ins

Orientação Objeto
Diamond Problem – Herança Multipla

```python
class A:
    def do_thing(self):
        print('From A')

class B(A):
    def do_thing(self):
        print('From B')

class C(A):
    def do_thing(self):
        print('From C')

class D(B, C):
    pass

d = D()
d.do_thing()

class D(C, B):
    pass

d = D()
d.do_thing()
```

```
From B
From C
```




```python
class A:
    def do_thing(self):
        print('From A')

class B(A):
    pass

class C(A):
    def do_thing(self):
        print('From C')

class D(B, C):
    pass

d = D()
d.do_thing()
```

```
From C
```

- O Diamond Problema em Python, acontece quando duas classes possuem uma sub-classe em comum e ambas possuem a mesma classe base.
- De qual Classe (B ou C) a Classe D vai herdar o método do_thing?
- Python, respeita o Method Resolution Order (MRO) que segue a ordem das classes pai declaradas na classe herdeira (código à esquerda).
- Da mesma forma se há conflito nos nomes dos métodos o primeiro é chamado (código à direita).

Orientação Objeto
Diamond Problem – Herança Multipla

```python
class A:
    def do_thing(self):
        print('From A')

class B(A):
    def do_thing(self):
        print('From B')
        super().do_thing()

class C(A):
    def do_thing(self):
        print('From C')
        super().do_thing()

class D(B,C):
    def do_thing(self):
        print('From D')
        super().do_thing()

d = D()
d.do_thing()
```

```
From D
From B
From C
From A
```

```python
class A:
    def __init__(self):
        print("A.__init__")

class B(A):
    def __init__(self):
        print("B.__init__")
        super().__init__()

class C(A):
    def __init__(self):
        print("C.__init__")
        super().__init__()


class D(B,C):
    def __init__(self):
        print("D.__init__")
        super().__init__()

d = D()
```

```
D.__init__
B.__init__
C.__init__
A.__init__
```

- A Forma Pythonica de resolver este problema é utilizando a função **super.**
- O super() é utilizado entre heranças de classes, ele nos proporciona extender/subscrever métodos de uma super classe (classe pai) para uma sub classe (classe filha)
- O método super é comumente utilizado no método __init__ da classe, no momento em que as instâncias são inicializadas.

NUMPY

Numpy
Introdução

- Utilizada principalmente para cálculos com arrays multidimensionais, possuindo diversas funções e operações para cálculo numérico;
- Amplamente utilizada para cálculos de:
  - Modelos de machine learning;
  - Processamento de imagem e computação gráfica;
  - Tarefas matemáticas.
- Pode ser instalado através do pip ou do Anaconda.
- Link para documentação do numpy:
  - https://numpy.org/doc/stable/user/index.html

Numpy
Nparrays

- Os arrays em numpy são objetos e são do tipo ndarray;
- O array numpy é uma tabela geralmente composta por números, todos do mesmo tipo, indexados por uma tupla de inteiros positivos;
- Ndarrays podem ter 0 ou múltiplas dimensões, e sua dimensões são chamadas de eixos. Por exemplo, coordenadas de um ponto num espaço 2D: [3,2] tem um eixo e dois elementos.

Numpy
Nparrays

- Por convenção, numpy é importado dessa forma, atribuindo o apelido **np**;
- Para criar um nparray, basta atribuir a função np.array([ ]) e passar os elementos como parâmetros;
- Um outro ponto é que nparray é um tipo diferente do array nativo ao Python.

```
import numpy as np
```

```
# Criando um array em numpy
arr = np.array([1, 3, 2, 8, 5])
print(arr)
```

```
[1 3 2 8 5]
```

```
arr_python = [1, 3, 2, 8, 5]
arr = np.array([1, 3, 2, 8, 5])
print("Array do Python:", arr_python, "Tipo:", type(arr_python))
print("Array numpy:", arr, "Tipo:", type(arr))
```

```
Array do Python: [1, 3, 2, 8, 5] Tipo: <class 'list'>
Array numpy: [1 3 2 8 5] Tipo: <class 'numpy.ndarray'>
```

Numpy
Nparrays

```
import numpy as np
arr_1 = np.array([1,2,3,5])
print("Np array:", arr_1)
print("Dimensões:", arr_1.shape)
```

```
Np array: [1 2 3 5]
Dimensões: (4,)
```

- A função **shape** retorna uma tupla que indica quantas dimentões o nparray possui;
- Neste explemo tempo um nparray com apenas uma dimensão que contém 4 elementos;

Numpy
Nparrays multidimensionais

```
arr_2 = np.array([[1,2,3], [3,4,5]])
print("Np array 2 dimensões")
print(arr_2)
```

```
Np array 2 dimensões
[[1 2 3]
 [3 4 5]]
```

```
print(arr_2.shape)
```

```
(2, 3)
```

- Para criar um nparray bidimensional, é semelhate a criar lista de listas em Python, mas utilizando o np.array, conforme o exemplo;
- Utilizando o shape, podemos ver que o array contém duas dimensões com 3 elementos em cada.

Numpy
np.arange

```
print("Utilizando o start e stop")
print(np.arange(1,8))
print("Utilizando o start, stop e step")
print(np.arange(1,8,2))
```

```
Utilizando o start e stop
[1 2 3 4 5 6 7]
Utilizando o start, stop e step
[1 3 5 7]
```

- A função np.arange cria um array de números inteiros ou reais uniformemente distribuídos;
- O formato da função é: **np.arange(start, stop, step)**;
  - **Start**: onde irá começar o intervalo, inclui esse número, parâmetro opcional;
  - **Stop**: até onde irá o intervalo, não inclui esse número;
  - **Step**: espaçamento entre os valore, parâmetro opcional.

Numpy
Np.full

```
print("2 dimensões com 2 elementos")
print("Elementos com valores 10")
print(np.full((2, 2), 10))
print("2 dimensões com 3 elementos")
print("Elementos com valores 42")
print(np.full((2, 3), 42))
```

```
2 dimensões com 2 elementos
Elementos com valores 10
[[10 10]
 [10 10]]
2 dimensões com 3 elementos
Elementos com valores 42
[[42 42 42]
 [42 42 42]]
```

- Retorna um novo nparray da forma e tipo especificado;
- O formato básico da função é: **np.full(shape, fill_value)**:
  - **Shape**: é o formato do array, semelhante ao retorno da função shape;
  - **Fill_value**: é o valor que será usado para preencher o array, podendo ser um valor escalar ou um do tipo array.

Numpy
Indexação utilizando inteiros

```
arr_1 = np.array([1,2,3,5])
print("1 dimensão")
print(arr_1)
print("Terceiro elemento")
print(arr_1[2])
```

```
1 dimensão
[1 2 3 5]
Terceiro elemento
3
```

- A indexação pode ser feita através de inteiros que representam o valor do indicie;
- Para um nparray com 1 dimensão, passamos apenas o valor do index, semelhante a uma lista. Mas para cada dimensão adicional, é necessário adicionar mais um valor para especificar o elemento.

```
arr_2 = np.array([[1,2,3], [3,4,5]])

print("2 dimensões")
print(arr_2)
print("Segundo elemento da primeiro elemento")
print(arr_2[0, 1])
```

```
2 dimensões
[[1 2 3]
 [3 4 5]]
Segundo elemento da primeiro elemento
2
```

Numpy
Indexação utilizando inteiros

```
matriz = np.array([[1,2,3], [4,5,6],
                   [7,8,9], [10,11,12]])
indicie = np.array([0,2,0,1])
print("Valores selecionados")
print(matriz[np.arange(4),indicie])
print("Valores selecionados mais escalar")
print(matriz[np.arange(4),indicie]+10)
```

```
Valores selecionados
[ 1  6  7 11]
Valores selecionados mais escalar
[11 16 17 21]
```

- Essa forma é útil para alterar um elemento de cada linha de uma matriz de valores através de um array de índices;
- Com o **np.arange** criamos um array para acessar cada linha da matriz e podemos alterar os valores de **indice** para acessar os valores que quisermos;
- Podemos utilizar escalares para alterar o valor dos elementos.

Numpy
Indexação utilizando booleano

```
array = np.array([[1,2], [3,4], [5,6]])
bool_index = (array>3)
print("Matriz booleana:")
print(bool_index)
print("Elementos filtrados:")
print(array[bool_index])
# Outro modo de utilizar
print(array[array>3])
```

```
Matriz booleana:
[[False False]
 [False  True]
 [ True  True]]
Elementos filtrados:
[4 5 6]
[4 5 6]
```

- É possível também utilizar indexação com arrays booleanos, que normalmente utilizam uma condição;
- Pode ser feito tanto utilizando um array ou matriz de booleanos como também passando a condicional direto no array.

Numpy
Exercício 1

- Crie um array com os número de 0 a 15. Depois identifique todos os números impares desse array e salve esses números em um outro array.
  - Tempo estimado: 10 minutos

Numpy
Exercício 1 - Resposta

```
array = np.arange(16)
impar = (array%2!=0)
array_impar = array[impar]
print("Array de 0 a 15")
print(array)
print("Array booleano")
print(impar)
print("Array com os impares")
print(array_impar)
```

```
Array de 0 a 15
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15]
Array booleano
[False  True False  True False  True False  True False  True False  True
 False  True False  True]
Array com os impares
[ 1  3  5  7  9 11 13 15]
```

Numpy
Funções matemáticas

```
x = np.array([[1,2], [3,4]])
y = np.array([[5,6], [7,8]])

print("Soma")
print(x+y)
print(np.add(x,y))

print("Subtração")
print(x-y)
print(np.subtract(x,y))

print("Multiplicação")
print(x*y)
print(np.multiply(x,y))

print("Divisão")
print(x/y)
print(np.divide(x,y))
```

```
Soma
[[ 6  8]
 [10 12]]
[[ 6  8]
 [10 12]]
Subtração
[[-4 -4]
 [-4 -4]]
[[-4 -4]
 [-4 -4]]
Multiplicação
[[ 5 12]
 [21 32]]
[[ 5 12]
 [21 32]]
Divisão
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
```

- Com arrays numpy é possível realizar função matemáticas de forma direta;
- Importante notar que essas funções são elemento a elemento, então a multiplicação resultante não é uma multiplicação matricial.

Numpy
Matemática com matrizes

```
mtz = np.array([[1,2,3],[4,5,6]])

print("Soma todos os valores:", np.sum(mtz))
print("Soma as colunas:", np.sum(mtz, axis=0))
print("Soma as linhas:", np.sum(mtz, axis=1))
print("Transpondo a matriz:")
print(mtz.T)
```

```
Soma todos os valores: 21
Soma as colunas: [5 7 9]
Soma as linhas: [ 6 15]
Transpondo a matriz:
[[1 4]
 [2 5]
 [3 6]]
```

- Algumas funções são úteis para lidar com matrizes, como a função de soma **np.sum**;
- O modo básico dessa função é: **np.sum(array, axis)**:
  - **array**: np.array que será feito a soma;
  - **axis**,: parâmetro opcional. Indica qual eixo será utilizado para a soma. Se igual a 0, será feita a soma das colunas. Igual a 1, será feita a soma das linhas. Se não for utilizado, será feito a soma de todos os elementos.

Numpy
Performance

```python
import time
import numpy as np


a = list(range(int(1e6)))

start = time.time()
x = a.copy()
for i, val in enumerate(x):
    if val%2 == 0:
        x[i] = val+1
print(f"Tempo de duração for : {time.time() - start:.4f}s")
print(x[:10])


start = time.time()
x = [val+1 if val%2 == 0 else val for val in a]
print(f"Tempo de duração listComp : {time.time() - start:.4f}s")
print(x[:10])

start = time.time()
na = np.array(a)
x = np.array(np.where(na%2 == 0, na+1, na))
print(f"Tempo de duração numpy : {time.time() - start:.4f}s")
print(list(x[:10]))
```

- Retomando o exemplo de comparação de performance.
- Podemos ver o ganho de tempo do **numpy** em relação à um for loop e a uma compreensão de lista.

```
Tempo de duração for : 0.1446s
[1, 1, 3, 3, 5, 5, 7, 7, 9, 9]
Tempo de duração listComp : 0.0998s
[1, 1, 3, 3, 5, 5, 7, 7, 9, 9]
Tempo de duração numpy : 0.0868s
[1, 1, 3, 3, 5, 5, 7, 7, 9, 9]
```

Numpy
Slicing

```
array = np.array([1,2,3,4,5,6])
print(array[1:3])
print(array[:5])
print(array[2:])
print(array[:])
```

```
[2, 3]
[1, 2, 3, 4, 5]
[3, 4, 5, 6]
[1, 2, 3, 4, 5, 6]
```

```
array_md = np.array([[1,2,3], [4,5,6],
                     [7,8,9], [10,11,12]])
print(array_md)
print(array_md[:2, :2])
```

```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
[[1 2]
 [4 5]]
```

- Slicing é uma forma de dividir um array de um indicie até outro indice;
- Diferente da indexação com inteiros, o slicing retorna os elementos dentro do intervalo;
- Para isso basta adicionar **:** e passar o intervalo. E para cada dimensão acrescentar a virgula entre as dimensões e o intervalo.

Numpy
Slicing

```
s = slice(2, 10)

print(list(range(20))[s])
```

```
[2, 3, 4, 5, 6, 7, 8, 9]
```

- Slicing é uma forma de dividir um array de um indicie até outro indice;
- Diferente da indexação com inteiros, o slicing retorna os elementos dentro do intervalo;
- Para isso basta adicionar **:** e passar o intervalo. E para cada dimensão acrescentar a virgula entre as dimensões e o intervalo.

Numpy
Função where

```
a = np.array([1,2,3,4,5,6,7,8])
print(np.where(a < 5, 1 ,0))
print(np.where(a < 5, 2-a ,a-2))
```

```
[1 1 1 1 0 0 0 0]
[ 1  0 -1 -2  3  4  5  6]
```

- A função where é utilizada para atribuir uma condional a um array;
- O modo da função é: **np.where(condition, x, y)**:
  - **condition**: a condição a ser atendida;
  - **x**: o que será retornado caso a condição seja verdadeira;
  - **y**: o que será retornado caso a condição seja falsa.

Numpy
Função newaxis

```
arr = np.arange(4)
print("Dimensão inicial:", arr.shape)
print("Vetor inicial:")
print(arr)
```

```
Dimensão inicial: (4,)
Vetor inicial:
[0 1 2 3]
```

```
row_vec = arr[np.newaxis, :]
print("Nova dimensão:", row_vec.shape)
print("Nova matriz:")
print(row_vec)
```

```
Nova dimensão: (1, 4)
Nova matriz:
[[0 1 2 3]]
```

```
col_vec = arr[:, np.newaxis]
print("Nova dimensão:", col_vec.shape)
print("Nova matriz:")
print(col_vec)
```

```
Nova dimensão: (4, 1)
Nova matriz:
[[0]
 [1]
 [2]
 [3]]
```

- De forma simples, o objetivo dessa função é adicionar mais uma dimensão ao np.array;
- A nova dimensão pode ser adicionada como uma nova linha ou como coluna;
- Muito utilizado para inferência com modelos de Deep Learning

Numpy
Função reshape

```
arr = np.array([1,2,3,4,5,6,7,8,9,10,11,12])
new_arr1 = arr.reshape(2,6)
new_arr2 = arr.reshape(3,4)
new_arr3 = arr.reshape(6,2)
print("Array com shape (2,6)")
print(new_arr)
print("Array com shape (3,4)")
print(new_arr2)
print("Array com shape (6,2)")
print(new_arr3)
```

```
Array com shape (2,6)
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]]
Array com shape (3,4)
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
Array com shape (6,2)
[[ 1  2]
 [ 3  4]
 [ 5  6]
 [ 7  8]
 [ 9 10]
 [11 12]]
```

- A função reshape é usada quando é necessário mudar a forma de um nparray;
- Ao utilizá-la podemos adicionar ou remover dimensões, ou alterar o número de elementos em cada dimensão.

Numpy
Função transpose

```
x = np.array([[[1,2],[3,4],[5,6],[7,8]]])
new = np.transpose(x, (0,2,1))
print("Antes do transpose:", x.shape)
print(x)
print("Depois do transpose:", new.shape)
print(new)
```

```
Antes do transpose: (1, 4, 2)
[[[1 2]
  [3 4]
  [5 6]
  [7 8]]]
Depois do transpose: (1, 2, 4)
[[[1 3 5 7]
  [2 4 6 8]]]
```

- Utilizada para reverter ou trocar os eixos de uma matriz;
- O formato da função é: **np.transpose(a, *axes=None*)**:
  - **a**: o array a ser transposto;
  - **axes**: parâmetro opcional, quando especificado, diz os eixos de a que deverão ser trocados.

Numpy
Função flatten

- Retorna a cópia do nparray transformada em apenas uma dimensão;

```
x_2 = np.array([[1,2,3], [4,5,6]])
x_3 = np.array([[[1,2],[3,4],[5,6],[7,8]]])
print(x_2.shape)
print(x_2.flatten())
print(x_2.flatten().shape)
print(x_3.shape)
print(x_3.flatten())
print(x_3.flatten().shape)
```

```
(2, 3)
[1 2 3 4 5 6]
(6,)
(1, 4, 2)
[1 2 3 4 5 6 7 8]
(8,)
```

Numpy
Processamento de imagens

- Numpy é utilizado na área de **processamento** e **manipulação de imagens** pois permite a edição das características das imagens carregadas através de operações com **matrizes numpy**.

- Alguns exemplos de bibliotecas para carregamento (e manipulação) das imagens são: **OpenCV**, matplotlib, skimage, PIL.

- Abordaremos OpenCV a fundo em um módulo posterior.

Numpy
Demonstrando carregamento

- OpenCV

```
# OpenCV
import cv2
img_cv2 = cv2.imread('art.jpg')
print(type(img_cv2))
print(img_cv2.shape)
```

```
<class 'numpy.ndarray'>
(531, 832, 3)
```

Numpy
Demonstrando carregamento

- Matplotlib

```
# Matplotlib
import matplotlib.pyplot as plt
img_mp= plt.imread('art.jpg')
print(type(img_mp))
print(img_mp.shape)
```

```
<class 'numpy.ndarray'>
(531, 832, 3)
```

Numpy
Demonstrando carregamento

- Skimage

```
# Skimage
from skimage import io
img_sk = io.imread('art.jpg')
print(type(img_sk))
print(img_sk.shape)
```

```
<class 'numpy.ndarray'>
(531, 832, 3)
```

Numpy
Demonstrando carregamento

- PIL

```
# PIL
from PIL import Image
img_pil = Image.open('art.jpg')

print(type(img_pil))
print(type(numpy.array(img_pil)))
print(numpy.array(img_pil).shape)
```

```
<class 'PIL.JpegImagePlugin.JpegImageFile'>
<class 'numpy.ndarray'>
(531, 832, 3)
```

Numpy
Diferença no carregamento de canais

```
plt.figure(figsize=(20,10))
plt.subplot(221)
plt.title('matplotlib read')  # RGB
plt.imshow(img_mp)
plt.subplot(222)
plt.title('opencv read')  # BGR
plt.imshow(img_cv2)
plt.subplot(223)
plt.title('PIL read')  # RGB
plt.imshow(img_pil)
plt.subplot(224)
plt.title('skimage read')  # RGB
plt.imshow(img_sk)
plt.show()
```

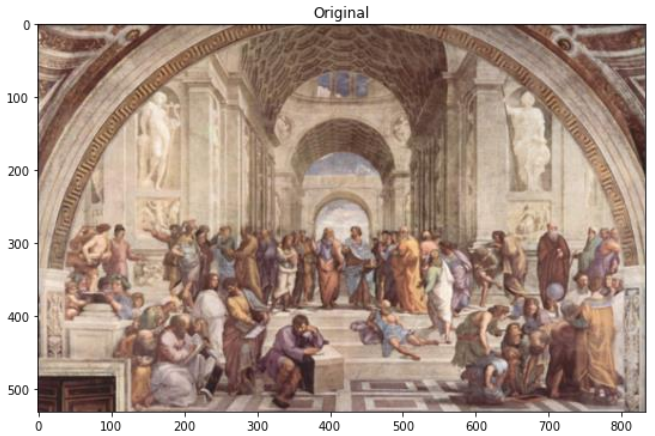
Numpy
Comparação de perfomance

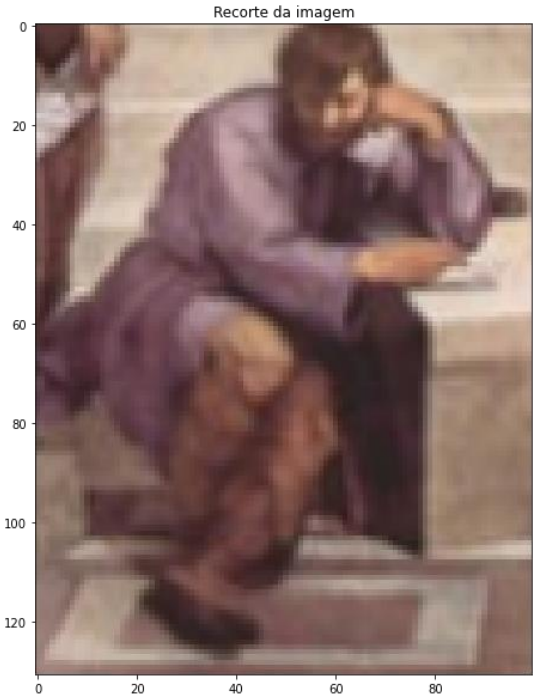
- Comparação de performance para leitura e carregamento das imagens em cada biblioteca.
- Para biblioteca PIL consideramos na conta o tempo de carregamento + conversão para numpy array.

```
plt.imread             : 0.0379967689514160 16
cv2.imread             : 0.0340015888214111 3
Image.open(+np.array) : 0.0209829807281494 14
io.imread              : 0.0210006237030029 3
```

Numpy
Recorte de imagem

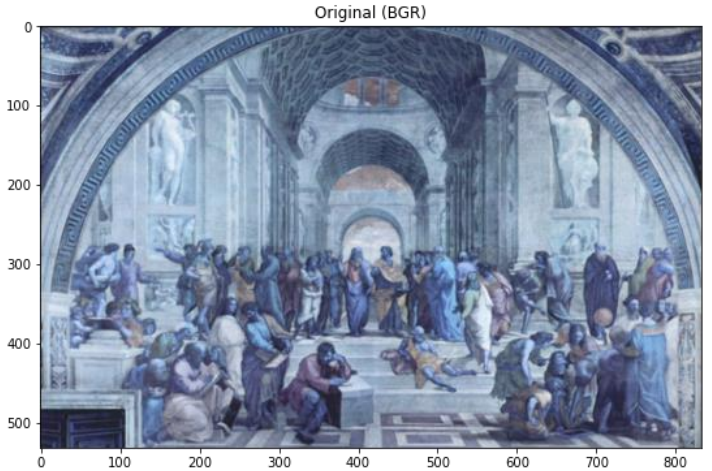
```
img_slc = img_mp [400:, 300:400]

plt.figure(figsize=(20,10))
plt.subplot(121)
plt.imshow(img_mp)
plt.title('Original')

plt.subplot(122)
plt.imshow(img_slc)
plt.title('Recorte da imagem');
```

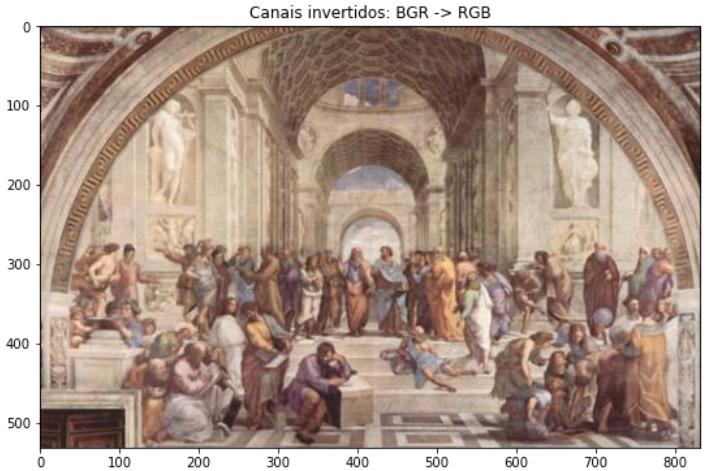




Numpy
Troca de canais

```
img_cv2_inv = img_cv2[:,:,::-1]

plt.figure(figsize=(20,10))
plt.subplot(121)
plt.imshow(img_cv2)
plt.title('Original (BGR)')

plt.subplot(122)
plt.imshow(img_cv2_inv)
plt.title('Canais invertidos: BGR -> RGB');
```





Processamento de imagens
Expansão de dimensão e criação de lotes

- Alguns problemas necessitam que os arrays estejam no formato 4D.

```
img_cv2_exp = numpy.expand_dims(img_cv2, axis = 0)

img_cv2_exp.shape
```

```
(1, 531, 832, 3)
```

```
img_pil_exp = numpy.expand_dims(img_pil, axis = 0)

img_pil_exp.shape
```

```
(1, 531, 832, 3)
```

Processamento de imagens
Expansão de dimensão e criação de lotes

- Após expandir a dimensão (3D para 4D) é possível criar um lote de imagens concatenando os arrays 4D.
- O formato adquirido será: (numero de imagens, altura, largura, canais).
- Através do índice da primeira dimensão é possível alternar entre as imagens.

```
img_batch = numpy.concatenate([img_cv2_exp, img_pil_exp])

img_batch.shape
```

```
(2, 531, 832, 3)
```

```
# Primeira imagem
print(img_batch[0].shape)
plt.imshow(img_batch[0]);
```

```
(531, 832, 3)
```

PYINSTALLER

PyInstaller
Introdução

- Pretendo distribuir minha aplicação Python para outras máquinas.
  - Preciso Instalar o Python em todas máquinas?
  - Preciso compartilhar o ambiente virtual?
  - A Máquina não tem acesso a internet, como eu faço para baixar os módulos?

PyInstaller
Características

PyInstaller

#!/bin/env python
import os
import sys

- Pyinstaller
  - É uma forma de transformar sua aplicação em um executável stand-alone.
  - É multi-plataforma.
  - Tem suporte para PyQT, Django e matplotlib.
  - Suporta integração com bibliotecas binárias
  - Single-file, Single-Folder ou customizada.

PyInstaller
Criando executáveis

- É necessário que sua aplicação tenha um arquivo main.py ou run.py, que irá executar toda a aplicação
- É Recomendado que Pyinstaller seja usado dentro de um virtualenv. Pyinstaller verifica todos os módulos utilizados pela sua aplicação, faz uma cópia destes módulos incluindo o interpretador Python e coloca dentro de um único diretório/arquivo.
- Para indicar bibliotecas específicas ou binários um arquivo "*additional-hooks*" precisará ser criado.
- "pyinstaller myscript.py" ou "pyinstaller –onefile –windowed –noconsole myscript.py"

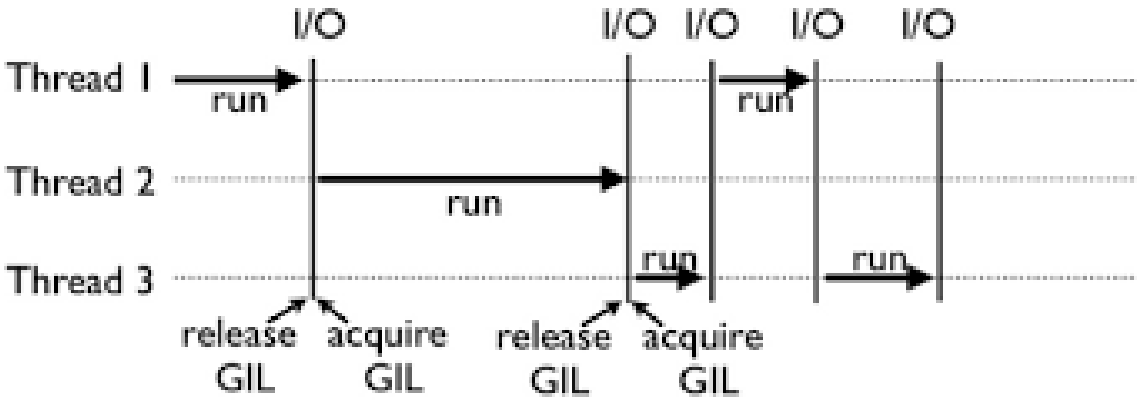
PyInstaller
Exemplo

BOSCH

MULTITHREADS VS MULTIPROCESS

Introdução
Python GIL

- Global Interpreter Lock (**GIL**) é um ***mutex*** (ou ***lock***) que permite que apenas uma ***thread*** tome conta do interpretador Python. Isso significa que somente uma ***thread*** pode estar em um estado de execução em qualquer ponto do tempo.



Fonte: https://github.com/wrtinho/GIL-Python

Introdução
Estrutura de Dados

```
x = []
sys.getrefcount(x)
```

```
2
```

- Quando é declarada uma variável em Python, é criado um objeto na memória *heap*. *Ex:* x = 5, onde a variável x conterá uma referência para o objeto 5.
- Caso x = None, o interpretador python irá verificar que o x terá 0 referências e será setado como um "*Dead Object*" pelo *Garbage Collector* do Python (*Reference Counting*).
- Essa forma de gerenciamento de memória precisa de proteção para um fenômeno chamado *'race conditions*', quando duas threads aumentam ou diminuem seu valor simultaneamente.
- Se isso acontecer, poderá causar problemas de memória, onde poderá nunca excluir um objeto ou até liberação incorreta do mesmo enquanto ainda existe referência.
- Devido à isso é utilizado o GIL. Transformando qualquer Código Python em *single-thread*

Introdução
Multithread vs MultiProcess

```python
# single_threaded.py
import time
from threading import Thread

COUNT = 50000000

def countdown(n):
    while n>0:
        n -= 1

start = time.time()
countdown(COUNT)
end = time.time()

print('Time taken in seconds -', end - start)
```

```
$ python single_threaded.py
Time taken in seconds - 6.20024037361145
```

```python
# multi_threaded.py
import time
from threading import Thread

COUNT = 50000000

def countdown(n):
    while n>0:
        n -= 1

t1 = Thread(target=countdown, args=(COUNT//2,))
t2 = Thread(target=countdown, args=(COUNT//2,))

start = time.time()
t1.start()
t2.start()
t1.join()
t2.join()
end = time.time()

print('Time taken in seconds -', end - start)
```

```
$ python multi_threaded.py
Time taken in seconds - 6.924342632293701
```

```python
from multiprocessing import Pool
import time

COUNT = 50000000
def countdown(n):
    while n>0:
        n -= 1

if __name__ == '__main__':
    pool = Pool(processes=2)
    start = time.time()
    r1 = pool.apply_async(countdown, [COUNT//2])
    r2 = pool.apply_async(countdown, [COUNT//2])
    pool.close()
    pool.join()
    end = time.time()
    print('Time taken in seconds -', end - start)
```

```
$ python multiprocess.py
Time taken in seconds - 4.060242414474487
```

Introdução
Multi-Thread Example

EXERCÍCIO PYTHON

BOSCH