

Semana 3

Objetivo: Introdução à Linguagem Python e conhecendo as bibliotecas Numpy e Matplotlib.

Conteúdo: Python:

1. Configuração do Ambiente
2. Introdução à Linguagem Python
3. Contêineres em Python
4. Funções em Python
5. Classes em Python
6. Bibliotecas
 - a. Numpy
 - b. Matplotlib

Desafio: Realizar o curso completo e responder todos os exercícios propostos neste material

Material adaptado de Prof. Felipe Meneguzzi e Henry Cagnini

Configuração do Ambiente

1 - Instalando o Interpretador Python

Faça o download do python 3.x para o seu sistema operacional. A distribuição Anaconda é a mais recomendada, pois possui diversas funcionalidades e já inclui os principais pacotes utilizados por cientistas de dados. Link para download: <https://www.anaconda.com/distribution/>

Após fazer o download, siga as instruções para instalação neste link: <https://docs.anaconda.com/anaconda/install/>

Você pode testar a instalação abrindo um terminal e executando o seguinte comando:

```
$ python --version
```

Se a instalação foi bem-sucedida, você deve ver a seguinte mensagem (Dependendo da versão que foi instalada):

```
Python 3.6.7 :: Anaconda, Inc.
```

A partir deste ponto, o interpretador está instalado e pronto para ser utilizado.

3 - Instalando Pacotes

Novos pacotes podem ser instalados facilmente no ambiente que está ativo no momento utilizando o comando `$ pip`. Para instalar um pacote basta executar `$ pip install <nomeDoPacote>`. Ex.: `pip install matplotlib`. Para listar todos os pacotes instalados, basta executar `$ pip list`.

4 - Utilizando Jupyter Notebooks

Inicie o jupyter executando o seguinte comando: `$ jupyter notebook`. Após executar este comando, o jupyter deve abrir no navegador padrão, então é só seguir as instruções para criar e editar notebooks.

Se você tem notebooks salvos que deseja abrir, basta navegar para o diretório onde estão os notebooks e executar o jupyter a partir deste diretório.



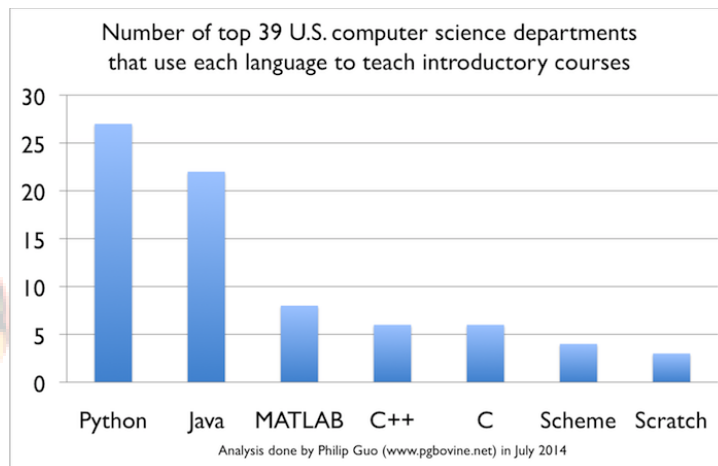
Introdução

Por quê Python?

- Rápida para prototipação (Interpretada)
- Grande comunidade de usuários
- Open Source
- Linguagem de propósito geral

Python: popularidade

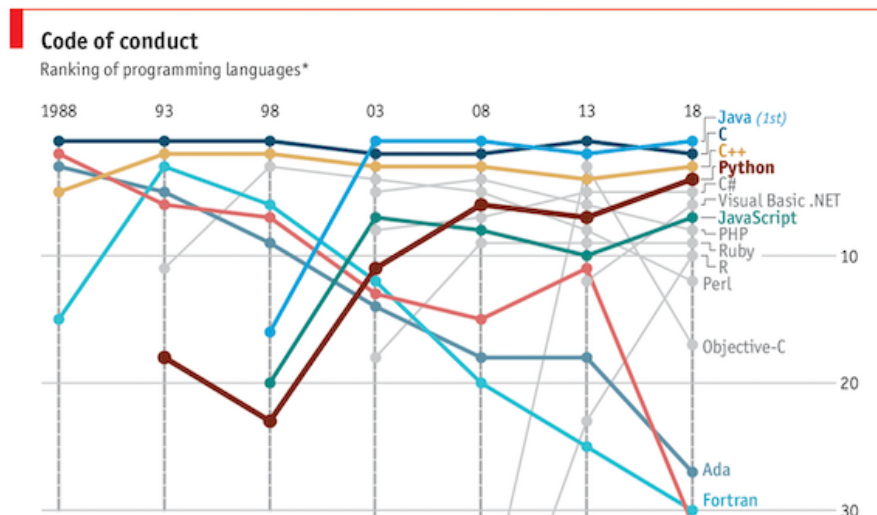
Como linguagem de ensino, Python é a linguagem mais utilizada atualmente.



Daily chart

Python is becoming the world's most popular coding language

But its rivals are unlikely to disappear



O que é Python?

- Linguagem de programação interpretada
- Multiparadigma
- Foco na legibilidade e (re)usabilidade
 - Excelente para prototipagem
- Possui diversas bibliotecas prontas para as mais diversas tarefas

Diferenciais do Python

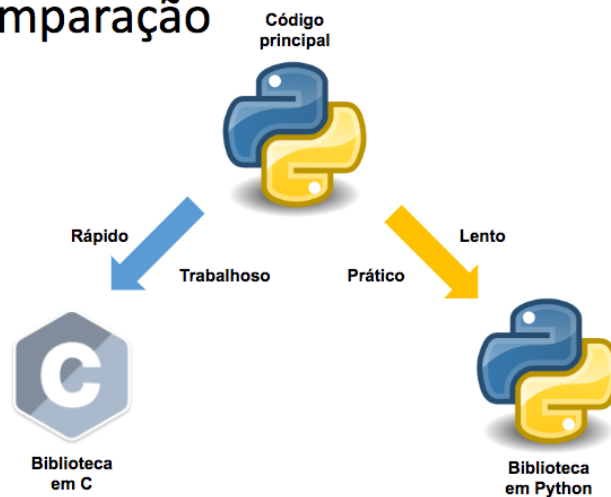
- Linguagem interpretada com alocação dinâmica de espaço
- Sem tipos primitivos
- todas as variáveis são objetos
- Todas as operações com overhead de verificações de tipos

Como mitigar ineficiência do Python?

Programando partes críticas da aplicação em C

- O nome “completo” da distribuição Python mais popular é CPython
 - Outras distribuições: <https://www.python.org/download/alternatives/>
- Isso porque é possível implementar bibliotecas inteiras em C e chamar funções a partir de uma interface Python

Comparação



Frameworks de Deep Learning



Executando Código Python

- Utilizando o terminal executando um programa python (arquivo texto): \$ python hello_world.py
- Utilizando um console interativo: \$ python e \$ ipython
- **Utilizando um Jupyter notebook**

Jupyter Notebooks

- Excelente plataforma para Ensino
- Permite a execução de código por células
- Permite a execução de código **não sequencial**

```
In [1]: print('Este código está sendo executado no jupyter!!!!!!!')
```

Este código está sendo executado no jupyter!!!!!!!

Python: Conceitos Básicos

Variáveis, tipos de dados

- Python manipula objetos
- Cada objeto possui um tipo
 - Tipos definem que operações podem ser realizadas em cada objeto
 - Tipos podem ser escalares ou não-escalares
 - Objetos escalares são indivisíveis
 - Objetos não-escalares possuem estrutura interna (e.g. strings)
 - Python possui quatro tipos de objetos escalares:
 - int representam inteiros
 - float representam números reais em ponto flutuante
 - bool representam valores lógicos booleanos True e False
 - NoneType é um tipo com valor único, denotando None representando a ausência de quaisquer outros valores

- **Variáveis associam nomes a objetos**
 - O operador = associa o valor de uma expressão a uma variável
 - Expressões sempre resultam em um objeto de algum tipo

```
In [2]: inteiro = 10
        decimal = 1.3
        booleano = True
        nulo = None

        print('inteiro: {}, decimal: {}, booleano: {}, nulo: {}'.format(inteiro, decimal, booleano, nulo))

inteiro: 10, decimal: 1.3, booleano: True, nulo: None
```

Conversões de tipos

- Tipos básicos de python possuem uma variedade de operações de conversão de tipo:
 - `int(p)` converterá p para inteiro
 - Possível converter strings str e números em ponto flutuante float
 - `float(p)` converterá p para ponto flutuante
 - Possível converter strings str e números inteiros int
 - `str(p)` converterá p para uma representação em string
 - Possível converter int, float, list, tuple, dict
 - `list(p)` converterá p para uma lista
 - Possível converter str, tuple, dict
 - `tuple(p)` converterá p para uma tupla
 - Possível converter str, list

```
In [3]: inteiro = int('10') + 1
        string = str(10)
        print(inteiro, string)

11 10
```

Operadores matemáticos e comparativos

Escalares em Python possuem um conjunto básico de operadores.

- Os tipos int e float possuem os seguintes operadores matemáticos:
 - `i+j` representa a adição de i e j
 - `i-j` representa a subtração de i e j
 - `i*j` representa a multiplicação de i e j,
 - `i**j` representa i elevado a potência j, para estas quatro operações:

- se ambos i e j forem do tipo int o resultado será do tipo int;
- se qualquer um deles for do tipo float, o resultado também será do tipo float
- $i//j$ representa a divisão *inteira* de i e j (então o resultado será sempre int)
- i/j representa a divisão *em ponto flutuante* de i e j (isto, em Python 3, então o resultado será sempre float)
- $i\%j$ representa o *resto da divisão inteira* de i e j (então o resultado será sempre int)
- De forma similar, estes tipos possuem os operadores comparativos:
 - $==$ (igual);
 - $!=$ (diferente)
 - $>$ (maior que)
 - $>=$ (maior ou igual a)
 - $<$ (menor que); e
 - $<=$ (menor ou igual a).
- Objetos bool possuem os seguintes operadores lógicos
 - a and b conjunção;
 - a or b disjunção;
 - not a negação.

Importação de módulos

- Conjuntos de programas e classes em python são organizados em arquivos individuais ou módulos
- Módulos organizam uma coleção logicamente relacionada de programas distribuídos em um ou mais arquivos
- Tipicamente um arquivo individual .py representa um módulo que pode ser importado
 - Por exemplo considere o arquivo circle.py com um conjunto de funções
 - Podemos acessar as funções de circle.py a partir de outro arquivo usando o comando `import circle`
- Módulos nos permite acessar diversas funções *built-in* da linguagem, por exemplo no módulo math (usando `import math`, temos acesso a diversas funções matemáticas:
 - `math.sin(x)` retorna o seno de x
 - `math.sqrt(x)` retorna a raiz quadrada de x

```
In [4]: import math  
        print(math.sqrt(81))
```

9.0

Ramificações, laços e iterações

- Condicionais em Python utilizam as palavras reservadas if, else, e elif:

if Boolean expression:

 block of code

elif Boolean expression:

 block of code

else:

 block of code

Por exemplo:

if x > 0:

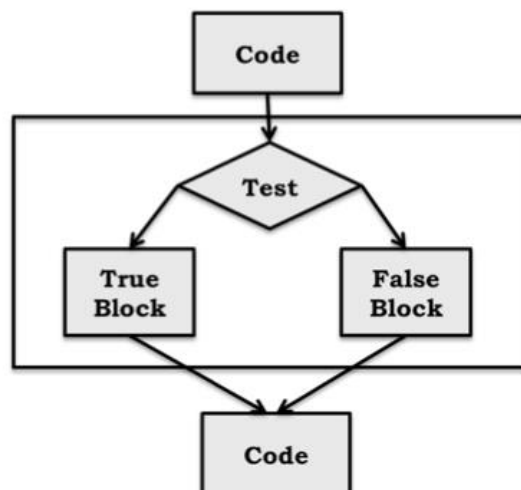
 return x

elif x < 0:

 return -x

else:

 return None



- Note que Python utiliza indentação de forma semanticamente significativa
 - Cada bloco de programa está em um nível de indentação diferente
- Laços em python utilizam a palavra reservada while

while Boolean expression:

block of code

Por exemplo:

```
i = 0
while i < 10:
    print(i)
    i = i + 1
```

Iterações (em tipos que suportam iteração) utilizam a palavra reservada for

```
for variable in sequence:
    code block

Por exemplo
for i in range(0,10):
    print(i)
```

Strings, Entrada e Saída

- Objetos do tipo str (para strings) são declarados tanto com aspas simples 'abs' quanto com aspas duplas "abs"
- Possuem diversos operadores:
 - Concatenação de strings "abc"+"def" resulta na string abcdef
 - Replicação um certo número de vezes 3*"a" resulta em aaa
 - Indexação pode ser usada para extrair caracteres específicos "abc"[0] resultará no caracter 'a'
 - Strings são constantes e imutáveis
- Python 3 possui apenas um comando de entrada: input
 - Único parâmetro é uma string texto a ser apresentada ao usuário:
- Entretanto diversas variações do comando de saída print:

```
>>> print("Algum texto")
>>> print("Texto formatado com número %d"%12) imprime "Texto formatado com número 12"
>>> print("Algum texto",end="") imprime sem quebra de linha
>>> print("Texto formatado co o número {0:f}".format(3.12))
>>> name = input('Enter your name: ')
```

- Enter your name: George Washington

```
>>> print('Are you really', name, '?')
```

Are you really George Washington ?

```
In [*]: name = input('Enter your name: ')
        print('Are you really', name, '?')
```

Enter your name:

```
In [6]: name = input('Enter your name: ')
        print('Are you really', name, '?')
```

Enter your name: José da Silva

Are you really José da Silva ?

Exceções

- Exceções indicam uma condição fora do comum ocorrendo em um programa
- Permitem tratamento de erros e eventualidades de forma separada à saída regular de uma função
- Utilizadas de diversas formas nas bibliotecas Python
- Tratamento de Exceções:
 - Quando ocorre uma exceção, um programa termina, retornando ao programa chamante
 - Caso a exceção não seja capturada, ela pode ser propagada até o programa principal (e terminar o programa inteiro) No exemplo:

```
successFailureRatio = numSuccesses/float(numFailures)
print ('The success/failure ratio is', successFailureRatio)
print ('Now here')
```

Caso o número de falhas numFailures for 0, teremos uma exceção ZeroDivisionError e nenhum print ocorrerá tratando a exceção, teremos o código

```
try:
    successFailureRatio = numSuccesses/float(numFailures)
    print('The success/failure ratio is', successFailureRatio)
except ZeroDivisionError:
    print('No failures so the success/failure ratio is undefined.')
print 'Now here'
```

```
In [7]: string = 'abc'
        print(string[0])
        string[0] = 'b'
```

```
a
```

```
-----
TypeError                                Traceback (most recent call last)
<ipython-input-7-a9d1a2366b96> in <module>
      1 string = 'abc'
      2 print(string[0])
----> 3 string[0] = 'b'

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

Exercícios – Parte 1

Exercício 1

Escreva um código Python que lê do teclado o nome e a idade de um usuário e imprime o ano em que o usuário completará 100 anos.

Dica: você pode ler strings digitadas no teclado utilizando a função builtin `input('mensagem')`. Lembre-se de converter números para seu respectivo tipo (`int` ou `float`) antes fazer operações aritméticas.

Exercício 2

Escreva um código Python que lê do teclado um número digitado pelo usuário e imprime se ele par ou ímpar.

Exercício 3

Escreva um código Python que imprime os números pares de 0 até 20 (incluso).

Dica: olhe a documentação da função `range()`. Mais informações no [link](#)

Contêineres em Python

Listas

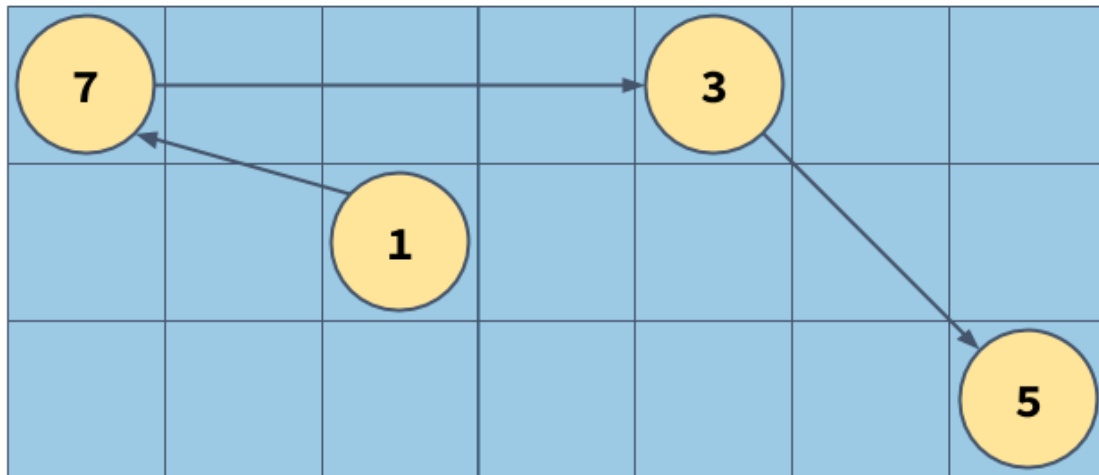
O que são listas?

- Listas são sequências ordenadas de valores onde cada valor é identificado por um índice
- Uma lista é denotada por colchetes, `[]`
- Listas contém elementos:
 - normalmente homogênea (i.e. uma lista de inteiros)
 - podem conter elementos misturados (incomum)
- Listas são mutáveis (permitem inserção e remoção de elementos)
- É possível aplicar slicing sobre listas

- Listas em Python são implementadas como listas encadeadas

Listas

- No console: [1, 7, 3, 5]
 - No hardware:



Quando usar listas em Python?

- Para escopos que duram **pouco tempo**
- Quando o tamanho da lista **será alterado**
- Quando o tamanho da lista é **pequeno**

Listas: índices e ordenação

```
In [8]: a_list = []
L = [2, 'a', 4, [1,2]]
print(len(L))    #avalia para 4
print(L[0])      #avalia para 2
print(L[2]+1)    #avalia para 5
print(L[3])      #avalia para [1,2], outra lista!
i=2 #
print(L[i-1])    #avalia para 'a' já que L[1]='a' acima

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

Listas: append e extend

- Diferença entre append e extend
 - Quando se faz append se acrescenta um objeto lista em outra lista
 - Quando se faz extend acrescenta-se uma cópia dos elementos de uma lista na outra lista

```
In [9]: a = [1, 2, 3]
        b = [4, 5, 6]
        a.extend(b)
        print(a)
        a.append(7)
        print(a)

[1, 2, 3, 4, 5, 6]
[1, 2, 3, 4, 5, 6, 7]
```

Listas: iteração

- Por exemplo, computar a soma dos elementos de uma lista
- Padrão comum, iterar sobre os elementos de uma lista:

```
total = 0
for i in range(len(L)):
    total += L[i]
print total

O código acima é equivalente a:

total = 0
for i in L:
    total += i
print total
```

- Note que:
 - os elementos são indexados de 0 até $\text{len}(L)-1$
 - $\text{range}(n)$ vai de 0 a $n-1$

Listas: indexação

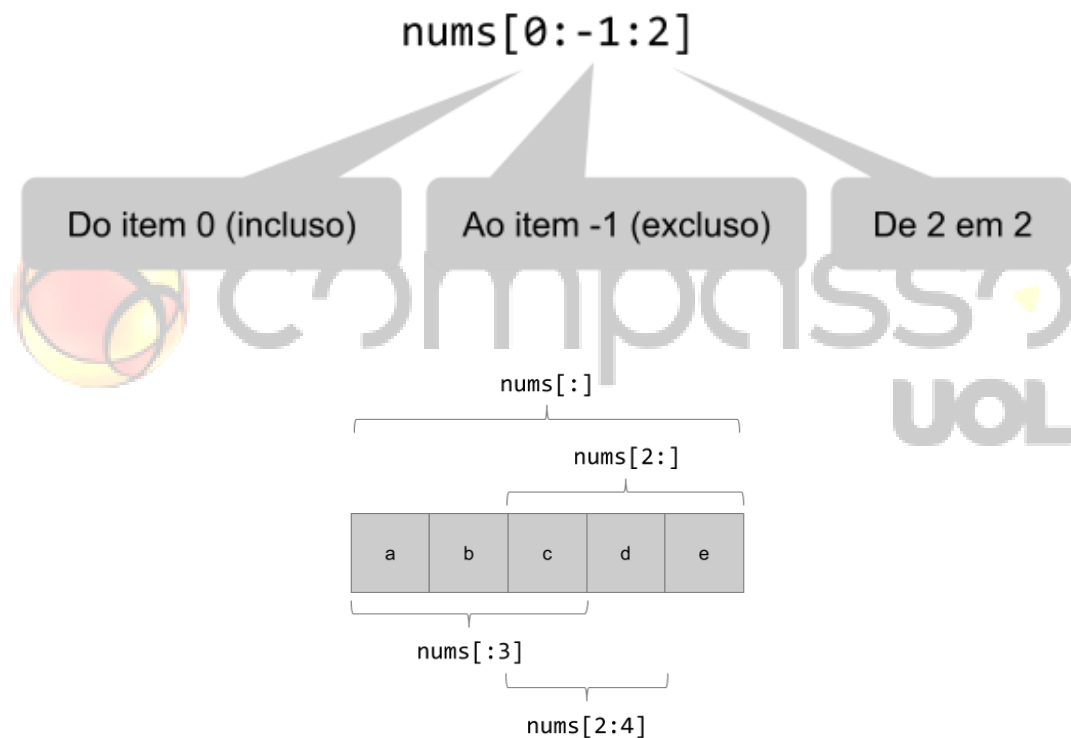
A indexação linear (para os contêineres que a suportam) em Python possui algumas particularidades:

- Suporta números negativos
- Nesse caso, indexa de trás para frente (-1, -2, ...)
- Aceita 3 parâmetros
 - Ponto de início (inclusivo)
 - Ponto de fim (exclusivo)
 - Passo

Dados na lista	a	b	c	d	e
Indexação crescente	0	1	2	3	4
Indexação decrescente	-5	-4	-3	-2	-1

Listas: slicing

- Slicing é a técnica de “fatiar” um contêiner que suporta indexação linear
- O fatiamento se dá adicionando um par de colchetes ao fim da variável
- Possui 3 parâmetros: início (incluso), fim (excluso) e passo



```
nums = list(range(5))
print(nums)
print(nums[:]) # do início ao fim
print(nums[2:]) # do 2o (incluso) até o fim
print(nums[:2]) # do início ao 2o (excluso)
print(nums[2:4]) # do 2o (incluso) ao 4o (excluso)
print(nums[:-1]) # do início ao último (excluso)
nums[2:4] = [8, 9] # atribuição
print(nums)
```

```
In [10]: nums = list(range(5))
print(nums)
print(nums[:])    # do início ao fim
print(nums[2:])   # do 2o (incluso) até o fim
print(nums[:2])   # do início ao 2o (excluso)
print(nums[2:4])  # do 2o (incluso) ao 4o (excluso)
print(nums[:-1])  # do início ao último (excluso)
nums[2:4] = [8, 9] # atribuição
print(nums)

[0, 1, 2, 3, 4]
[0, 1, 2, 3, 4]
[2, 3, 4]
[0, 1]
[2, 3]
[0, 1, 2, 3]
[0, 1, 8, 9, 4]
```

Listas: construção com list comprehension

- *List Comprehension* (compreensão de lista) é um mecanismo conciso de aplicar uma operação nos valores de uma lista
 - Cria uma nova lista
 - Cada elemento é o resultado de uma operação em outra sequência (e.g. uma outra lista)

```
L = [x**2 for x in range(1,7)]
```

Resultará na lista [1, 4, 9, 16, 25, 36]

- O comando for na compreensão pode ser seguido de um ou mais comandos if para filtrar o conteúdo da lista

```
mixed = [1, 2, 'a', 3, 4.0]
mixed = [x**2 for x in mixed if type(x) == int]
```

Aplicará a potência quadrática apenas nos números inteiros, resultando na lista [1, 4, 9]

```
In [11]: mixed = [1, 2, 'a', 3, 4.0]
mixed = [x**2 for x in mixed if type(x) == int]
print(mixed)

[1, 4, 9]
```

Listas: map

- map é uma função de alta ordem pré-definida
 - Aplica uma função em cada elemento de uma lista
 - Chamada map(f,l) onde f é uma função a ser aplicada na lista l
- Por exemplo, o código (utilizando a função lambda $x = x + 'a'$):

```
lista = ['b', 'n', 'n']
lista1 = list(map(lambda x: x + 'a', lista))
```

```
print(lista1)
```

Resultará na lista ['ba', 'na', 'na']

Listas: métodos

- L.append(e) adiciona um objeto e no final de L
- L.count(e) retorna o número de vezes que ocorre em L
- L.insert(i, e) insere o objeto e em L no índice i
- L.extend(L1) adiciona os itens da lista L1 no final de L
- L.remove(e) deleta a primeira ocorrência de e de L.
- L.index(e) retorna o índice da primeira ocorrência de e em L. Cria uma exceção caso e não esteja em L.
- L.pop(i) remove e retorna o elemento no índice i
 - Se i for omitido i será assumido como -1 por default, retornando o último elemento de L
- L.sort() ordena os elementos de L em ordem ascendente
- L.reverse() inverte a ordem dos elementos em L

```
In [12]: l = [3, 'a', 4, 'b', 1, 4, 3]
         l.sort()
         print(l)

-----
TypeError                                 Traceback (most recent call last)
<ipython-input-12-4a34dcdbd058> in <module>
      1 l = [3, 'a', 4, 'b', 1, 4, 3]
----> 2 l.sort()
      3 print(l)

TypeError: '<' not supported between instances of 'str' and 'int'
```

Tuplas

- Tuplas são sequências de elementos de qualquer tipo
- Exemplos:
 - Tuplas que representam produtos:

```
Prod1 = (10, "Banana", 1.5)
```

```
Prod2 = (22, "Maca", 4.5)
```

- Tuplas que representam pontos cartesianos:

```
P1 = (101, 22)
```

```
P2 = (-3, 18)
```

- São imutáveis
 - Uma vez criados, não podem ser estendidos, reduzidos ou alterados
 - Similares a strings

- Podem ser indexados tal como listas: e.g. P1[0] → 101
- Admitem slicing Prod2[1:] → ("Maca", 4.5)

- Preste atenção:

X = ("ola") → type(X) == String

X = ("ola",) → type(X) == Tuple


Tuplas: zip

- zip permite iterar sobre valores agregados de múltiplas coleções iteráveis
- Retorna um iterador de tuplas onde cada elemento iterado corresponde a uma tupla com o i-ésimo elemento de cada coleção

Por exemplo

```
x = [1, 2, 3]
y = [4, 5, 6]
zipped = list(zip(x, y))
```

Resultará na lista de tuplas [(1, 4), (2, 5), (3, 6)]



```
In [13]: x = [1, 2, 3]
        y = [4, 5, 6]

        for i, j in zip(x, y):
            print(i, j)

1 4
2 5
3 6
```

Tuplas: enumerate

- enumerate permite iterar sobre uma coleção mantendo uma variável com a contagem de elementos vistos até agora
 - Esta contagem corresponde ao índice de coleções sequenciais (e.g. strings, listas e tuplas)

Por exemplo:

```
In [14]: seasons = ['Spring', 'Summer', 'Fall', 'Winter']
tuples = list(enumerate(seasons))

print(tuples)

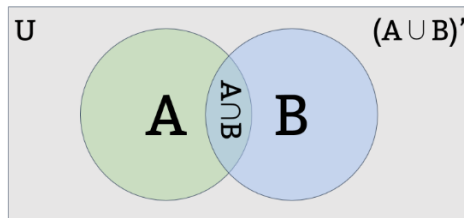
[(0, 'Spring'), (1, 'Summer'), (2, 'Fall'), (3, 'Winter')]
```

```
In [15]: seasons = ['Spring', 'Summer', 'Fall', 'Winter']
for i, season in enumerate(seasons):
    print(i, season)

0 Spring
1 Summer
2 Fall
3 Winter
```

Conjuntos

- Implementam uma coleção de objetos únicos respeitando propriedades de um conjunto matemático
- Implementam operadores de [teoria dos conjuntos](#)
- Não possuem ordem garantida



Docs: <https://docs.python.org/3/tutorial/datastructures.html#sets>

```
In [16]: a = {1, 2, 3}
        b = set([3, 4, 5])
```

```
In [17]: len(a)
```

```
Out[17]: 3
```

```
In [18]: 1 in a
```

```
Out[18]: True
```

```
In [19]: a ^ b
```

```
Out[19]: {1, 2, 4, 5}
```

```
In [20]: a | b
```

```
Out[20]: {1, 2, 3, 4, 5}
```

```
In [21]: a - b
```

```
Out[21]: {1, 2}
```

```
In [22]: a = {1, 2, 3}
        b = set([3, 4, 5])
```

```
In [23]: c = {1, 2}
```

```
In [24]: c.issubset(a)
```

```
Out[24]: True
```

```
In [25]: a.issuperset(c)
```

```
Out[25]: True
```

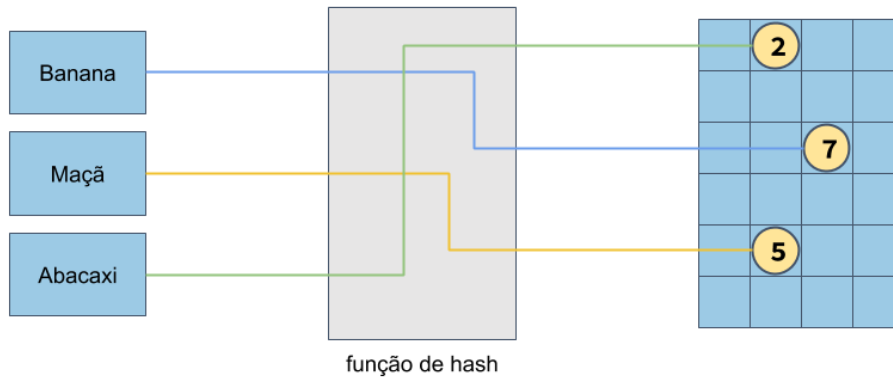
Conjuntos: operadores

- `len(s)`: número de elementos no conjunto `s` (cardinalidade)
- `x in s`: testa se `x` é um membro de `s`
- `x not in s` testa se `x` não é membro de `s`
- `s.issubset(t)` (equivalente a `s <= t`) testa se cada elemento de `s` está em `t`
- `s.issuperset(t)` (equivalente a `s >= t`) testa se cada elemento de `t` está em `s`
- `s.union(t)` (equivalente a `s | t`) gera um novo conjunto com todos os elementos de `s` e `t`
- `s.intersection(t)` (equivalente a `s & t`) gera um novo conjunto com os elementos comuns entre `s` e `t`
- `s.difference(t)` (equivalente a `s - t`) gera um novo conjunto com os elementos de `s` que não estão em `t`
- `s.symmetric_difference(t)` (equivalente a `s ^ t`) gera um novo conjunto com elementos em `s` ou `t` que não estão em ambos

Dicionários

- Dicionários são estruturas de dados indexadas indiretamente utilizando chaves no lugar de índices numéricos
- Dicionários implementam tabelas hash (hash tables)

- Indexação utiliza uma função hash que converte valores de tipos arbitrários para um índice numérico (idealmente único)
 - Todos os objetos em Python implementam uma função de hash padrão no método `__hash__(self)`



```
In [26]: d = {'camila': 21, 'roberto': 53, 'carla': 66}
```

```
In [27]: d['camila']
```

```
Out[27]: 21
```

```
In [28]: # print d[21] # resultará em um erro
```

```
In [29]: 'camila' in d
```

```
Out[29]: True
```

```
In [30]: for key, value in d.items():
          print('chave:', key, '\tvalor:', value)
```

```
chave: camila    valor: 21
chave: roberto   valor: 53
chave: carla     valor: 66
```

```
In [31]: del d['roberto']
```

```
In [32]: for key, value in d.items():
          print('chave:', key, '\tvalor:', value)
```

```
chave: camila    valor: 21
chave: carla     valor: 66
```

Contêineres: resumo

Estrutura	Indexação	Flexibilidade	Objetivo Principal
tuple	linear	imutável	• Organizar os objetos em pares ordenados
str			• Representar texto
list		mutável	• Organizar linearmente um conjunto de dados

set	não suporta		• Representação da Teoria dos Conjuntos
dict	hashable		• Organizar de forma não-ordenada um conjunto de dados; • Indexação por chaves

Itertools

Iteradores infinitos:

Iterator	Arguments	Results	Example
<code>count()</code>	start, [step]	start, start+step, start+2*step, ...	<code>count(10) --> 10 11 12 13 14 ...</code>
<code>cycle()</code>	p	p0, p1, ... plast, p0, p1, ...	<code>cycle('ABCD') --> A B C D A B C D ...</code>
<code>repeat()</code>	elem [,n]	elem, elem, elem, ... endlessly or up to n times	<code>repeat(10, 3) --> 10 10 10</code>

Leitura recomendada

- Documentação oficial sobre contêineres: <https://docs.python.org/3/library/stdtypes.html#list>
- Documentação sobre a biblioteca nativa itertools: <https://docs.python.org/3/library/itertools.html>

Funções em Python

Boas práticas de programação

- Grandes volumes de código não são necessariamente bons para legibilidade
- Idealmente, melhor codificar unidades de funcionalidade
- Para isto, utilizamos a noção de **funções**
- Mecanismo para atingir **decomposição** e **abstração**

Por que utilizar funções?

- Decomposição
 - Cria uma estrutura
 - Quebra o programa em partes razoavelmente autocontidas
 - Estrutura pode ser facilmente reusada em diferentes contextos
- Abstração
 - Esconde detalhes
 - Permite usar trecho de código como “caixa-preta”
 - Preserva a informação relevante para um contexto
 - Ignora detalhes que não são relevantes naquele contexto

Declaração de Funções

As declarações de função utilizam a palavra reservada **def**

```
def function():  
    pass
```

Parâmetros

- Parâmetros proveem algo chamado: **lambda abstraction**
 - Permite escrever código que manipula objetos não específicos
 - Os objetos serão definidos pelo chamador da função

Exemplo de sintaxe

```
def printName(nome, sobrenome, inverte=False):  
    if inverte:  
        print(nome, sobrenome)  
    else:  
        print(sobrenome, nome)
```

Tipos de parâmetros

- Posicionais:
 - A ordem dos parâmetros reais deve ser a mesma dos parâmetros formais
 - `printName("José", "da Silva", True)`
- Palavra-chave:
 - Usa-se o nome do parâmetro para indicar seu valor
 - `printName(sobrenome="da Silva", nome="José")`
- Default:
 - Se omitido assume valor padrão

```
def func(a, norm=False):  
    if norm:  
        return [(x - min(a)) / (max(a) - min(a)) for x in a]  
    else:  
        return a
```

```
In [33]: def func(a, norm=False):  
        if norm:  
            return [(x - min(a)) / (max(a) - min(a)) for x in a]  
        else:  
            return a  
  
        a = [1, 2, 3]  
        print(func(a))  
        print(func(a, False))  
        print(func(a, True))  
        print(func(a, norm=True))  
  
        [1, 2, 3]  
        [1, 2, 3]  
        [0.0, 0.5, 1.0]  
        [0.0, 0.5, 1.0]
```

Parâmetros não nomeados

- Se você não souber quantos parâmetros uma função receberá, utilize `*args` para denotar este comportamento:
 - Na função abaixo: `args` codifica uma lista de parâmetros sem nome que foram passados para a função

```
def func(*args):  
    for item in args:  
        print(item)
```

```
In [34]: def func(*args):  
         print(args)  
         for item in args:  
             print(item)  
func(1, 2, 3, 'a', ['b', 'c'], 6.3)  
  
(1, 2, 3, 'a', ['b', 'c'], 6.3)  
1  
2  
3  
a  
['b', 'c']  
6.3
```

Desenrolando listas em parâmetros não nomeados

- Da mesma maneira, você pode usar um comportamento similar ao chamar a função
 - Ao chamar a função `f(*args)` onde `args` é uma lista ou tupla, cada elemento da lista ocupará o espaço de um parâmetro.

```
def func(a, b, c):  
    print('a: {}, b: {}, c: {}'.format(a, b, c))  
  
params = [1, 2, 3]  
func(*params) # é equivalente a func(params[0], params[1], params[2])
```

```
In [35]: def func(a, b, c):  
         print('a: {}, b: {}, c: {}'.format(a, b, c))  
  
minha_lista = ['hello', 3, 1.7]  
func(*minha_lista)  
  
a: hello, b: 3, c: 1.7
```

Parâmetros não nomeados

- Se você não souber quantos parâmetros uma função receberá, mas apenas que eles devem ser nomeados, utilize `**kwargs`:
 - `kwargs` codifica um dicionário, onde a chave é o nome do parâmetro

```
def func(**kwargs):  
    for key, value in kwargs.items():  
        print(key, value)
```



```
In [36]: def func(*args, **kwargs):
          for item in args:
              print(item)
          for key, value in kwargs.items():
              print(key, value)
          # func(10) error
          func(10, param1=10, param2='hello', x=6.3)

10
param1 10
param2 hello
x 6.3
```

Desenrolando dicionários em parâmetros nomeados

- Da mesma maneira, você pode usar um comportamento similar ao chamar a função
 - Ao chamar a função `f(**args)` onde `args` é dicionário, cada chave do dicionário assumirá o nome do parâmetro e cada valor assumirá o valor do parâmetro.

```
def func(a, b, c):
    print('a: {}, b: {}, c: {}'.format(a, b, c))

params = {'a': 10, 'b': 'hello', 'c': 1.7}
func(**params) # é equivalente a func(a=params['a'], b=params['b'], c=params['c'])
```

```
In [37]: def func(a, b, c):
          print('a: {}, b: {}, c: {}'.format(a, b, c))

          params = {'b': 'hello', 'a': 10, 'c': 1.7}
          func(**params)

a: 10, b: hello, c: 1.7
```

Tipos de retorno

- Python permite que uma função retorne vários objetos de apenas uma vez. Por exemplo

```
def func(a, b, c):
    return a**2, b**3, c**4

a, b, c = func(2, 3, 4)
```

```
In [38]: def func(a, b, c):
          return a**2, b**3, c**4

          a, b, c = func(2, 3, 4)
          print(a, b, c)

4 27 256
```

Desenrolando listas e tuplas

- Da mesma maneira, podemos "desenrolar" em variáveis listas e tuplas a qualquer momento.

```
a = [1, 2, 3]
b, c, d = a
print(b, c, d)
```

```
In [39]: a = [1, 2, 4]
         b, c, d = a
         print(b, c, d)

1 2 4
```

Verificação de tipos

- Em python 3.X, o programador pode documentar o tipo dos parâmetros
- Python 2.7 não possui verificação automática de tipos
 - Código abaixo ilustra como se faz verificação explícita de tipos em Python 2.7

```
def func(param):
    if isinstance(param, list):
        print('list')
    else:
        print(type(param))
```

```
In [40]: def my_func(some_list: list):
         for p in some_list:
             print(p)

         a = [1, 2, 3]
         my_func(a)

1
2
3
```

```
In [41]: def my_func(some_list: list) -> list:
         for i in range(len(some_list)):
             some_list[i] **= 2 # eleva cada membro da lista ao quadrado
         return some_list

         a = [1, 2, 3]
         my_func(a)
```

```
Out[41]: [1, 4, 9]
```

Documentação de funções

- Definem um *contrato* entre o programador que escreve a função e os programadores que irão usar a função em seus programas
- O contrato tem duas partes:
 - Pré condições (premissas): asserções que devem ser verdadeiras para que a função possa ser utilizada
 - Pós condições (garantias): asserções que o desenvolvedor da função garante que serão verdadeiras após a execução da função

```
In [42]: def findRoot(x, power, epsilon):  
    """  
    :param x: base  
    :type x: float  
    :param power: Expoente. deve ser maior ou igual a 1  
    :type power: int  
    :param epsilon: margem de erro. Deve ser maior que zero.  
    :type epsilon:  
    :return: Retorna um valor y, de tal forma que y ** power é aproximado de x  
    (dada a margem de erro epsilon). Se este valor não existir, retorna None  
    :rtype: float  
    """  
    if x < 0 and power % 2 == 0:  
        return None  
    low = min(-1.0, x)  
    high = max(1.0, x)  
    ans = (high + low)/2.0  
    while abs(ans ** power - x) >= epsilon:  
        if ans ** power < x:  
            low = ans  
        else:  
            high = ans  
        ans = (high + low)/2.0  
    return ans
```

```
In [43]: expoente = 2  
x = 10  
epsilon = 0.1  
y = findRoot(x, expoente, epsilon)  
  
print('achando a base para o expoente 2 que mais se aproxima de 10:', y)  
print('%0.2f^%.0f = %f = %f - %f' % (y, expoente, (y**expoente), x, epsilon))
```

```
achando a base para o expoente 2 que mais se aproxima de 10: 3.16796875  
3.17^2 = 10.036026 = 10.000000 - 0.100000
```

Referência para funções

- Funções em Python são "objetos de primeira classe"
 - Podem ser manipuladas como qualquer outro objeto
 - Podem ser passadas por parâmetro
 - Permitem realizar programação de alta ordem (útil para listas)

```
In [44]: def addOne(n):  
         return n+1  
  
         def applyToEach(maxN, f):  
             for i in range(1,maxN):  
                 print(f(i))  
  
         applyToEach(10, addOne)  
  
2  
3  
4  
5  
6  
7  
8  
9  
10
```

Funções λ (lambda)

- São como funções anônimas em Java
- Retornam valores, porém não precisam do statement "return"
- Geralmente são usadas em um escopo muito curto e restrito, como a chamada de uma função
- Podem ser atribuídas a variáveis:

```
In [45]: def f(x):  
         return x**2  
  
         func = f  
         func(3.5)
```

```
Out[45]: 12.25
```

Módulos

- Funções são a unidade fundamental de decomposição
- Programas muito grandes costumam exigir recursos adicionais
- Módulos permitem agrupar conjuntos de variáveis e funções em um único arquivo
- Cada módulo define seu próprio contexto
- Um módulo pode importar outros usando o comando import
- Usa-se <nome do modulo>.<nome da função> para acessar funções definidas em outros módulos

Exemplo

Módulo em circulo.py

```
pi = 3.14159
```

```
def area(raio):  
    return pi*(raio**2)  
  
def perimetro(raio):  
    return 2*pi*raio  
  
def superficie(raio):  
    return 4.0*area(raio)  
  
def volume(raio):  
    return(4.0/3.0)*pi*(raio**3)
```

Uso do módulo

```
import circulo as cr  
from circulo import perimetro  
  
raio = float(input("Digite o raio do circulo: "))  
  
print("Area do circulo:", cr.area(raio))  
print("perímetro do circulo:", perimetro(raio))  
print("Volume da esfera:", cr.volume(raio))
```

Definição de módulos

Para definir um módulo (dentro da pasta de um projeto), é necessário:

- Especificar o nome do módulo como o nome da pasta que conterà os arquivos de código
 - Por exemplo, treelib
- Colocar um arquivo `__init__.py` dentro desta pasta
 - Este arquivo contém as instruções iniciais que devem ser executadas assim que um módulo é importado. Se nenhuma instrução for necessária, ele pode ficar em branco
- Colocar os outros arquivos de código fonte nesta pasta
 - Por exemplo, um arquivo `creation.py` dentro da pasta do módulo `treelib`
 - Ele será importado com a seguinte sintaxe: `treelib.creation`

Arquivos

Cria arquivo:

```
import random
arq = open("numeros.txt", "w")
for i in range(0, 100):
    val = random.random()
    arq.write(str(val) + "\n")
print("Arquivo criado")
arq.close()
```

Lê arquivo:

```
arq = open("numeros.txt", "r")
for line in arq.readlines():
    print(float(line)*10)
arq.close()
```

Ou ainda:

```
with open("numeros.txt", "r") as f:
    for line in f.readlines():
        print(float(line)*10)
```

```
In [46]: import random
         arq = open("numeros.txt", "w")
         for i in range(0, 30):
             val = random.random()
             arq.write(str(val) + "\n")
         print("Arquivo criado")
         arq.close()

         print("Lendo conteúdo do arquivo...")
         with open("numeros.txt", "r") as f:
             for line in f.readlines():
                 print(float(line)*10)
```

```
Arquivo criado
Lendo conteúdo do arquivo...
0.9709881950153887
3.520620888031262
1.3355152946863147
2.572769876216703
1.296121720878206
4.233718195586964
6.550763938221236
2.6752533254268753
2.537991675050466
8.384637422946838
2.907415823678371
9.823243507420962
2.912445500248668
4.935236595199438
5.281561648019805
6.3523460503041695
3.7102143891470907
5.931021656036075
6.233802026930198
4.940314266180743
0.4837371227185794
6.812073927180312
0.4955775293368436
1.3397112558844815
5.171307124752049
8.032219520765052
3.0368837654094616
0.4395139188622532
5.681620199379393
1.5416384405994799
```

Classes em Python

Introdução

- Não existem palavras reservadas como public, private, static, final
- Permite herança múltipla
- Classes podem ser modificadas em tempo de execução

Declarando a classe mais simples

- O nome da classe deve ser declarado utilizando a notação CamelCase
- Todas as classes descendem da classe object em algum nível

Declaração:

```
In [1]: class MyClass(object):  
        pass # NO-OP
```

Utilização:

```
In [2]: inst = MyClass()
```

Construtor

- Independente do nome da classe, o nome do construtor **sempre** será `__init__`
- O primeiro parâmetro do construtor é **sempre** a instância que está chamando o construtor
 - Por convenção, o nome deste parâmetro é `self`
 - Pode assumir qualquer outro nome, mas não é recomendado por prejudicar a legibilidade
 - Este mesmo comportamento ocorre com outros métodos

```
In [3]: class MyClass(object):  
        def __init__(self, value):  
            print('o tipo do parâmetro value é %r' % type(value))  
            print('o tipo do parâmetro self é %r' % type(self))
```

```
In [4]: inst = MyClass('abc')
```

```
o tipo do parâmetro value é <class 'str'>  
o tipo do parâmetro self é <class '__main__.MyClass'>
```

```
In [5]: inst = MyClass(123)
```

```
o tipo do parâmetro value é <class 'int'>  
o tipo do parâmetro self é <class '__main__.MyClass'>
```


A comunidade Python recomenda, para fins de clareza, que todos os atributos de uma classe sejam declarados e instanciados em seu construtor:

```
In [6]: class Carro(object):
        # a própria instância deve ser o primeiro parâmetro
        def __init__(self, fabricante, modelo):
            # atribuindo o valor do parâmetro fabricante ao atributo fabricante desta instância
            self.fabricante = fabricante
            self.modelo = modelo

        carro1 = Carro('toyota', 'corolla')
        print(carro1.fabricante, carro1.modelo)

toyota corolla
```

Getters e setters

- Métodos getter são decorados com `@property`
 - são acessados como se fossem atributos
- Métodos setter são decorados com `@<nome_da_variavel>.setter`
 - são atribuídos como se fossem atributos

Declaração:

```
In [8]: class MyClass(object):
        def __init__(self, var): # construtor da classe
            self._var = var # atribui o valor do parâmetro var ao atributo _var

        @property # decorador de getter
        def var(self):
            return self._var # retorna o valor do atributo _var

        @var.setter # decorador de setter
        def var(self, value):
            if value > 0:
                self._var = value # atribui o valor do parâmetro value ao atributo _var
            else:
                print('o valor atribuído é inválido')
```

Utilização:

```
In [9]: instance = MyClass(1)
        print(instance.var)
```

1

```
In [10]: instance.var = 2
         print(instance.var)
```

2

```
In [11]: instance.var = -1
         print(instance.var)
```

o valor atribuído é inválido
2

Destrutores

Existem destrutores para **variáveis** também:

Declaração:

```
In [12]: class MyClass(object):
          def __init__(self, var):
              self._var = var

          @property
          def var(self):
              return self._var

          @var.setter
          def var(self, value):
              self._var = value

          @var.deleter
          def var(self):
              print('deletando var!')
              del self._var
```

Utilização:

```
In [13]: instance = MyClass(1)
          del instance.var
```

deletando var!

E se eu quiser...

- ...declarar um **método final**?[1]
 - Você não pode!
- ...declarar métodos e atributos privados/protegidos, para nenhuma classe descendente acessar?
 - você não pode!

[1] Um método é dito final quando não pode ser sobrescrito por nenhuma classe descendente

Métodos e atributos privados (mais ou menos)

Para declará-los, você deve se basear na **documentação**:

```
In [47]: class MyClass(object):
          def __method__(self, var1):
              self._attribute = var1
```

- O underline _ (um para atributos, dois para métodos) notifica outros programadores que aquele método/atributo não deve ser acessado por eles, e que se o fizerem, resultará em um comportamento inesperado
- Isso é um padrão da comunidade Python (não é enforcado pelos interpretadores)

- O principal motivo para isto é a falta de fé em métodos 100% seguros
- [Acessando métodos/atributos privados em Java](#)

Métodos e atributos privados

Outro exemplo de métodos e atributos privados:

```
In [48]: class Aviao(object):
    def __init__(self, motor):
        self._motor = motor # atributo que não deve ser acessado fora do escopo de Aviao

    def voa(self):
        self.__queima_combustivel__()

    def __queima_combustivel__(self): # método que não deve ser acessado fora do escopo de Aviao
        print('queimando combustivel com o motor %s' % self._motor)

    @property # metodologia correta para acessar um atributo privado
    def motor(self):
        return self._motor

atr_72 = Aviao('turboprop')
atr_72.voa()
# note que uma propriedade é tratada como um atributo na chamada, mas implementada como uma função na classe
print(atr_72.motor)

# print(atr_72._motor) # na prática é acessível; porém você não deve fazer isso!
# atr_72.__queima_combustivel__() # na prática é acessível; porém você não deve fazer isso!

queimando combustivel com o motor turboprop
turboprop
```

Métodos estáticos

Métodos estáticos não interagem com nenhum atributo da instância

Declaração:

```
In [49]: class MyClass(object):
    @staticmethod # decorador para métodos estáticos
    def my_static_method(): # não exigem a passagem do parâmetro self
        return 'hello world!'
```

Instanciação:

```
In [50]: instance = MyClass()
print('método da classe:\t', MyClass.my_static_method())
print('método da instância:\t', instance.my_static_method())

método da classe:      hello world!
método da instância:   hello world!
```

Métodos de classe

Métodos de classe são voltados a instanciar objetos a partir de outros métodos que não os construtores

```
In [51]: class Date(object):
    day = 1
    month = 1
    year = 1970
    def __init__(self, day, month, year):
        self.day, self.month, self.year = (day, month, year)
    @classmethod
    def from_string(cls, string): # o primeiro parâmetro de um método de classe é a própria classe
        some_list = string.split('-')
        date = cls(some_list[0], some_list[1], some_list[2])
        return date
```

Atributos declarados na classe são comuns a todas as instâncias. Considere esse o valor default para o atributo year:

```
In [52]: print(Date.year)  # valor referente a classe

1970
```

Usando um método de classe:

```
In [53]: date1 = Date(2, 3, 1973)
         date2 = Date.from_string('2-3-1973')
         print('usando construtor:\t\t', date1.year)
         print('usando um método de classe:\t', date2.year)

         usando construtor:          1973
         usando um método de classe: 1973
```

Métodos de classe

- Em Python, não existe sobrecarga de métodos a nível da mesma classe (apenas classes descendentes podem sobrescrever métodos das classes pai)
 - também pode ser chamado de polimorfismo
- Portanto, métodos de classe são importantes para tratar diversos tipos de dados
- Mais sobre métodos estáticos e métodos de classe neste [link](#)

Métodos de classe

Para que atributos de uma classe possuam valores default, declare-os no corpo da classe:

```
In [54]: class Bairro(object):
         nome = 'sem nome'
         def __init__(self, nome):
             self.nome = nome  # sobrescreve o atributo nome desta instância, mas não desta classe

         bairro1 = Bairro('partenon')
         print(bairro1.nome)
         print(Bairro.nome)
         # sobrescreve o valor do atributo "nome" da classe; válido apenas durante a execução do código
         Bairro.nome = 'ainda sem nome'
         print(Bairro.nome)

         partenon
         sem nome
         ainda sem nome
```

Herança múltipla

Python não restringe herança múltipla como outras linguagens de programação podem fazer:

```
In [55]: class Fulano(object):
        def fala(self):
            print("sou um fulano")

        class Ciclano(Fulano):
            def fala(self):
                print("sou um ciclano")

        class Sicrano(Fulano):
            def fala(self):
                print("sou um sicrano")

        class Beltrano(Ciclano, Sicrano): # herança múltipla
            pass
```

Sobrescrita de métodos

Para sobrescrever métodos de uma superclasse, basta reescrevê-la na subclasse:

```
In [56]: class Animal(object):
        def mover(*args, **kwargs):
            print('implemente!')

        class Cobra(Animal):
            def mover(*args, **kwargs):
                print('rasteja')

        print('algum animal:')
        algum_animal = Animal()
        algum_animal.mover()

        print('naja:')
        naja = Cobra()
        naja.mover()

        algum animal:
        implemente!
        naja:
        rasteja
```

Sobrecarga de operadores

Em Python, operadores podem ser [sobrescritos](#):

- + - / *
- < > >= <= == !=
- A sobrecarga é similar à que ocorre em C++

Declaração:

```
In [57]: class Vector(object):
        coords = None
        def __init__(self, coords):
            self.coords = coords
        def __add__(self, other): # operador +
            return [x[0] + x[1] for x in zip(self.coords, other.coords)]
        def __str__(self): # operador print
            return '[' + ', '.join([str(x) for x in self.coords]) + '']
```

Sobrecarga de operadores

Instanciação:

```
In [58]: a, b = Vector([2, 3, 5]), Vector([7, 11, 13])
         c = a + b
         print(a)
         print(b)
         print(c)

[2, 3, 5]
[7, 11, 13]
[9, 14, 18]
```

Leitura recomendada

- Documentação oficial sobre classes: <https://docs.python.org/3/tutorial/classes.html>
- Sobrecarga de operadores em Python: <http://blog.teamtreehouse.com/operator-overloading-python>

Exercícios – Parte 2

Exercício 1

Dada duas listas como no exemplo abaixo:

a = [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]

b = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]

Escreva um programa que retorne o que as listas têm comum (sem repetições). O seu programa deve funcionar para listas de qualquer tamanho.

Exercício 2

Dada a seguinte lista:

a = [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

Faça um programa que gere uma nova lista contendo apenas números ímpares.

Exercício 3

Peça para o usuário digitar uma palavra pelo teclado e determina se a palavra digitada é ou não um palíndromo. Um palíndromo é uma palavra que permanece igual se lida de trás pra frente.

Exercício 4

Dada as listas a seguir:

primeirosNomes = ['João', 'Douglas', 'Lucas', 'José']

sobrenomes = ['Soares', 'Souza', 'Silveira', 'Pedreira']

idades = [19, 28, 25, 31]

Faça um programa que imprima os dados na seguinte estrutura: " - está com anos"

Exercício 5

Escreva uma função que recebe uma lista e retorna uma nova lista sem elementos duplicados. Utilize um exemplo para testar sua função.

Exercício 6

Escreva um programa que lê o conteúdo do arquivo texto `arquivo_texto.txt` e imprime o seu conteúdo.

Dica: leia documentação da função `open(...)`,

link: <https://docs.python.org/3/library/functions.html#open>

Exercício 7

Leia o arquivo `person.json`, faça o parsing e imprima seu conteúdo.

Dica: leia a documentação do pacote json, link: <https://docs.python.org/3/library/json.html>

Exercício 8

Implemente a função `my_map(list, f)` que recebe uma lista como primeiro argumento e uma função como segundo argumento. Esta função aplica a função recebida para cada elemento da lista recebida e retorna o resultado em uma nova lista.

Teste sua função para saber se está ok.

Exercício 9

Escreva uma função que recebe um número variável de parâmetros não nomeados e um número variado de parâmetros nomeados e imprime o **valor** de cada parâmetro recebido.

Exercício 10

Implemente a classe `Lampada`. A classe `Lâmpada` recebe um booleano no seu construtor, `True` se a lâmpada estiver ligada, `False` caso esteja desligada. A classe `Lampada` possui os seguintes métodos:

- `liga()`: muda o estado da lâmpada para ligada
- `desliga()`: muda o estado da lâmpada para desligada
- `esta_ligada()`: retorna verdadeiro se a lâmpada estiver ligada, falso caso contrário

Exercício 11

Escreva um programa que leia do teclado uma sequência de número separados por vírgula (e.g. 2,4,5,6,1,6) e imprime a soma de todos eles.

Exercício 12

Escreva uma função que recebe como parâmetro uma lista e retorna 3 listas: a lista recebida dividida em 3 partes iguais. Teste sua implementação

b = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]

Exercício 13

Dado o dicionário a seguir:

```
speed = {'jan':47, 'feb':52, 'march':47, 'April':44, 'May':52, 'June':53, 'july':54, 'Aug':44, 'Sept':54}
```

Crie uma lista com todos os **valores** (não as chaves!) e coloque numa lista de forma que não haja valores duplicados.

Exercício 14

Calcule o valor mínimo, valor máximo, valor médio e a mediana da lista gerada na célula abaixo:

Obs.: Lembrem-se, para calcular a mediana a lista deve estar **ordenada**!

```
import random  
  
# amostra aleatoriamente 50 números do intervalo 0...500  
  
random_list = random.sample(range(500), 50)
```

Exercício 15

Imprima a lista da célula abaixo de trás para frente.

```
a = [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Exercício 16

Leia o arquivo actors.csv e faça os seguintes cálculos sobre o conjunto de dados:

1. O ator/atriz com maior número de filmes e o respectivo número de filmes.
2. A média do número de filmes.
3. O ator/atriz com a maior média por filme.
4. O nome do(s) filme(s) mais frequente(s) e sua respectiva frequência.

NumPy

Introdução

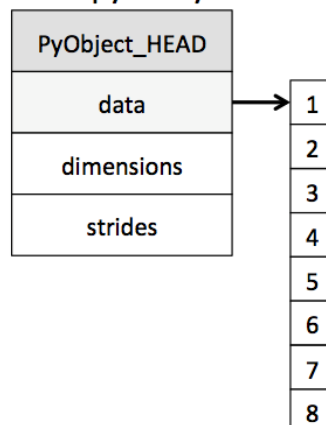
- NumPy é uma biblioteca para cálculo vetorial e matricial disponibilizada em Python
- Várias outras bibliotecas utilizam NumPy como base para seus cálculos
- Utilizar NumPy ao invés das estruturas básicas de Python (e.g. listas) apresenta melhorias de desempenho
- Mais informações em <https://numpy.org/>
- Instalar NumPy via pip

```
C:\WINDOWS\system32>pip install numpy
Collecting numpy
  Using cached https://files.pythonhosted.org/packages/bd/51/7df1a3858ff0465f760b482514f1292836f8be08d84aba411b48dda72f9/numpy-1.17.2-cp37-cp37m-win_amd64.whl
Installing collected packages: numpy
Successfully installed numpy-1.17.2
WARNING: You are using pip version 19.2.1, however version 19.2.3 is available.
You should consider upgrading via the 'python -m pip install --upgrade pip' command.
```

Arrays NumPy

- Funcionam como arrays em C++/Java
- Alocam espaços contíguos em memória
- São utilizados em várias funções numpy

NumPy Array



Arrays NumPy

- Utilizar arrays permite cálculos ainda mais rápidos que funções em C de Python
- Abaixo realizamos três testes: usando list comprehension, map e lambda, e arrays com funções numpy
- Os testes são executados 100 vezes e a média é mostrada

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

```
In [2]: setup = 'a = range(1000000)'
stmt = '[x**2 for x in a]'

times = timeit.repeat(setup=setup, stmt=stmt, number=1, repeat=100)
print('média de 100 execuções com list comprehension: %r segundos' % np.mean(times))

média de 100 execuções com list comprehension: 0.338589221 segundos
```

```
In [3]: setup = 'a = range(1000000); pow = lambda x: x**2'
stmt = 'list(map(pow, a))'

times = timeit.repeat(setup=setup, stmt=stmt, number=1, repeat=100)
print('média de tempos usando map: %r segundos' % np.mean(times))

média de tempos usando map: 0.38213473000000008 segundos
```

```
In [4]: setup = 'import numpy as np; a = np.arange(1000000)'
stmt = 'a**2'

times = timeit.repeat(setup=setup, stmt=stmt, number=1, repeat=100)
print('média de tempos usando numpy arrays: %r segundos' % np.mean(times))

média de tempos usando numpy arrays: 0.0025919060000012453 segundos
```

Arrays NumPy

Por que numpy é tão rápido?

- Além de usar dados contíguos em memória, as funções numpy usam a biblioteca BLAS (Basic Linear Algebra Subprograms):
 - São sub-rotinas para realizar cálculos matemáticos e matriciais, disponíveis para CPUs e GPUs
 - MATLAB e Octave também utilizam BLAS
- São estruturas homogêneas
- Possui um tipo, dentre eles:
 - **np.int8**
 - **np.int16**
 - **np.int32**
 - **np.int64**
 - **np.uint8**
 - **np.uint16**
 - **np.uint32**
 - **np.uint64**
 - **np.float32**
 - **np.float64**
 - **np.complex64**
 - **np.complex128**
 - **np.unicode** (Utilizado para string. Possui algum tamanho específico, e.g. <U10)
 - Lista completa em: [Numpy Data types](#)
- Possuem os seguintes atributos:
 - **ndarray.ndim**: Número de dimensões do array (e.g. 2 se for uma matriz)
 - **ndarray.shape**: Tupla que representa o formato do array (e.g. (3,3))
 - **ndarray.size**: Número total de elementos do array, é equivalente a `np.prod(ndarray.shape)`
 - **ndarray.dtype**: Tipo de dado do array (e.g. `np.float32`)
 - **ndarray.itemsize**: Tamanho em bytes que cada elemento do array ocupa (e.g. 8 para um `np.float64`)

Construção de arrays

Existem diversas formas de iniciar um array em numpy:

- **np.array(...)**
- **np.arange(...)**
- **np.zeros(...)**
- **np.ones(...)**
- **np.loadtxt(...)**
- [np.array\(...\)](#)
 - Constrói um array com base nos dados que são passados a estrutura
 - Dados podem ser provenientes de outros contêineres (e.g. lista, tupla)

```
In [5]: import numpy as np
        tupla = ('a', 'b', 'c')
        lista = [1, 2, 3]

        a = np.array(lista)
        b = np.array(tupla)
        print('Conteúdo: {}, shape: {}, Tipo: {}'.format(a, a.shape, a.dtype))
        print('Conteúdo: {}, shape: {}, Tipo: {}'.format(b, b.shape, b.dtype))

        Conteúdo: [1 2 3], shape: (3,), Tipo: int32
        Conteúdo: ['a' 'b' 'c'], shape: (3,), Tipo: <U1
```


```
In [6]: a = np.array([1, 2, 3], dtype=np.int32) # inicia um array de inteiros
        b = np.array([1, 2, 3], dtype=np.float32) # inicia um array de ponto flutuante

        print('Conteúdo: {}, shape: {}, Tipo: {}'.format(a, a.shape, a.dtype))
        print('Conteúdo: {}, shape: {}, Tipo: {}'.format(b, b.shape, b.dtype))

        Conteúdo: [1 2 3], shape: (3,), Tipo: int32
        Conteúdo: [1. 2. 3.], shape: (3,), Tipo: float32
```


1D array

```
>>> import numpy as np
>>> x = np.arange(2, 5).reshape(3)
>>> x
array([ 2, 3, 4])
>>>
```



2D array

```
>>> import numpy as np
>>> x = np.arange(2, 10).reshape(2, 4)
>>> x
array([[ 2, 3, 4, 5],
       [ 6, 7, 8, 9]])
>>>
```



3D array

```
>>> import numpy as np
>>> x = np.arange(24).reshape(4, 3, 2)
>>> x
array([[[ 0, 1], [ 6, 7], [12, 13], [18, 19]],
       [[ 2, 3], [ 8, 9], [14, 15], [20, 21]],
       [[ 4, 5], [10, 11], [16, 17], [22, 23]]])
>>>
```



- [np.arange\(...\)](#)
 - Mesmo funcionamento da função range
 - Várias assinaturas da função

```
In [7]: import numpy as np
```

```
In [8]: a = np.arange(10) # limite superior (excluso)
b = np.arange(2, 5) # limite inferior (incluso) e superior (excluso)
c = np.arange(0, 10, 2) # limite inferior (incluso), superior (excluso), passo
d = np.arange(5, dtype=np.float32)

print('Conteúdo: {}, shape: {}, Tipo: {}'.format(a, a.shape, a.dtype))
print('Conteúdo: {}, shape: {}, Tipo: {}'.format(b, b.shape, b.dtype))
print('Conteúdo: {}, shape: {}, Tipo: {}'.format(c, c.shape, c.dtype))
print('Conteúdo: {}, shape: {}, Tipo: {}'.format(d, d.shape, d.dtype))

Conteúdo: [0 1 2 3 4 5 6 7 8 9], shape: (10,), Tipo: int32
Conteúdo: [2 3 4], shape: (3,), Tipo: int32
Conteúdo: [0 2 4 6 8], shape: (5,), Tipo: int32
Conteúdo: [0. 1. 2. 3. 4.], shape: (5,), Tipo: float32
```

- [zeros e ones](#)
 - constrói arrays onde todos os valores são zeros ou uns

```
In [9]: import numpy as np
```

```
In [10]: a = np.zeros((3,3))
b = np.ones(7)

print('Conteúdo: {}, shape: {}, Tipo: {}'.format(a, a.shape, a.dtype))
print('Conteúdo: {}, shape: {}, Tipo: {}'.format(b, b.shape, b.dtype))

Conteúdo: [[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]], shape: (3, 3), Tipo: float64
Conteúdo: [1. 1. 1. 1. 1. 1. 1.], shape: (7,), Tipo: float64
```

```
In [11]: c = np.zeros((10, 10))
d = np.ones((5, 5), dtype=np.int32)

print('Conteúdo: {}, shape: {}, Tipo: {}'.format(c, c.shape, c.dtype))
print('Conteúdo: {}, shape: {}, Tipo: {}'.format(d, d.shape, d.dtype))

Conteúdo: [[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. 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. 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. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]], shape: (10, 10), Tipo: float64
Conteúdo: [[1 1 1 1 1]
 [1 1 1 1 1]
 [1 1 1 1 1]
 [1 1 1 1 1]
 [1 1 1 1 1]], shape: (5, 5), Tipo: int32
```

- Cada uma das funções vistas anteriormente possui diversos parâmetros que permitem um melhor controle das suas funções
- Vale a pena olhar a documentação de cada uma delas

Slicing

- Assim como em listas de Python, numpy arrays também podem ser fatiados (*slicing*)
 - Slicing é a técnica de “fatiar” um contêiner que suporta indexação linear
 - O fatiamento se dá adicionando um par de colchetes ao fim da variável
 - Possui 3 parâmetros: início (inclusivo), fim (exclusivo) e passo

```
In [12]: nums = np.arange(5)
print('array completo:\t\t\t', nums)
print('do início ao fim:\t\t\t', nums[:])
print('do 2o (inclusivo) até o fim:\t\t', nums[2:])
print('do início ao 2o (exclusivo):\t\t', nums[:2])
print('do 2o (inclusivo) ao 4o (exclusivo):', nums[2:4])
print('do início ao último (exclusivo)\t\t', nums[:-1])
nums[2:4] = [8, 9] # atribuição
print('após atribuição:\t\t\t', nums)

array completo:                [0 1 2 3 4]
do início ao fim:              [0 1 2 3 4]
do 2o (inclusivo) até o fim:   [2 3 4]
do início ao 2o (exclusivo):   [0 1]
do 2o (inclusivo) ao 4o (exclusivo): [2 3]
do início ao último (exclusivo) [0 1 2 3]
após atribuição:               [0 1 8 9 4]

In [13]: a = np.random.uniform(size=(3,5))
print('Conteúdo:\n{}\nshape: {},\ntipo: {}'.format(a, a.shape, a.dtype))
print('Selecionando a primeira linha:\n', a[0, :])
print('Selecionando a primeira coluna:', a[:, 0])
print('Selecionando a última linha:\n', a[-1, :])
print('Selecionando as colunas 1 e 2:\n', a[:, 1:3])

Conteúdo:
[[0.73408075 0.430622 0.08496626 0.79776572 0.68468292]
 [0.43489659 0.67935446 0.40029743 0.09846711 0.01860839]
 [0.93130564 0.72289667 0.90646891 0.00391196 0.58886441]]
shape: (3, 5),
Tipo: float64

Selecionando a primeira linha:
[0.73408075 0.430622 0.08496626 0.79776572 0.68468292]
Selecionando a primeira coluna: [0.73408075 0.43489659 0.93130564]
Selecionando a última linha:
[0.93130564 0.72289667 0.90646891 0.00391196 0.58886441]
Selecionando as colunas 1 e 2:
[[0.430622 0.08496626]
 [0.67935446 0.40029743]
 [0.72289667 0.90646891]]
```

Operações com arrays

Algumas operações com arrays incluem:

- divisão, multiplicação, soma, subtração
- junção
- transposição
- atribuição
- operações binárias e lógicas

Algumas operações com arrays incluem:

- divisão, multiplicação, soma, subtração

```
In [14]: a = np.arange(0, 5)
b = np.arange(5, 10)
print(a, a.dtype)
print(b, b.dtype)
```

```
[0 1 2 3 4] int32
[5 6 7 8 9] int32
```

```
In [15]: print(a * b)
print(a * 5)
```

```
[ 0  6 14 24 36]
[ 0  5 10 15 20]
```

```
In [16]: print(a / b)
print(a / 5)
```

```
[0.          0.16666667 0.28571429 0.375          0.44444444]
[0.   0.2 0.4 0.6 0.8]
```

```
In [17]: print(a - b)
print(a - 5)
```

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

```
In [18]: print(a + b)
print(a + 5)
```

```
[ 5  7  9 11 13]
[5 6 7 8 9]
```

```
In [19]: print(a, a.dtype)
a = a.astype(np.float32)
a /= b
print(a)
```

```
[0 1 2 3 4] int32
[0.          0.16666667 0.2857143  0.375          0.44444445]
```

- **Transposição**

```
In [20]: # transposição
a = np.array([
    [0, 1, 2],
    [3, 4, 5],
    [6, 7, 8]
])
print('Conteúdo:\n{}, shape: {}, Tipo: {}'.format(a, a.shape, a.dtype))
```

Conteúdo:
[[0 1 2]
 [3 4 5]
 [6 7 8]], shape: (3, 3), Tipo: int32

```
In [21]: print(a)
print(a.T)
```

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

```
In [22]: print(np.transpose(a))
```

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

- **Transformação**

```
In [23]: import numpy as np
```

```
In [24]: a = b = np.arange(9)
print(a, a.shape)
```

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

```
In [25]: a = a.reshape(-1, 3)
a[0, 0] = 10
print(a, a.shape)
print(b, b.shape)
```

[[10 1 2]
 [3 4 5]
 [6 7 8]] (3, 3)
[10 1 2 3 4 5 6 7 8] (9,)

```
In [26]: print(a.ravel())
```

[10 1 2 3 4 5 6 7 8]

```
In [27]: print(a.flatten())
```

[10 1 2 3 4 5 6 7 8]

- **Somas e médias**

```
In [28]: a = np.arange(0, 9).reshape(3, 3)
print('Conteúdo:\n{0}, shape: {1}, Tipo: {2}'.format(a, a.shape, a.dtype))

print('Soma total dos elementos:\n{0}'.format(np.sum(a)))
print('Média dos elementos:\n{0}'.format(np.mean(a)))
print('Soma das linhas:\n{0}'.format(np.sum(a, axis=0)))
print('Média das linhas:\n{0}'.format(np.mean(a, axis=0)))
print('Soma das colunas:\n{0}'.format(np.sum(a, axis=1)))
print('Média das colunas:\n{0}'.format(np.mean(a, axis=1)))

Conteúdo:
[[0 1 2]
 [3 4 5]
 [6 7 8]], shape: (3, 3), Tipo: int32
Soma total dos elementos:
36
Média dos elementos:
4.0
Soma das linhas:
[ 9 12 15]
Média das linhas:
[3. 4. 5.]
Soma das colunas:
[ 3 12 21]
Média das colunas:
[1. 4. 7.]
```

- **Operações binárias**

```
In [29]: import numpy as np
a = np.array([0, 0, 1, 1], dtype=np.bool)
b = np.array([0, 1, 0, 1], dtype=np.bool)
print(a)
print(b)

[False False  True  True]
[False  True False  True]
```

```
In [30]: print(a & b)

[False False False  True]
```

```
In [31]: print(np.logical_and(a, b))

[False False False  True]
```

```
In [32]: print(a | b)

[False  True  True  True]
```

```
In [33]: print(np.logical_or(a, b))

[False  True  True  True]
```

```
In [34]: print(np.logical_not(a))

[ True  True False False]
```

```
In [35]: print(np.logical_xor(a, b))

[False  True  True False]
```

- **Leitura recomendada**

- Documentação do NumPy: <https://docs.scipy.org/doc/numpy/reference/index.html>

Exercícios – Parte 3

Exercício 1

Crie um array 5x5 com a sequência 1...25 (incluso) e faça a soma dos elementos da diagonal.

Exercício 2

Crie um array 5x5 com a sequência de números pares entre 0...50 (incluso) e faça as seguintes operações:

1. Soma das linhas
2. Soma das colunas
3. Média dos elementos da última linha
4. Média dos elementos da última coluna

Exercício 3

Dado o array 2D a seguir, substitua o seu maior valor por 0.

```
array = np.random.normal(0, 3, size=(5, 5))  
print(array)
```

Exercício 4

Dado o par de arrays 2D abaixo, determine se é possível realizar uma multiplicação de matriz, realize a multiplicação caso seja possível.

```
a = np.random.uniform(size=(5, 3))  
b = np.random.uniform(size=(3, 4))
```

Exercício 5

Dado o array abaixo, converta-o para um array do tipo int32.

```
a = np.arange(9, dtype=np.float32).reshape(3, 3)
```

Exercício 6

Dado o array abaixo, some 10 em todos os elementos da última coluna.

```
a = np.random.uniform(size=(8, 8))
```

Exercício 7

Crie um array 2D 4x4 de inteiros que possua a seguinte estrutura:

```
[[ 1  5  9 13]  
 [ 2  6 10 14]  
 [ 3  7 11 15]  
 [ 4  8 12 16]]
```

Exercício 8

Dados os dois arrays 2x4 abaixo (a e b), empilhe os dois formando um novo array 4x4

```
a = np.array([[0,3,4,5], [7, 10, 9, 2]])  
b = np.array([[1,4,9,12], [15, 22, 19, 17]])
```

Exercício 9

Leia o arquivo actors.csv e faça os seguintes cálculos sobre o conjunto de dados utilizando Numpy:

1. O ator/atriz com maior número de filmes e o respectivo número de filmes.
2. A média do número de filmes.
3. O ator/atriz com a maior média por filme.
4. O nome do(s) filme(s) mais frequente(s) e sua respectiva frequência.

Dica: Faça o parsing do arquivo utilizando o pacote [csv](#) do python e depois converta os dados para array e então realize as operações solicitadas. O [numpy não oferece](#) uma maneira de ler csv que possa lidar com vírgulas que estão entre aspas ("). Bem-vindos(as) à ciência de dados :)

NumPy – Algebra linear

Álgebra linear

- NumPy é uma das bibliotecas preferidas para cálculo matricial
- Apesar de existir um tipo [matrix](#), na prática ele não é muito utilizado

```
In [36]: import numpy as np

In [37]: a = np.arange(9).reshape(3,3)
          b = np.arange(10, 19).reshape(3,3)

          print(a)
          print(b)

          [[0 1 2]
           [3 4 5]
           [6 7 8]]
          [[10 11 12]
           [13 14 15]
           [16 17 18]]
```

Multiplicação de matrizes

```
In [38]: np.dot(a,b)

Out[38]: array([[ 45,  48,  51],
                [162, 174, 186],
                [279, 300, 321]])

In [39]: a@b

Out[39]: array([[ 45,  48,  51],
                [162, 174, 186],
                [279, 300, 321]])
```

Diagonal principal

```
In [40]: print(a)

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

In [41]: np.diagonal(a)

Out[41]: array([0, 4, 8])
```

Determinante

```
In [42]: np.linalg.det(a)

Out[42]: 0.0
```

Inverso da matriz

```
In [43]: c = np.array([
        [7, 5],
        [3, 4]
    ])

In [44]: np.linalg.det(c)

Out[44]: 13.0

In [45]: np.linalg.inv(c)

Out[45]: array([[ 0.30769231, -0.38461538],
               [-0.23076923,  0.53846154]])
```

Ordenação

Existem diversos algoritmos para ordenação na biblioteca NumPy:

- Quicksort
- Mergesort
- Heapsort

Cada um desses algoritmos possui características que os tornam mais atrativos, dependendo do caso

Algoritmo	Estável?	Inplace?	Pior tempo de execução
Quicksort	não	sim	$O(N^2)$
Mergesort	sim	não	$O(N \log(N))$
Heapsort	não	sim	$O(N \log(N))$

```
In [46]: np.random.seed(0)
w = np.random.choice(10, size=10, replace=False)
```

```
In [47]: print(w)

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

```
In [48]: np.sort(w)
```

```
Out[48]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In [49]: np.sort(w, kind='mergesort')
```

```
Out[49]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In [50]: np.random.seed(0)
z = np.random.choice(10, size=(2,5), replace=False)
```

```
In [51]: print(z)

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

Ordena baseado em cada coluna

```
In [52]: np.sort(z, axis=0)
```

```
Out[52]: array([[2, 7, 3, 0, 1],
               [6, 8, 4, 9, 5]])
```

Ordena baseado em cada linha

```
In [53]: np.sort(z, axis=1)
```

```
Out[53]: array([[1, 2, 4, 8, 9],
               [0, 3, 5, 6, 7]])
```

Busca por valores

Busca por valores máximos e mínimos

- [argmax](#)
- [argmin](#)
- [max](#)
- [min](#)

```
In [54]: np.random.seed(0)
a = np.random.choice(10, replace=False, size=10)
```

```
In [55]: print(a)

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

```
In [56]: np.max(a)
```

```
Out[56]: 9
```

```
In [57]: np.min(a)
```

```
Out[57]: 0
```

Para recuperar o **índice** do menor valor do array

```
In [58]: print(a)

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

```
In [59]: np.argmax(a)
```

```
Out[59]: 3
```

```
In [60]: a[np.argmax(a)]
```

```
Out[60]: 9
```



Leitura recomendada

- Documentação do NumPy: <https://docs.scipy.org/doc/numpy/reference/index.html>

Matplotlib

Matplotlib é uma biblioteca para visualização de dados

- Permite fazer diversos tipos de projeção
- Solução computacionalmente barata
- Para projeções mais complexas ou visando fazer apresentações, considerar uma biblioteca mais robusta (e.g. [plotly](#))
- Instalar matplotlib via pip

```
C:\WINDOWS\system32>pip install matplotlib
Collecting matplotlib
  Using cached https://files.pythonhosted.org/packages/1a/c0/69e3f695d7384012e90be1e16570c08953baae0fd98094179ef87c7d5a2/matplotlib-3.1.1-cp37-cp37m-win_amd64.whl
Requirement already satisfied: python-dateutil>=2.1 in c:\users\mateus.balen\appdata\roaming\python\python37\site-packages (from matplotlib) (2.8.0)
Collecting kiwisolver>=1.0.1 (from matplotlib)
  Using cached https://files.pythonhosted.org/packages/c6/ea/e5474014a13ab2dcb5956608e0716c600c3d8a8bcff10ed55ccd6a42eb0/kiwisolver-1.1.0-cp37-none-win_amd64.whl
Collecting pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 (from matplotlib)
  Using cached https://files.pythonhosted.org/packages/11/fa/0160cd525c62d7abd076a070ff02b2b94de589f1a9789774f17d7c54058e/pyparsing-2.4.2-py2.py3-none-any.whl
Requirement already satisfied: numpy>=1.11 in c:\program files\python37\lib\site-packages (from matplotlib) (1.17.2)
Collecting cython>=0.10 (from matplotlib)
  Using cached https://files.pythonhosted.org/packages/f7/d2/e07d3ebb2bd7af696440ce7e754c59dd546ffe1bbe732c8ab68b9c834e61/cython-0.10.0-py2.py3-none-any.whl
Requirement already satisfied: six>=1.5 in c:\users\mateus.balen\appdata\roaming\python\python37\site-packages (from python-dateutil>=2.1->matplotlib) (1.11.0)
Requirement already satisfied: setuptools in c:\program files\python37\lib\site-packages (from kiwisolver>=1.0.1->matplotlib) (41.0.1)
Installing collected packages: kiwisolver, pyparsing, cython, matplotlib
Successfully installed cython-0.10.0 kiwisolver-1.1.0 matplotlib-3.1.1 pyparsing-2.4.2
WARNING: You are using pip version 19.2.1, however version 19.2.3 is available.
You should consider upgrading via the 'python -m pip install --upgrade pip' command.
```

Projeções 2D

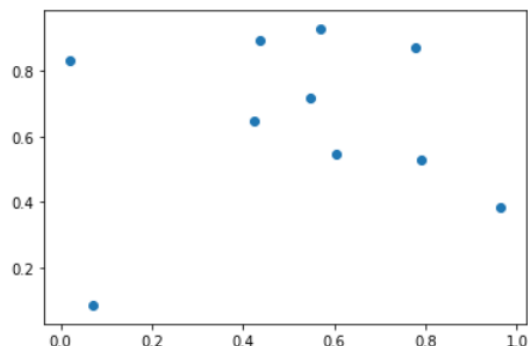
Dispersão

- [matplotlib.pyplot.scatter](#)
- Permite projetar os dados em um plano cartesiano

```
In [2]: %matplotlib inline
import numpy as np
from matplotlib import pyplot as plt
```

```
In [3]: np.random.seed(0)
a = np.random.random((10, 2))

plt.scatter(a[:, 0], a[:, 1])
plt.show()
```



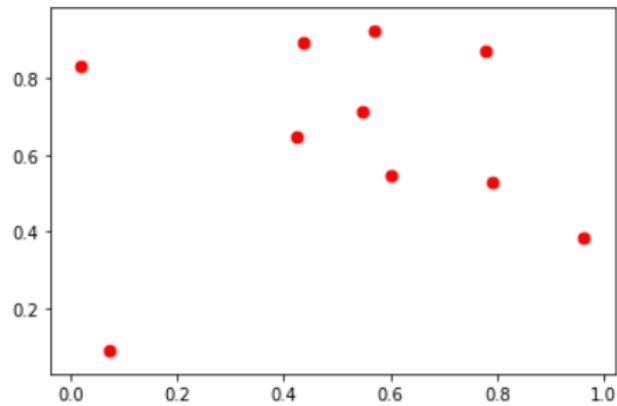
Assim como outras funções da biblioteca matplotlib, existem vários parâmetros que podem ser fornecidos. Os mais comuns são:

- s: Tamanho dos itens

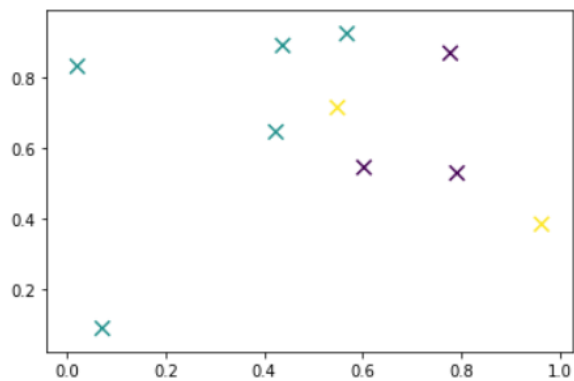
- c: cor dos itens
- marker: estilo do marcador

```
In [4]: np.random.seed(0)  
a = np.random.random((10, 2))
```

```
In [5]: plt.scatter(a[:, 0], a[:, 1], c='red', s=45)  
plt.show()
```



```
In [6]: b = np.random.randint(0, 3, size=10)  
plt.scatter(a[:, 0], a[:, 1], c=b, s=90, marker='x')  
plt.show()
```

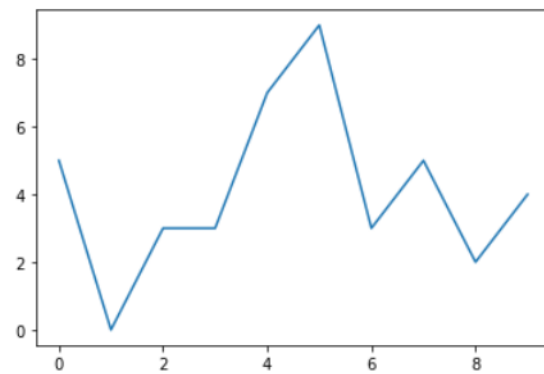


Linhas

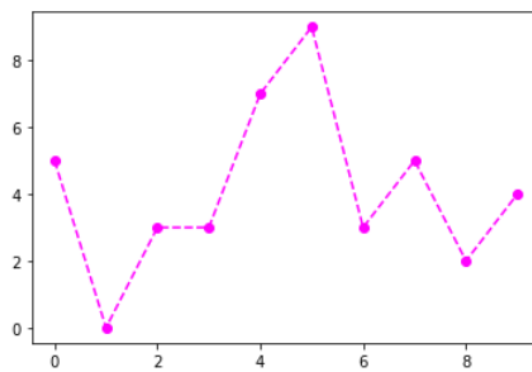
- [matplotlib.pyplot.plot](https://matplotlib.org/3.1.1/api/pyplot_api.html#matplotlib.pyplot.plot)
- Similar a projeção de dispersão, mas conecta os pontos através de uma linha


```
In [7]: np.random.seed(0)
x = np.arange(10)
y = np.random.randint(low=0, high=10, size=10)

plt.plot(x, y)
plt.show()
```



```
In [8]: plt.plot(x, y, marker='o', color='#FF00FF', linestyle='--')
plt.show()
```

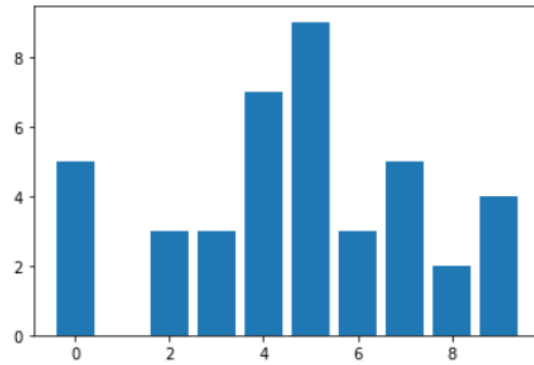


Barras

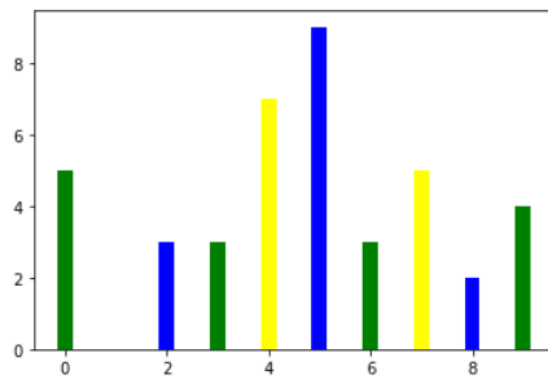
- matplotlib.pyplot.bar

```
In [9]: np.random.seed(0)
x = np.arange(10)
y = np.random.randint(low=0, high=10, size=10)

plt.bar(x, y)
plt.show()
```



```
In [10]: b = np.random.randint(0, 3, size=10)
plt.bar(x=x, height=y, width=0.3, color=['green', 'yellow', 'blue'])
plt.show()
```

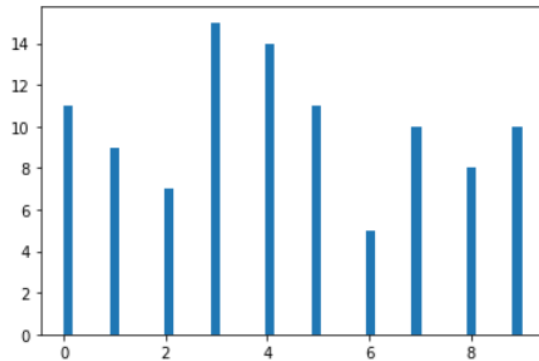


Histograma

- [matplotlib.pyplot.hist](#)
- Uma projeção importante para verificar como a distribuição de valores está se comportando
- Agrupa valores por intervalos
- No eixo x são mostrados os valores que a distribuição toma, e no eixo y o número de amostras que apresentaram aquele valor

```
In [11]: np.random.seed(0)
values = np.random.randint(low=0, high=10, size=100)

plt.hist(values, bins=50)
plt.show()
```



Boxplot

- [matplotlib.pyplot.boxplot](#)
- Outra projeção para verificar a distribuição de valores
- Separa os valores por quartis, desenha um box contendo os 3 primeiros quartis, bem como os limites inferiores e superiores
- Também pode mostrar *outliers*

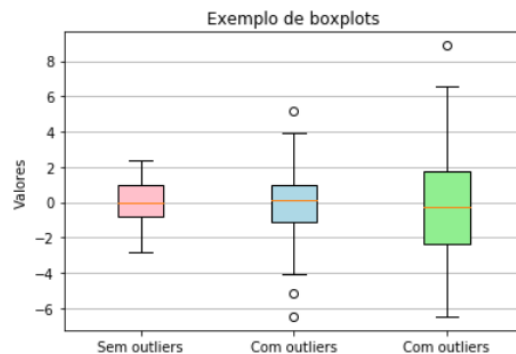
```
In [12]: # gera os dados
np.random.seed(123)
dados = [np.random.normal(0, std, 100) for std in range(1, 4)]
```

```
In [13]: fig = plt.figure()
bplot = plt.boxplot(
    dados,
    vert=True, # alinhamento vertical dos boxplots
    patch_artist=True # preenche os boxplots com cores
)

# pinta boxplots
for patch, color in zip(bplot['boxes'], ['pink', 'lightblue', 'lightgreen']):
    patch.set_facecolor(color)

# customiza eixos
plt.grid(True, axis='y')
plt.xticks(np.arange(3) + 1, ['Sem outliers', 'Com outliers', 'Com outliers'])
plt.title('Exemplo de boxplots')
plt.ylabel('Valores')

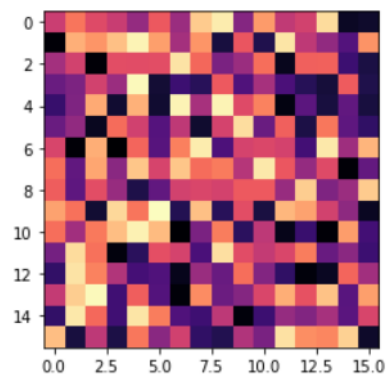
plt.show()
```



Mapa de calor

- matplotlib.pyplot.imshow

```
In [14]: np.random.seed(0)
a = np.random.random((16, 16))
plt.imshow(a, cmap='magma', interpolation='nearest')
plt.show()
```



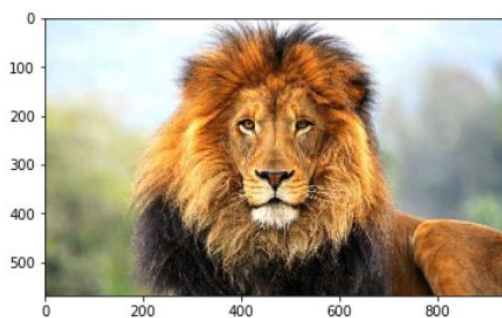
Imagens

- A mesma função utilizada para fazer heatmaps é utilizada para mostrar imagens ([imshow](https://matplotlib.pyplot.imshow))
- Todavia, precisamos carregar a imagem para uma matriz multidimensional

- Copie a imagem “leao.png” para a pasta do Jupyter
- Instale o Pillow

```
C:\WINDOWS\system32>pip install Pillow
Collecting Pillow
  Downloading https://files.pythonhosted.org/packages/14/db/f4c72818297d4d92083c1b3d3404efdb6b2734bea11c0958cfc48b832021/Pillow-6.2.0-cp37-cp37m-win_amd64.whl (2.0MB)
    2.0MB 2.2MB/s
Installing collected packages: Pillow
Successfully installed Pillow-6.2.0
WARNING: You are using pip version 19.2.1, however version 19.2.3 is available.
You should consider upgrading via the 'python -m pip install --upgrade pip' command.
```

```
In [19]: %matplotlib inline
from matplotlib import pyplot as plt
from PIL import Image
img = Image.open('leao.png')
plt.imshow(img)
#plt.axis('off') # desativa eixos
#plt.show()
plt.savefig('myfig.pdf')
```



Exercícios – Parte 4

```
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
```

Exercício 1

Faça o plot da função logística (sigmoid) dada abaixo. Utilize um intervalo $[-5, 5]$ para o eixo X.

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```

Exercício 2

Faça o plot da função logística (sigmoid) e função ReLU dadas abaixo no mesmo gráfico. Utilize cores diferentes para os plotes e mostre as legendas. Utilize um intervalo $[-5, 5]$ para o eixo X.

Documentação para legendas: [Matplotlib Legends](#)

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def relu(x):
    return np.maximum(0, x)
```

Exercício 3

1. Visualize a imagem carregada abaixo utilizando matplotlib.
2. Faça um flip horizontal da imagem e visualize o resultado.
3. Faça um flip vertical da imagem e visualize o resultado.
4. Gire a imagem em 90 graus

```
image = np.array(Image.open('test.jpg'))  
print('Shape da imagem: {}, dtype: {}'.format(image.shape, image.dtype))
```

Exercício 4

Leia o arquivo csv landmarks.csv que contém os pontos faciais (landmarks) para a imagem que é carregada abaixo. Este arquivo é organizado da seguinte maneira:

x, y # para o landmark 1

x, y # para o landmark 2

x, y # para o landmark 3

..., ...

x, y # para o landmark 68

Após ler o arquivo, utilize matplotlib para exibir a imagem bem como os landmarks sobre a face. Experimente mudar os marcadores, bem como seus tamanhos e cores.

```
image = np.array(Image.open('test.jpg'))  
print('Shape da imagem: {}, dtype: {}'.format(image.shape, image.dtype))
```

Exercício 5

Dada a imagem abaixo que contém uma face, utilize o bounding box fornecido para deixar toda a imagem preta exceto o conteúdo do bounding box. Para atribuir a cor preta a um pixel, basta atribuir o valor zero para o pixel específico em todos os canais (dimensão 2) do array (e.g. `image[0, 0, :] = 0`, atribui a cor preta ao pixel na posição `[0, 0]`).

```
image = np.array(Image.open('test.jpg'))  
print('Shape da imagem: {}, dtype: {}'.format(image.shape, image.dtype))  
bbox = [67, 80, 175, 187] # x1, y1, x2, y2  
print('Coordenadas do bounding box: x1={}, y1={}, x2={}, y2={}'.format(*bbox))
```

Exercício 6

Faça um gráfico de barras com o nome dos top 5 atores/atrizes por número de filmes no eixo X e o número de filmes no eixo Y. O índice dos top 5 atores/atrizes é dado no código abaixo.

Experimente mudar o critério de ordenação do top 5 para plotar outras informações.

```
import csv
```

```
# estrutura para guardar os dados
dataset = {'Actor': [],
           'Total Gross': [],
           'Number of Movies': [],
           'Average per Movie': [],
           '#1 Movie': [],
           'Gross': []}

with open('actors.csv', 'r') as f:
    arquivo_csv = csv.reader(f, delimiter=',', quotechar='"')
    for i, row in enumerate(arquivo_csv):
        # pulando o header
        if i == 0:
            continue
        # parsing
        dataset['Actor'].append(row[0])
        dataset['Total Gross'].append(float(row[1]))
        dataset['Number of Movies'].append(int(row[2]))
        dataset['Average per Movie'].append(float(row[3]))
        dataset['#1 Movie'].append(row[4])
        dataset['Gross'].append(float(row[5]))

# convertendo tudo para np arrays
for k in dataset.keys():
    dataset[k] = np.array(dataset[k])

criterio = 'Number of Movies'
indice_top_5 = np.argsort(dataset[criterio])[::-1][:5]
```

Referências

Básica

1. Zed A. Shaw. Learn Python 3 the Hard Way: A Very Simple Introduction to the Terrifyingly Beautiful World of Computers and Code. Addison Wesley, 2017. 320p.
2. Mark Lutz. Learning Python. O'Reilly, 2013. 1540p.
3. Wes Mckinney. Python for Data Analysis (2nd edition). O'Reilly, 2017. 522p.

Complementar

1. WICKHAM, H.; GROLEMUND, G. R for Data Science. 1st ed., O'Reilly, 2017.
2. FACELI, K., LORENA, A.C., GAMA, J., CARVALHO, A.C.P.L.F. Inteligência Artificial: Uma Abordagem de Aprendizado de Máquina. Rio de Janeiro: LTC, 2011. 378 p.
3. SIMON, PI. The visual organization : data visualization, big data, and the quest for better decisions. Hoboken : Wiley, c2014. 202 p.
4. GOODFELLOW, I., BENGIO, Y., COURVILLE, A.. Deep Learning. MIT Press, 775p., 2016.
5. Matthew O. Ward, Georges Grinstein, Daniel Keim, Interactive Data Visualization: Foundations, Techniques, and Applications. A K Peters/CRC Press, 2010. 513 p.

