Tutorial de Python

Recursos

Alguns recursos úteis:

- Sítio official - https://www.python.org - Para downloads, documentação completa, ...
- Tutorial de Python 3 - https://docs.python.org/3/tutorial/ -
- Standard Library - https://docs.python.org/3/library/ -
- Language Reference - https://docs.python.org/3/reference/ -
- Style Guide - https://www.python.org/dev/peps/pep-0008/ - convenções a usar na escrita do código Python.
- A Wiki de Python - https://wiki.python.org/moin/
- Python Tutor - http://pythontutor.com/ - Execução online de programas com possibilidade de visualização da evolução do estado do programa ("live programming mode")
- Tutorialspoint – Documentação sintética do essencial para começar a programar em Python (https://www.tutorialspoint.com/python3/)
- Python Course - https://www.python-course.eu/python3_course.php -
- Think Python: How to Think Like a Computer Scientist, 2nd edition (PDF, HTML)

  (https://greenteapress.com/thinkpython2/thinkpython2.pdf)

- Wiki Python: https://wiki.python.org/moin/PythonBooks

Ambiente de desenvolvimento de Python

Existem inúmeros ambientes de desenvolvimento que suportam Python:

- **IDLE** - fornecido com o pacote standard oficial. Leve, com algumas limitações;
- **Pycharm** – versão community e edu;
- **Anaconda** - plataforma que inclui um IDE mais artilhado e muitas bibliotecas não standard do Python;
- **Visual Studio Code**....

Vamos assumir o IDLE como ambiente de desenvolvimento.

Ao abrir a respetiva aplicação, notem que se pretende usar o Python 3 (e não o Python 2, também disponível). Verificar, portanto, se a versão é a pretendida. Deverá aparecer, na janela "Python Shell", algo como:

```
Python 3.7.7 (default, May 6 2020, 11:45:54) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
```

Python é uma linguagem interpretada. O *prompt* do interpretador é o símbolo '>>> '. Assim que este aparece, poderão ser dadas instruções que serão imediatamente executadas. Poderão também ser avaliadas expressões.

```
>>> x = 10
>>> print(x*2)
20
>>> x*4
40
```

A forma mais comum de trabalhar consiste em ter abertas duas janelas:

- Uma com a *Shell*, aberta automaticamente quando se arranca o IDLE (ilustrada no exemplo acima)
- Outra para editar o ficheiro que vai conter o programa (aberta no menu *File -> New File*)

Operadores

O interpretador de Python pode ser usado para avaliar expressões, como por exemplo, expressões aritméticas simples. Se escrever essas expressões na prompt, elas serão avaliadas e o resultado será devolvido na linha seguinte.

```
>>>1 + 1
2
>>>2 * 3
6
```

Existem também em Python operadores booleanos que manipulam valores True e False.

```
>>>True
True
```

```
>>>False
False
```

```
>>>1 == 0
False

>>>not (1==0)
True
```

```
>>>(2==4-2) and (2==3)
False
```

:

```
>>>(2==4-2) or (2==3)
True
```

Strings

O Python tem um tipo String predefinido. Podemos usar ' ' ou " " para envolver as cadeias de caracteres.

"artificial"

'artificial'

O operador quando tem string como operandos, executa uma concatenação.

```
>>>'artificial' + " " + "intelligence"
'artificial intelligence'
```

Existem muitos métodos pré-definidos para manipular strings.

```
>>>'artificial'.upper()
'ARTIFICIAL'
```

```
>>>'HELP'.lower()
'help'
```

```
>>>len('Help')
4
```

Podemos guardar strings em variáveis, que não têm de ser declaradas antes de serem usadas

```
>>>s = 'olá mundo'
```

Podemos imprimir a variável no ecrã.

```
>>>print(s)
olá mundo
```

```
>>>s.upper()
'OLÁ MUNDO'
```

```
>>>len(s)
9
```

Dir e Help

A função ***dir()*** permite saber quais os métodos associados a um certo objecto, neste caso uma string.

```
>>>dir(s)
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__',
'__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'capitalize', 'casefold', 'center', 'count',
'encode', 'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal', 'isdigit',
'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition',
'removeprefix', 'removesuffix', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip',
'swapcase', 'title', 'translate', 'upper', 'zfill']
```

help permite saber alguma coisa sobre um método específico

```
>>>help(s.split)
Help on built-in function split:

split(sep=None, maxsplit=-1) method of builtins.str instance
    Return a list of the substrings in the string, using sep as the separator string.

      sep
        The separator used to split the string.

        When set to None (the default value), will split on any whitespace
        character (including \\n \\r \\t \\f and spaces) and will discard
        empty strings from the result.
      maxsplit
        Maximum number of splits (starting from the left).
        -1 (the default value) means no limit.

    Note, str.split() is mainly useful for data that has been intentionally
    delimited.  With natural text that includes punctuation, consider using
    the regular expression module.
```

Vamos usá-lo

```
>>>''a-maria-e-o-manel''.split('-')
```

```
['a', 'maria', 'e', 'o', 'manel']
```

Exercício: Tentar usar algumas das funções sobre strings listadas quando se fez dir (ignorar as que possuem '__').

Estruturas de dados pré-definidas

O Python vem equipado com algumas estruturas de dados "built-in", semelhantes ao pacote collections do Java.

Listas

As listas guardam sequências de elementos mutáveis. Por exemplo, eis uma lista de 4 frutas:

```
>>>frutas = ['maçã','laranja','pêra','banana']
```

Para acedermos ao primeiro elemento da lista, fazemos:

```
>>>frutas[0]
'maçã'
```

Podemos usar o operador '+' para concatenar listas

```
>>>outrasFrutas = ['kivi','morangos']
>>>frutas + outrasFrutas
['maçã', 'laranja', 'pêra', 'banana', 'kivi', 'morangos']
```

O Python permite a indexação negativa para aceder aos elementos pela ordem de trás para a frente. Por exemplo, com frutas[-1] acede-se ao último elemento.

```
>>>frutas[-1]
'banana'
>>>frutas[-2]
'pêra'
```

Podemos obter o último elemento de uma lista depois de ser removido

```
>>>frutas.pop()
'banana'
>>>frutas
['maçã', 'laranja', 'pêra']
```

Podemos mudar o último elemento da lista para anánas fazendo

```
>>>frutas[-1] = 'ananás'
>>>frutas
['maçã', 'laranja', 'ananás']
```

Podemos colocar no fim da lista de frutas, mais uma fruta, uvas

```
>>>frutas.append('uvas')
>>>frutas
['maçã', 'laranja', 'ananás', 'uvas']
```

Podemos aceder a sublistas de uma lista através do operador de slice. Por exemplo, fruits[1:3] devolve a lista contendo os elementos da posição 1 à 2. Geralmente frutas[inicio:fim] devolverá a sublista com elementos desde a posição inicio até fim-1. Também poderemos fazer frutas[inicio:] que devolve uma sublista com os elementos que vão da posição inicio até ao fim da lista. Da mesma maneira, frutas[:fim] devolverá todos os elementos até à posição fim-1:

```
>>>frutas[0:2]
['maçã', 'laranja']
>>>frutas[:3]
['maçã', 'laranja', 'ananás']
>>>frutas[2:]
['ananás', 'uvas']
>>>len(frutas)
4
```

Os elementos de uma lista podem ser de qualquer tipo

```
>>>listaDeListas = [['a','b','c'],[1,2,3],['um','dois','três']]
>>>listaDeListas[0].pop()
'c'
>>>listaDeListas
[['a', 'b'], [1, 2, 3], ['um', 'dois', 'três']]
```

Exercício (Listas): Teste as funções sobre listas. Podem encontrar os métodos usando o dir e help, mas ignorem os métodos com ''underscores''. Por exemplo,

```
>>dir(list)
['__add__', '__class__', '__class_getitem__', '__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']
```

```
>>><help(list.reverse)
Help on method_descriptor:

reverse(self, /)
    Reverse *IN PLACE*.
```

```
>>>lista= ['a','b','b']
>>>lista.reverse()
>>>lista
['b', 'b', 'a']
```

Tuplos

Uma estrutura de dados semelhante a uma lista é o tuplo, que é tudo igual a uma lista mas que é imutável a partir do momento em que é criado, i.e., não pode mudar. Notem que os tuplos são envolvidos em parêntesis enquanto as listas são-no por parêntesis rectos.

```
>>>par = (3,5)
>>>par[0]
3
```

Na verdade, nem precisamos de parêntesis.

```
>>>outro_par = 5,5
>>>outro_par
(5, 5)
```

Podemos desempacotar um tuplo da seguinte maneira:

```
>>>x,y = par
>>>x
3
>>>y
5
```

Podemos testar a imutabilidade do seu conteúdo

```
>>>par[1]=4
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Conjuntos

Um conjunto é outra estrutura de dados que é uma lista não ordenada sem elementos duplicados. Não se assume que a ordem com que são impressos no ecrã seja a mesma para todas as máquinas. Mostramos já como se cria um conjunto, criando uma lista e convertendo-a em conjunto:

```
>>>formas = ['círculo','quadrado','triângulo','círculo']
>>>formas = set(formas)
>>>formas
{'triângulo', 'quadrado', 'círculo'}
```

Mas podemos criar um conjunto de um modo mais directo

```
>>>outras_formas = {'quadrado', 'triângulo','círculo'}
>>>outras_formas
{'triângulo', 'quadrado', 'círculo'}
```

Na criação os elementos repetidos desaparecem

```
>>>outras_formas = {'quadrado', 'triângulo','quadrado','círculo'}
>>>outras_formas
{'triângulo', 'quadrado', 'círculo'}
```

Vamos comparar conjuntos para verificar que não dependem da ordem com que foram criados os seus elementos.

```
>>>formas == outras_formas
True
```

A seguir, iremos mostrar como adicionar elementos a um conjunto, como verificar se um elemento pertence a um conjunto e vamos também executar algumas operações comuns sobre conjuntos (diferença, intersecção, união):

```
>>>formas = set()
>>>formas
set()
```

```
>>>formas.add('polígono')

>>>formas
{'polígono'}
```

```
>>>'círculo' in formas
```

```
False
```

```
>>>formas.add('círculo')
>>>'círculo' in formas
True
```

```
>>>formas & outras_formas
{'círculo'}
```

```
>>>formas | outras_formas
{'quadrado', 'polígono', 'círculo', 'triângulo'}
```

```
>>>outras_formas – formas
{'triângulo', 'quadrado'}
```

Dicionários

A última estrutura de dados que vamos apresentar é o dicionário, que corresponde a uma tabela, mapeia um tipo de objectos (a chave) noutro (o valor). A chave tem de ser de um tipo imutável (string, número ou tuplo). O valor pode ser de qualquer tipo. Um dicionário é uma tabela hash em que não existe uma ordem das chaves.

```
>>>estudantes = {19777: 'Pedro', 20200: 'Liza', 21999: 'Zanga'}
>>>estudantes[19777]
'Pedro'
```

```
>>>del estudantes[20200]
>>>estudantes
{19777: 'Pedro', 21999: 'Zanga'}
```

```
>>>del estudantes[1000]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 1000
```

```
>>>estudantes[20202]='Chico'
>>>estudantes
{19777: 'Pedro', 21999: 'Zanga', 20202: 'Chico'}
```

```
>>>estudantes.items()
dict_items([(19777, 'Pedro'), (21999, 'Zanga'), (20202, 'Chico')])
```

```
>>>estudantes.keys()
dict_keys([19777, 21999, 20202])
```

```
>>>list(estudantes.keys())
[19777, 21999, 20202]
```

```
>>>estudantes.values()
dict_values(['Pedro', 'Zanga', 'Chico'])
```

```
>>>list(estudantes.values())
['Pedro', 'Zanga', 'Chico']
```

Podemos criar dicionários de dicionários

```
>>>professores = {'iia': 'LuigiLuis','ec': 'Lou'}
>>>staff = {'professores': professores, 'estudantes':estudantes}

>>>staff
{'professores': {'iia': 'LuigiLuis', 'ec': 'Lou'}, 'estudantes': {19777: 'Pedro', 21999: 'Zanga', 20202: 'Chico'}}
>>>staff['professores']
{'iia': 'LuigiLuis', 'ec': 'Lou'}
```

Exercício: Usar dir e help para se familiarizarem com as funções que se podem invocar sobre dicionários.

Ciclo for e if then else

Vamos fazer um pequeno programa que exemplifica o uso do ciclo for e da instrução condicional:

```
# Este é um comentário
frutas = ['maçãs', 'laranjas', 'pèras', 'bananas']
for fruta in frutas:
    print(fruta + ' para venda')

precosFruta = {'maçãs': 2.00, 'laranjas': 1.50, 'pêras': 1.75}
for fruta, preco in precosFruta.items():
    if preco < 2.00:
        print('As %s custam %f o kg' % (fruta, preco))
    else:
        print(''As '' + fruta + ' são demasiado caras!')
```

Usem o range para gerarem uma sequência de inteiros,

```
>>>range(10)
range(0, 10)
```

```
>>>list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

que é útil para os ciclos:

```
for index in range(3):
    print(index)
ferramentas = ['enxada', 'ancinho', 'pá']
for index in range(len(ferramentas)):
    print(ferramentas[index])
```

Map e Filter

Podem usar as funções **map** e **filter**.

Vamos aplicar a função quadrado (anónima, lambda) a uma lista de inteiros

```
>>>list(map(lambda x: x * x, [1,2,3]))
[1, 4, 9]
```

Vamos filtrar de uma lista todos os elementos menores do que 4

```
>>>list(filter(lambda x: x > 3, [1,2,3,4,5,4,3,2,1]))
[4, 5, 4]
```

Listas por compreensão

Se quisermos gerar a lista que resulta de adicionar um a cada elemento de uma lista de input fazemos:

```
>>>nums = [1,2,3,4,5,6]
>>>plusOneNums = [x+1 for x in nums]
>>>plusOneNums
[2, 3, 4, 5, 6, 7]
```

Se quisermos apenas os números ímpares podemos fazer:

```
>>>oddNums = [x for x in nums if x % 2 == 1]
>>>oddNums
[1, 3, 5]
```

Cuidado com a identação!

Ao contrário das restantes linguagens o Python usa a identação para interpretar o código.

```
if 0 == 1:
    print('Estamos num mundo de sofrimento matemático')
print('Obrigado por jogar')
```

Mas se fizermos assim,

```
if 0 == 1:
    print('Estamos num mundo de sofrimento matemático')
    print('Obrigado por jogar')
```

não haverá qualquer output.

Funções

Podem definir as vossas próprias funções

```
>>>precosFruta = {'maçãs':2.00, 'laranjas': 1.50, 'pêras': 1.75}
```

```
def comprarFruta(fruta, numKgs):
    if fruta not in precosFruta:
        print("Lamento mas não temos %s" % (fruta))
    else:
        custo = precosFruta[fruta] * numKgs
        print("Serão %f euros, por favor" % (custo))
```

```
>>>comprarFruta('maçãs',2.4)
Serão 4.800000 euros, por favor
```

```
>>comprarFruta('côcos',2)
Lamento mas não temos côcos
```

Definindo classes

Vejam este exemplo de uma definição de classe para uma loja de fruta em que a classe tem dois atributos, o nome da loja e os preços por kg de cada tipo de fruta, e fornece métodos para saber o preço por kg dado um tipo de fruta e o preço de uma lista de compras.

```
class LojaFrutas:

    def __init__(self, nome, precosFruta):
                #nome: Nome da loja de fruta
                #precosFruta: Dicionário com a fruta como chave e os preços como
                #valor. Eis um exemplo:
               #      {'maçãs':2.00, 'laranjas': 1.50, 'pêras': 1.75
        self.precosFruta = precosFruta
        self.nome = nome
        print('Bem vindo à loja de fruta %s' % (nome))
```

```
    def getCustoPorKg(self, fruta):
                #  fruta: a string que designa a fruta
                        #  Devolve o custo da fruta, assumindo que a fruta está oinventário, #senão devolve None

        if fruta not in self.precosFruta:
            return None
        return self.precosFruta[fruta]

    def getPrecoCompras(self,listaCompras):
                #  listaCompras: Lista de tuplos do tipo (fruta, numKgs)
                 #Devolve o custo da lista de compras, apenas incluindo os pedidos de   #fruta que exista na loja.
         custoTotal = 0.0
        for fruta, numKgs in listaCompras:
            custoPorKg = self.getCustoPorKg(fruta)
            if custoPorKg != None:
                custoTotal += numKgs * custoPorKg
        return custoTotal

    def getNome(self):
        return self.nome
```

Vamos criar uma instância da classe, um objecto:

```
>>>nomeLoja = 'Pomar de Zizu'
>>precosFruta = {'maçãs': 1.00, 'laranjas': 1.50, 'pêras': 1.75}
>>>lojaZizu = LojaFrutas(nomeLoja, precosFruta)

Bem vindo à loja de fruta Pomar de Zizu
```

A linha lojaZizu = lojaFrutas(nomeLoja, precosFruta) constrói uma instância da classe LojaFruta, chamando a função **init** dessa classe. Notem que passamos apenas 2 argumentos no construtor, enquanto **init** tem 3 argumentos: (self, nome, precosFruta). A razão para isto é que todos os métodos de uma classe têm de ter self como primeiro argumento. O valor do argumento self é atribuído automaticamente ao próprio objecto; ao chamar um método, fornece-se apenas os argumentos restantes. A variável self contém toda a informação (nome e precosFruta) para a instância atual (semelhante em Java). As instruções de print usam o operador de substituição (descrito na documentação de Python).

Se quisermos aceder ao preço por kg das pêras, fazemos

```
>>>precoPera=lojaZizu.getCustoPorKg('pêras')
>>>print('As pêras custam %.2f euros no %s.' % (precoPera, lojaZizu.nome))

As pêras custam 1.75 euros no Pomar de Zizu.
```

Se quisermos saber o valor dos abacates receberemos None

```
>>>print(lojaZizu.getCustoPorKg('abacates'))

None
```

Vamos criar mais um objecto da mesma classe, a loja Chère em que os preços são sempre o dobro dos da loja do Zizu

```
>>>lojaZizu.precosFruta.keys()
```

```
dict_keys(['maçãs', 'laranjas', 'pêras'])


precosChere = {}
for fruta in lojaZizu.precosFruta.keys():
    precosChere[fruta] = lojaZizu.precosFruta[fruta] * 2
lojaChere = LojaFrutas('Chère', precosChere)
print('Preços da Chère:',precosChere)
```

```
Bem vindo à loja de fruta Pomar de Zizu

Bem vindo à loja de fruta Chère

Preços da Chère: {'maçãs': 2.0, 'laranjas': 3.0, 'pêras': 3.5}
```

Variáveis estáticas vs variáveis dinâmicas

O exemplo seguinte ilustra como usar as variáveis estáticas e as de instância em Python

Vamos criar a classe Pessoa que tem uma variável estática, **populacao**, partilhada por todas as instâncias da classe e uma de instância, **idade**. Cada vez que criamos uma nova pessoa a população é incrementada.

```
class Pessoa:
    populacao = 0
    def __init__(self, minhaldade):
        self.idade = minhaldade
        Pessoa.populacao += 1
    def get_populacao(self):
        return Pessoa.populacao
    def get_idade(self):
        return self.idade
```

```
>>>p1=Pessoa(12)

>>>print('População = ',p1.populacao)

População =  1
```

```
>>>p2 = Pessoa(62)

>>>print('População = ',p2.populacao)
População =  2
```

```
>>>print('População = ',p1.populacao)

População =  2
```

Como puderam ver, a variável **populacao** é partilhada por todos os objectos da classe, enquanto a variável **idade** é privada.

```
>>>p1.get_idade()
```

```
12
```

```
>>>p2.get_idade()
62
```

Se quisermos imprimir um dos objectos obteremos:

```
>>>print(p1)
<__main__.Pessoa object at 0x0000023BDEF7B3D0>
```

Se quisermos imprimir apenas a informação relevante de cada objecto, teremos de redefinir o método **str** e voltar a criar as instâncias.

```
class Pessoa:
    populacao = 0
    def __init__(self, minhaIdade):
        self.idade = minhaIdade
        Pessoa.populacao += 1
    def get_populacao(self):
        return Pessoa.populacao
    def get_idade(self):
        return self.idade
    def __str__(self):
        return str(self.idade)
```

```
>>>p1=Pessoa(12)
>>>print('População = ',p1.populacao)
População =  1
```

```
>>>p2 = Pessoa(62)
>>>print('População = ',p2.populacao)
População =  2
```

```
>>>print(p1)
12
```

Conjunto de objectos

Vamos criar um conjunto de pessoas, e guardaremos o conjunto de idades. Queremos apenas um objecto pessoa para cada idade. Comecemos pelo conjunto das 2 pessoas que já criámos.

```
>>>pessoas = {p1,p2}
>>>print(pessoas)
{<__main__.Pessoa object at 0x0000020A6452B490>, <__main__.Pessoa object at 0x0000020A6452B130>}
```

```
>>>print(p1)
12
```

```
>>>p3 = Pessoa(12)
>>>print(p3)
12
```

```
>>>p1.populacao
3
```

Como p3 é diferente de p1, embora o conteúdo seja igual, é possível adicioná-lo ao conjunto.

```
>>>pessoas.add(p3)
>>>pessoas
{<__main__.Pessoa object at 0x000001397BCBBCA0>, <__main__.Pessoa object at 0x000001397BCBBA60>,
<__main__.Pessoa object at 0x000001397BCBB370>}
```

Por defeito, dois objectos só são considerados iguais se forem o mesmo objecto, o que quer dizer que teremos que redefinir o conceito de ==. No entanto, para os conjuntos é preciso ter elementos que são hashable, o que é o caso das strings, números e objectos.

```
>>print(hash(4))
4
```

```
>>>print(hash("morangos"))
-2684595218499097407
```

```
>>>print(hash((1,2,3)))
529344067295497451
```

```
>>>print(hash(p1.idade))
12
```

```
>>>print(hash(p1))
84150107082
```

Mas cada objecto tem um hash() diferente dos outros mesmo que os dados dos objectos sejam os mesmos, caso de p1 e de p3.

```
>>>print(hash(p3))
84150106935
```

Assim, teremos que sobrepor os métodos **eq** e **hash()**. Dois objectos Pessoa serão iguais se tiverem o mesmo valores nos atributos respectivos idade, e a função de hash do objecto passa a ser o resultado da invocação da função standard

hash tendo como input o valor do atributo idade. Assim, dois objectos distintos mas com o mesmo conteúdo serão iguais e terão o mesmo hash().

```
class Pessoa:
    populacao = 0
    def __init__(self, minhaIdade):
        self.idade = minhaIdade
        Pessoa.populacao += 1
    def get_populacao(self):
        return Pessoa.populacao
    def get_idade(self):
        return self.idade
    def __str__(self):
        return str(self.idade)
    def __eq__(self,other):
        return self.idade == other.idade
    def __hash__(self):
        return hash(self.idade)
```

```
>>>p1=Pessoa(12)
>>>print('População = ',p1.populacao)
População =  1
```

```
>>>p2 = Pessoa(62)
>>>print('População = ',p2.populacao)
População =  2
```

```
>>>p3 = Pessoa(12)
>>>print('População = ',p3.populacao)
População =  3
```

```
>>>pessoas = {p1,p2,p3}
>>>print(pessoas)
{<__main__.Pessoa object at 0x0000020A6452BCD0>, <__main__.Pessoa object at 0x0000020A6452BA00>}
```

```
>>>[str(pessoa) for pessoa in pessoas]
['12', '62']
```

Tutorial de Pyhton baseado em Luís Correia, Inteligência Artificial, FCUL