Atributos de objeto e de classe

- Os atributos são variáveis que se encontram “confinadas” ao 'espaço de nomes' de uma classe ou de um objeto
  - *atributos de classe são compartilhados por todos os objetos da classe – há apenas uma cópia dessa variável*
  - *atributos de objeto são particulares a cada objeto individual de uma classe – cada objeto possui a sua própria cópia desse atributo.*
- Também os métodos poderão estar associados a uma classe a um objeto
  - *Veja-se o exemplo que se segue…*

Membros de objeto e de classe – exemplo

- population pertence à classe Robot e, portanto, é um atributo de classe.
- Já o atributo name pertence ao objeto, dado ser definido com o qualificador self
- Os dois atributos passam a ser referenciados com qualificadores diferentes: self.name e Robot.population.
- howMany() é um método de classe, uma vez que tem como 1º argumento 'cls' (abrev. de classe) e é precedido pelo decorador @classmethod

(para mais detalhes consultar [4])

```
class Robot:
    """Represents a robot, with a name."""

    # A class variable, counting the number of robots
    population = 0

    def __init__(self, name):
        """Initializes the data."""
        self.name = name
        print("(Initializing {})".format(self.name))
        # When this person is created, the robot
        # adds to the population
        Robot.population += 1

    def die(self):
        """I am dying."""
        print("{} is being destroyed!".format(self.name))
        Robot.population -= 1
        if Robot.population == 0:
            print("{} was the last one.".format(self.name))
        else:
            print("There are still {:d} robots working.".format(
                Robot.population))

    def say_hi(self):
        print("Greetings, my masters call me {}.".format(self.name))

    @classmethod
    def how_many(cls):
        print("We have {:d} robots.".format(cls.population))
```

Um **decorador** é uma função especial que é usada para adicionar funcionalidades a outra função, andes e depois da sua execução, sem a alterar diretamente.

Teste

```
droid1 = Robot("R2-D2")
droid1.say_hi()
Robot.how_many()

droid2 = Robot("C-3PO")
droid2.say_hi()
Robot.how_many()

print("\nRobots can do some work here.\n")

print("Robots have finished their work. So let's destroy them.")
droid1.die()
droid2.die()

Robot.how_many()
```

- Viu-se, neste exemplo, a utilização de docstrings, quer nos métodos, quer na própria classe
  - *podemos depois aceder à documentação da classe com* Robot.__doc__
  - *e aceder à documentação do método com* Robot.sayHi.__doc__
  - ou aceder à documentação de ambas estruturas via função help(.)

Em Python, **método de classe** e método estático não são bem a mesma coisa:

- um método de classe tem o parâmetro cls
  - *que referencia a própria classe*
  - *podendo, por isso, aceder a outros membros da classe*
  - *e é precedido por @classmethod*
- já um **método estático**, não tem nem o parâmetro cls nem o self (não tendo assim acesso nem à referência da classe nem do objeto)
  - *será apenas uma função utilitária que opera unicamente sobre os seus próprios parâmetros (mas que, ainda assim, terá alguma relação lógica com a classe)*
  - *e não terá necessidade de lidar nem com a classe nem com objetos da classe*
  - *é precedido por @staticmethod*

Output:

```
(Initializing R2-D2)
Greetings, my masters call me R2-D2.
We have 1 robots.
(Initializing C-3PO)
Greetings, my masters call me C-3PO.
We have 2 robots.

Robots can do some work here.

Robots have finished their work. So let's destroy them.
R2-D2 is being destroyed!
There are still 1 robots working.
C-3PO is being destroyed!
C-3PO was the last one.
We have 0 robots.
```

Herança em Python

- Em Python, a classe object é a 'mãe' de todas as classes
  - *À semelhança do Java e do C#, todas as classes derivam da classe object, de forma direta (quando não derivam explicitamente de outra) ou através das suas superclasses,*
  - *garantindo-se dessa forma algumas funcionalidades básicas em qualquer classe que seja criada*
- A Herança múltipla é permitida entre classes
- Para se criar uma classe por derivação de outra, coloca-se à frente do nome da classe que está a ser criada o nome da classe base, entre parêntesis curvos
  - `class SubClasse(ClasseBase): ...`
- O construtor duma subclasse, quando definido, deve invocar o construtor da classe base
  - *Caso não se defina na subclasse qualquer construtor, então, e só nessa situação, pode-se considerar que o construtor da classe base é herdado*
- A partir duma subclasse, há duas formas de invocar o construtor da sua classe base:
  - `super().__init__([argumentos])`
  - `ClasseBase.__init__(self, [argumentos])`
- Como, por norma, cada classe é definida num ficheiro separado, a definição duma subclasse deve ser precedida pela importação da respetiva classe base

```
from moduloDaClasseBase import ClasseBase
class SubClasse(ClasseBase): ...
```

Herança/Polimorfismo

```
class SchoolMember:
    '''Represents any school member.'''
    def __init__(self, name, age):
        self.name = name
        self.age = age
        print('(Initialized SchoolMember: {})'.format(self.name))
    def tell(self):
        '''Tell my details.'''
        print('Name:"{}" Age:"{}"'.format(self.name, self.age), end=" ")

class Teacher(SchoolMember):
    '''Represents a teacher.'''
    def __init__(self, name, age, salary):
        SchoolMember.__init__(self, name, age)#super().__init__(name, age)
        self.salary = salary
        print('(Initialized Teacher: {})'.format(self.name))
    def tell(self):
        SchoolMember.tell(self) # or super().tell()
        print('Salary: "{:d}"'.format(self.salary))

class Student(SchoolMember):
    '''Represents a student.'''
    def __init__(self, name, age, marks):
        SchoolMember.__init__(self, name, age)#super().__init__(name, age)
        self.marks = marks
        print('(Initialized Student: {})'.format(self.name))
    def tell(self):
        SchoolMember.tell(self) # or super().tell()
        print('Marks: "{:d}"'.format(self.marks))
```

Em Python,

- *todos os métodos são virtuais*
- *e uma vez que aquilo que temos são sempre referências para os objetos,*

o polimorfismo é implícito...

```
t = Teacher('Mrs. Shrividya', 40, 30000)
s = Student('Swaroop', 25, 75)

# prints a blank line
print()

members = [t, s]
for member in members:
    # Works for both Teachers and Students
    member.tell()
```

Output:

```
(Initialized SchoolMember: Mrs. Shrividya)
(Initialized Teacher: Mrs. Shrividya)
(Initialized SchoolMember: Swaroop)
(Initialized Student: Swaroop)

Name:"Mrs. Shrividya" Age:"40" Salary: "30000"
Name:"Swaroop" Age:"25" Marks: "75"
```

Ficheiros

```
poem = '''\
Programming is fun
When the work is done
if you wanna make your work also fun:
    use Python!
'''
# Open for 'w'riting
f = open('poem.txt', 'w')
f.write(poem) # Write text to file
f.close()  # Close the file
# If no mode is specified,
# 'r'ead mode is assumed by default
f = open('poem.txt')
while True:
    line = f.readline()
    # Zero length indicates EOF
    if len(line) == 0:
        break
    # The `line` already has a newline
    # at the end of each line
    # since it is reading from a file.
    print(line, end='')
f.close() # close the file
```

- É possível abrir ficheiros e escrever e ler neles, criando um objeto da classe file, através da função open(), e usando os métodos read(), readline() e write()
  - *No fim termina-se com o método close()*
  - *No modo de abertura podemos especificar se vamos ler ('r'), escrever ('w') ou acrescentar ('a'), em modo de texto ('t') ou em modo binário ('b'), entre outras opções*
    - por defeito, o ficheiro é considerado de texto e é aberto em modo de leitura

Output:

```
Programming is fun
When the work is done
if you wanna make your work also fun:
    use Python!
```

Tornar os objetos persistentes

- O Python fornece-nos uma forma muito simples de gravar um objeto num ficheiro e recuperá-lo mais tarde – a forma de o tornar persistente, portanto.
  - *Basta usar o módulo standard chamado pickle, como ilustrado no exemplo.*
  - *Para além do objeto principal, o pickle, atuando recursivamente, envia também para o ficheiro todos os seus subobjectos (objetos referenciados pelos atributos do principal)*

```
import pickle

# The name of the file where we will store the object
shoplistfile = 'shoplist.data'
# The list of things to buy
shoplist = ['apple', 'mango', 'carrot']

# Write to the file
f = open(shoplistfile, 'wb')
# Dump the object to a file
pickle.dump(shoplist, f)
f.close()

# Destroy the shoplist variable
del shoplist

# Read back from the storage
f = open(shoplistfile, 'rb')
# Load the object from the file
storedlist = pickle.load(f)
print(storedlist)
f.close()
```

Output:

```
['apple', 'mango', 'carrot']
```

Biblioteca Standard do Python

- A Biblioteca Standard do Python contém um vasto número de módulos úteis e acompanha todas as instalações do Python.
  - *É importante sabê-la utilizar, uma vez que com ela muitos problemas podem ser resolvidos rapidamente.*
- Vamos ilustrar apenas um dos módulos mais usados da biblioteca (módulo sys), mas uma grande diversiade de outros módulos poderão ser encontrados e usados,
  - *como será o caso, por exemplo, das expressões regulares (re), das opções de linha de comando (argparse), dos tipos data/hora (datetime) e debugging (pdb).*
- A documentação detalhada de todos os módulos da Biblioteca Standard do Python pode ser encontrada no seguinte endereço:

  https://docs.python.org/3/library/

O módulo sys

- O módulo sys contém funcionalidades específicas do sistema
  - *Vimos já anteriormente que este módulo integra a lista sys.argv, que contém os argumentos da linha de invocação da script em execução.*
- Suponha agora que se pretende verificar se a versão do Python instalada é a 3
  - *O módulo sys dá-nos essa informação:*

```
>>> import sys
>>> sys.version_info
sys.version_info(major=3, minor=6, micro=0, releaselevel='final', serial=0)
>>> sys.version_info.major == 3
True
```

Listas por Compreensão

- As Listas por Compreensão permitem-nos obter uma nova lista a partir de outra já existente (ou, mais rigorosamente, a partir de um objeto iterável existente),
  - *usando o 'for' para percorrer a lista original,*
  - *e indicando a operação a efetuar sobre cada elemento da lista original*
- Trata-se de uma forma rápida e muito compacta de construir listas
- Suponha-se que temos uma lista de números e pretendemos obter uma nova lista com todos esses números multiplicados por dois:

```
lista1=[2,3,4]
lista2=[2*x for x in lista1]
print(lista2)
```

```
[4, 6, 8]
```

Repare-se que ' lista1' não precisa de ser uma lista; basta tratar-se de um objeto iterável.

Listas por Compreensão

- E se a ideia fosse obter os valores dobrados apenas para os números superiores a 2?
  - *nessa altura aplicávamos um filtro (if…) à Lista por Compreensão:*

```
lista1=[2,3,4]
lista2=[2*x for x in lista1 if x>2]
print(lista2)
```

```
[6, 8]
```

- A sintaxe faz lembrar a notação matemática usada na teoria de conjuntos:

$$\{2x : x \in \{2, 3, 4\} \wedge x > 2\}$$

- No fundo, as Listas por Compreensão realizam 3 tipos de operações:
  - *selecionam valores a partir de uma iteração,*
  - *aplicam uma transformação a cada um desses valores,*
  - *e formam uma lista com esses novos valores.*

Expressões geradoras

- Basta trocarmos, numa lista por compreensão, os parêntesis retos por curvos, para se obter, não uma lista, mas um objeto gerador, capaz de reproduzir iterativamente (não fosse ele um iterador) os valores resultantes
  - *essa nova expressão é designada Expressão Geradora*

```
M↓
it=(2*x for x in lista1 if x>2)
print(it)
```

```
<generator object <genexpr> at 0x000002E9BC54A6D0>
```

```
M↓
next(it)
```

```
6
```

```
M↓
next(it)
```

```
8
```

```
M↓
next(it)
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-16-bc1ab118995a> in <module>
----> 1 next(it)

StopIteration:
```

Função Lambda

- Uma função lambda é uma função anónima de uma só linha
  - *Trata-se de um mecanismo de abstração procedimental adicional que o Python suporta, à semelhança de outras linguagens.*
- A sua formulação é simples:

```
lambda <parâmetros formais>: <expressão>
```

  - *sintaticamente, resume-se a uma simples expressão*
  - *é definida usando o prefixo* lambda*, em vez do* def
  - *é anónima porque não fica com um nome associado*
  - *não precisa da instrução return (o retorno do resultado está implícito)*
  - *não pode conter nem instruções de controlo, nem variáveis locais, para além dos parâmetros*
  - *pode ser usada sempre que seja esperado um objeto função*
- As funções lambda são úteis para definir funções simples e pequenas de uma forma rápida, evitando a criação de funções separadas através da clausula def.

Função Lambda

- São normalmente pequenas funções que são usadas
  - *quando apenas necessitam de ser executadas uma vez (são definidas no momento em que as usamos),*
  - *e sintaticamente não for adequada, ou não se justificar, a definição convencional da função*
- O objetivo da função lambda é produzir, em determinados contextos, código de melhor qualidade
  - *Porém, será sempre possível implementar as soluções sem recorrer a esse tipo de função*

  (basicamente, o que se consegue com esse tipo de funções é melhorar o estilo do código)
- Exemplo de função lambda:

```
(lambda x,y: x.lower()==y.lower())('ana','Ana')
```

```
True
```

Função lambda com map()/filter()

- A utilidade da função lambda torna-se mais evidente quando combinada com outras funções
  - *como é o caso das "funções" map() e filter()*

map(*função*, *lista*):

- tem como objetivo aplicar uma função a todos os elementos duma lista, resultando um iterador (objeto da classe map) para os respetivos resultados

```
lista1=[2,3,4]
lista2=list(map(lambda x: 2*x, lista1))
print(lista2)
```

```
[4, 6, 8]
```

filter(*função*, *lista*):

- tem como objetivo aplicar um filtro (definido por uma função booleana) a todos os elementos duma lista, resultando um iterador (objeto da classe filter) para os respetivos resultados

```
lista1=[2,3,4]
lista2=list(filter(lambda x: x%2==0, lista1))
print(lista2)
```

```
[2, 4]
```