

Módulos

- Ainda que de uma forma algo simplista, podemos afirmar:
 - *que se usam funções para se reutilizar um conjunto de linhas de código num mesmo programa;*
 - *e que se usam módulos para se reutilizar um conjunto de funções em diferentes programas;*
- Um módulo é essencialmente um conjunto de funções e outros elementos que podem ser integrados (importadas para) num qualquer programa.
 - *Pode consistir num ficheiro com extensão .py (ou .pyc, quando compilado), contendo funções, classes e variáveis em Python,*
 - *mas pode também ser escrito e compilado, por exemplo em C, para depois ser usado dentro de código Python.*

Módulos

- Um programa para usar as funcionalidades dum módulo deve primeiramente importá-lo.
 - É também isso que deve ser feito quando se pretende usar os módulos da biblioteca standard do Python.
- Quando o módulo é importado, o seu conteúdo é compilado (só 1^a vez) e executado

```
$ python module_using_sys.py we are arguments      # each arg is separated by white space
```

- O comando `import sys`, procura o módulo `sys`
 - *e sabe onde se encontra dado ser um módulo interno;*
 - *se não fosse, seria procurado num dos diretórios da variável `sys.path`.*
- A variável `sys.argv` é uma lista de strings contendo os argumentos passados na invocação do programa, sendo o 1º deles o nome do próprio programa em execução

Example (save as `module_using_sys.py`):

`import sys`

```
print('The command line arguments are:')
for i in sys.argv:
    print(i)

print('\n\nThe PYTHONPATH is', sys.path, '\n')
```

Output:

```
The command line arguments are:
module_using_sys.py
we
are
arguments
```

```
The PYTHONPATH is ['/tmp/py',
# many entries here, not shown here
'/Library/Python/2.7/site-packages',
'/usr/local/lib/python2.7/site-packages']
```

Módulos compilados para bytecode

- A importação de módulos pode revelar-se numa operação algo pesada
 - *Uma forma de tornar essa operação menos pesada passa por converter o módulo num ficheiro .pyc*
 - *Um ficheiro .pyc contém bytecode, um código intermédio que resulta da compilação do código Python original,*
 - sendo de mais rápida execução
 - e independente da plataforma.
 - *Quando um módulo .py com o código fonte é importado, o próprio Python gera automaticamente o ficheiro .pyc, e grava-o para evitar a necessidade de recompilar o módulo a cada execução.*

Declaração from..import

- Se se pretender importar não todo o módulo, mas apenas um dos elementos desse módulo (variável, classe ou função), deve-se usar a declaração from..import
 - *Por exemplo, para importar diretamente a variável argv no programa (e também para evitar usar repetidas vezes o qualificador sys.), deve usar-se a declaração*
`from sys import argv`
 - *Ou então, para importar a função sqrt do módulo 'math', escreve-se*

```
from math import sqrt  
print("Square root of 16 is", sqrt(16))
```

O atributo `__name__`

- Um módulo tem sempre um nome
 - esse nome pode ser consultado a partir de dentro do próprio módulo, durante a sua execução, acedendo ao atributo `__name__`
 - Tal como se pode ver no exemplo, é possível, através desse atributo, perceber-se se um módulo está a ser executado diretamente ou por importação

Example (save as `module_using_name.py`):

```
if __name__ == '__main__':
    print('This program is being run by itself')
else:
    print('I am being imported from another module')
```

Output:

```
$ python module_using_name.py
This program is being run by itself

$ python
>>> import module_using_name
I am being imported from another module
>>>
```

Criar e importar um módulo

- Como um programa em Python é ele próprio um módulo, podemos afirmar que já andamos a criar módulos há muito tempo...
(os módulos devem ser colocados no mesmo diretório do programa que os importa, ou então em um dos diretórios contidos em sys.path)

Example (save as `mymodule.py`):

```
def say_hi():
    print('Hi, this is mymodule speaking.')
__version__ = '0.1'
```

Another module (save as `mymodule_demo.py`)

```
import mymodule

mymodule.say_hi()
print('Version', mymodule.__version__)
```

utilising the `from..import` syntax

```
from mymodule import say_hi, __version__
say_hi()
print('Version', __version__)
```

Output:

```
$ python mymodule_demo.py
Hi, this is mymodule speaking.
Version 0.1
```

A função dir()

- A função dir() devolve a lista de nomes definidos por um objeto
 - Se esse objeto *for um módulo*, a lista inclui os nomes das funções, classes e variáveis definidas nesse módulo.
 - Se não for usado qualquer argumento na função dir(), é devolvida a lista de nomes do módulo atual

```
$ python
>>> import sys

# get names of attributes in sys module
>>> dir(sys)
['__displayhook__', '__doc__',
'argv', 'builtin_module_names',
'version', 'version_info']

# only few entries shown here

# get names of attributes for current module
>>> dir()
['__builtins__', '__doc__',
'__name__', '__package__', 'sys']

# create a new variable 'a'
>>> a = 5

>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'sys', 'a']

# delete/remove a name
>>> del a

>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'sys']
```

Packages

- Até aqui já foi perceptível alguma organização hierárquica dos programas em Python:
 - as variáveis *locais* encontram-se dentro de funções;
 - as variáveis *globais*, funções e classes encontram-se dentro de módulos;
- E como organizar os módulos?
 - é aqui que os packages entram em cena...
 - packages não são mais que pastas de módulos, com um ficheiro especial: `__init__.py`
 - é através desse ficheiro que o Python se apercebe que a pasta é especial (especial porque contém módulos)
 - é também esse ficheiro o primeiro a ser executado quando um módulo do package for invocado
(opcional a partir do Python 3.3)

Subpackages

- Cada Package pode conter outros Packages, designados subpackages.
 - A estrutura de pastas que se ilustra no exemplo, pretende representar a criação dum package denominado 'world' com os subpackages 'asia' e 'africa', e estes por sua vez com os supackages 'india' e 'madagascar', cada um contendo um módulo, foo.py e bar.py, respetivamente.
- ```
- <some folder present in the sys.path>/
 - world/
 - __init__.py
 - asia/
 - __init__.py
 - india/
 - __init__.py
 - foo.py
 - africa/
 - __init__.py
 - madagascar/
 - __init__.py
 - bar.py
```

# Estruturas de Dados

O Python disponibiliza-nos 4 estruturas de dados (coleções) nativas que nos facilitam bastante a vida...

- **Lista** – coleção heterogénea e ordenada de objetos (ordenada no sentido de que cada elemento ocupa uma posição bem determinada na estrutura, não dependendo essa posição da relação de ordem dos valores em si)
  - *na especificação de uma lista, os elementos devem ser colocados dentro de um par de parêntesis retos: lista=[2, 5, -3]*
  - *é uma estrutura variável, dado permitir remover e adicionar elementos*
  - *os seus elementos são acessíveis por indexão*
  - *as listas são instâncias da classe list*
- **Tuplo** – também uma coleção heterogénea e ordenada de objetos, mas com menos funcionalidades
  - *na especificação de um tuplo, os elementos são colocados, opcionalmente, dentro de um par de parêntesis curvos: tuplo1=(2, 5, -3); tuplo2=1, 2, 3;*
  - *é uma estrutura imutável, dado não permitir alterações ao seu conteúdo, o que a torna mais eficiente*
  - *habitualmente usada quando se pretende assegurar que os elementos colecionados não são alterados*
  - *os seus elementos também são acessíveis por indexão*
  - *os tuplos são instâncias da classe tuple*

# Estruturas de Dados

- **Dicionário** – coleção de pares chave/valor
  - na especificação de um dicionário, pares 'chave:valor' são colocados dentro de um par de chavetas:  
`dicionario={'Ana':18, 'Rui':15, 'Rita':20}`
  - as chaves têm que ser objetos imutáveis
  - já os valores podem ser imutáveis ou não
  - a partir do Python 3.7, preservam a ordem de inserção
  - os dicionários são instâncias da classe `dict`
- **Set** – coleção heterogénea, não ordenada e sem possibilidade de repetições
  - especifica-se com a função `set()` ou colocando os seus valores entre chavetas: `c1=set([2,5,3])`, `c2=set((2,5,3))`,  
`c3=set("253")`, `c4={2,5,3}`
  - os sets são instâncias da classe `set`
- Sequência – trata-se de um tipo de coleção mais conceptual e mais abrangente, que apresenta como principais características o facto de permitir o acesso aos seus elementos (ordenados posicionalmente) através das operações 'in' e 'not in' (*membership tests*), e através de indexação.
  - As *listas*, *tuplos*, *ranges* e *strings* são sequências

# Exemplo de utilização de uma lista

Não esquecer que a própria lista (de objetos) é um objeto

- dispõe por isso dum conjunto de funcionalidades (métodos) que facilitam a sua manipulação
  - *Para se conhecerem todos os seus métodos, basta fazer: help(list)*

```
This is my shopping list
shoplist = ['apple', 'mango', 'carrot', 'banana']

print('I have', len(shoplist), 'items to purchase.')

print('These items are:', end=' ')
for item in shoplist:
 print(item, end=' ')

print('\nI also have to buy rice.')
shoplist.append('rice')
print('My shopping list is now', shoplist)

print('I will sort my list now')
shoplist.sort()
print('Sorted shopping list is', shoplist)
print('The first item I will buy is', shoplist[0])
olditem = shoplist[0]
del shoplist[0]
print('I bought the', olditem)
print('My shopping list is now', shoplist)
```

# Output

```
I have 4 items to purchase.
These items are: apple mango carrot banana
I also have to buy rice.
My shopping list is now ['apple', 'mango', 'carrot', 'banana', 'rice']
I will sort my list now
Sorted shopping list is ['apple', 'banana', 'carrot', 'mango', 'rice']
The first item I will buy is apple
I bought the apple
My shopping list is now ['banana', 'carrot', 'mango', 'rice']
```

# Utilização de tuplos

```
zoo = ('python', 'elephant', 'penguin')
print('Number of animals in the zoo is', len(zoo))

new_zoo = 'monkey', 'camel', zoo # parentheses not required but are a good idea
print('Number of cages in the new zoo is', len(new_zoo))
print('All animals in new zoo are', new_zoo)
print('Animals brought from old zoo are', new_zoo[2])
print('Last animal brought from old zoo is', new_zoo[2][2])
print('Number of animals in the new zoo is',
 len(new_zoo)-1+len(new_zoo[2]))
```

## Output:

```
Number of animals in the zoo is 3
Number of cages in the new zoo is 3
All animals in new zoo are ('monkey', 'camel', ('python', 'elephant', 'penguin'))
Animals brought from old zoo are ('python', 'elephant', 'penguin')
Last animal brought from old zoo is penguin
Number of animals in the new zoo is 5
```

# Utilização de tuplos – retorno de 2 valores

- Uma das aplicações interessantes dos tuplos é na criação de funções que devolvam, não um, mas vários resultados.
  - Veja-se *um exemplo de função que retorna dois valores*

```
>>> def get_error_details():
... return (2, 'details')
...
>>> errnum, errstr = get_error_details()
>>> errnum
2
>>> errstr
'details'
```

- Ao podermos usar um tuplo à esquerda de uma atribuição, isso vai permitir, por exemplo, que a troca do valor de duas variáveis se passe a fazer de uma forma muito simples

```
>>> a = 5; b = 8
>>> a, b
(5, 8)
>>> a, b = b, a
>>> a, b
(8, 5)
```

# Utilização de um dicionário

O método `items()` do dicionário `ab` devolve uma lista de tuplos, contendo cada um deles um par chave/valor

- no exemplo, cada par é atribuído às variáveis `name` e `address` usando o `for..in`

```
ab = { # 'ab' is short for 'a'ddress'b'ook
 'Swaroop': 'swaroop@swaroopch.com',
 'Larry': 'larry@wall.org',
 'Matsumoto': 'matz@ruby-lang.org',
 'Spammer': 'spammer@hotmail.com'
}

print("Swaroop's address is", ab['Swaroop'])

Deleting a key-value pair
del ab['Spammer']

print('\nThere are {} contacts in the address-book\n'.format(len(ab)))

for name, address in ab.items():
 print('Contact {} at {}'.format(name, address))

Adding a key-value pair
ab['Guido'] = 'guido@python.org'

if 'Guido' in ab:
 print("\nGuido's address is", ab['Guido'])
```

# Output

```
Swaroop's address is swaroop@swaroopch.com
```

```
There are 3 contacts in the address-book
```

```
Contact Swaroop at swaroop@swaroopch.com
```

```
Contact Matsumoto at matz@ruby-lang.org
```

```
Contact Larry at larry@wall.org
```

```
Guido's address is guido@python.org
```

# Sets (conjuntos)

- Sets são coleções não ordenadas de objetos não repetidos.
  - Com estas estruturas de dados é possível realizar as operações típicas de conjuntos, tais como:
    - verificar se um set é subconjunto de outro,
    - e encontrar a interseção entre dois sets.

```
>>> bri = set(['brazil', 'russia', 'india'])
>>> 'india' in bri
True
>>> 'usa' in bri
False
>>> bric = bri.copy()
>>> bric.add('china')
>>> bric.issuperset(bri)
True
>>> bri.remove('russia')
>>> bri & bric # OR bri.intersection(bric)
{'brazil', 'india'}
```

# Utilização de sequências

```
shoplist = ['apple', 'mango', 'carrot', 'banana']
name = 'swaroop'
```

```
Indexing or 'Subscription' operation
print('Item 0 is', shoplist[0])
print('Item 1 is', shoplist[1])
print('Item 2 is', shoplist[2])
print('Item 3 is', shoplist[3])
print('Item -1 is', shoplist[-1])
print('Item -2 is', shoplist[-2])
print('Character 0 is', name[0])
```

```
Slicing on a list
print('Item 1 to 3 is', shoplist[1:3])
print('Item 2 to end is', shoplist[2:])
print('Item 1 to -1 is', shoplist[1:-1])
print('Item start to end is', shoplist[:])
```

```
Slicing on a string
print('characters 1 to 3 is', name[1:3])
print('characters 2 to end is', name[2:])
print('characters 1 to -1 is', name[1:-1])
print('characters start to end is', name[:])
```

## Output:

```
Item 0 is apple
Item 1 is mango
Item 2 is carrot
Item 3 is banana
Item -1 is banana
Item -2 is carrot
Character 0 is s
```

```
Item 1 to 3 is ['mango', 'carrot']
Item 2 to end is ['carrot', 'banana']
Item 1 to -1 is ['mango', 'carrot']
Item start to end is ['apple', 'mango', 'carrot', 'banana']
```

```
characters 1 to 3 is wa
characters 2 to end is aroop
characters 1 to -1 is waroo
characters start to end is swaroop
```

# Referências

- À semelhança do que acontece, por exemplo, em Java e C#, quando se atribui um objeto diretamente a uma variável, essa variável apenas irá conter uma referência para o objeto, e não uma cópia dele...
  - *E isso acontece com a atribuição duma sequência, que é também ela um objeto, como tudo no Python*
  - *Por vezes, para contrariar esse comportamento (i.e., para duplicar o objeto) na atribuição de sequências, usa-se a operação 'slicing' (ver próximo exemplo)*

# Referências

```
print('Simple Assignment')
shoplist = ['apple', 'mango', 'carrot', 'banana']
mylist is just another name pointing to the same object!
mylist = shoplist

I purchased the first item, so I remove it from the list
del shoplist[0]

print('shoplist is', shoplist)
print('mylist is', mylist)
Notice that both shoplist and mylist both print
the same list without the 'apple' confirming that
they point to the same object

print('Copy by making a full slice')
Make a copy by doing a full slice
mylist = shoplist[:]
Remove first item
del mylist[0]

print('shoplist is', shoplist)
print('mylist is', mylist)
Notice that now the two lists are different
```

## Output:

```
Simple Assignment
shoplist is ['mango', 'carrot', 'banana']
mylist is ['mango', 'carrot', 'banana']
Copy by making a full slice
shoplist is ['mango', 'carrot', 'banana']
mylist is ['carrot', 'banana']
```

# Ainda sobre strings

- Tratando-se de objetos, as strings dispõem de uma grande variedade de métodos que nos ajudam a fazer quase tudo o que precisamos...
  - são instâncias da classe str
  - para uma listagem completa dos seus métodos, executar `help(str)`

```
This is a string object
name = 'Swaroop'

if name.startswith('Sw'):
 print('Yes, the string starts with "Sw"')

if 'a' in name:
 print('Yes, it contains the string "a"')

if name.find('war') != -1:
 print('Yes, it contains the string "war"')

delimiter = '_*_'
mylist = ['Brazil', 'Russia', 'India', 'China']
print(delimiter.join(mylist))
```

Output:

```
Yes, the string starts with "Sw"
Yes, it contains the string "a"
Yes, it contains the string "war"
Brazil_*_Russia_*_India_*_China
```

# Leitura de dados do utilizador

```
def reverse(text):
 return text[::-1]

def is_palindrome(text):
 return text == reverse(text)

something = input("Enter text: ")
if is_palindrome(something):
 print("Yes, it is a palindrome")
else:
 print("No, it is not a palindrome")
```

Output:

Enter text: sir

No, it is not a palindrome

Enter text: madam

Yes, it is a palindrome

Enter text: racecar

Yes, it is a palindrome