UNIVERSITÀ di VERONA

Linguaggio di Programmazione Python

02/05/2022

Giulio Mazzi

giulio.mazzi@univr.it

Moduli

Python è un linguaggio "battery included", ovvero viene forniti insieme a una ricca raccolta di **pacchetti e moduli:** libreria software che possiamo importre per facilitarci il lavoro.

In genere, uso **`import`** per importare un modulo, ma esistono delle varianti.

Un pacchetto è una raccolta di moduli. Potete pensare al pacchetto come a una cartella, e ai suoi moduli come ai file contenuti cartella (anche se non è sempre così diretto).

```python
# importa il modulo
import math
# devo explictare il modulo
print(math.pi)

# importa solo uno elemento
from math import pi
# posso usarli direttamente
print(pi)

# m è un alias di math
import math as m
print(m.pi)
```

```
3.141592653589793
3.141592653589793
3.141592653589793
```

Creare un modulo

Un modulo è un semplice file python. Possiamo importarlo usando il suo nome (e omettendo l'estensione .py)

```
1 def somma_1(x):
2     return x + 1
3
```

File my_module.py con una sola funzione

```
import my_module

print(my_module.somma_1(5))
```

```
6
```

Main

Possiamo specificare un "main" del modulo (usando `if __name__ == '__main__'`): una sezione di codice che viene eseguita solo se uso il codice direttamente.

Se importo un modulo, il codice in main non viene eseguito.

```python
def somma_1(x):
    return x + 1

def main():
    print('ciao')
    print(somma_1(41))

if __name__ == '__main__':
    main()
```

```
$ python3 my_module.py
ciao
42
```

La chiamata diretta esegue il main

```python
import my_module

print(my_module.somma_1(5))
```

```
6
```

Importare il modulo no

Ambienti virtuali - venv

È facile installare una libreria, ma in genere vogliamo tenere il nostro progetto separato dal resto del sistema. Questo ha diversi benefici:

- Possiamo specificare esattamente la versione di python da usare
- Possiamo usare una versione specifica di una libreria (diversa per ogni progetto)
- Possiamo rendere il nostro ambiente di lavoro facilmente riproducibile

Lo facciamo creando un **ambiente virtuale**, una "bolla" che separa il nostro progetto dal resto del sistema.

Python offre un ambiente virtuale di default, **venv**, ma ce ne sono altri (virtualenv, conda...)

Documentazione: https://docs.python.org/3/library/venv.html

Venv - esempio

```
giulio@giulio-ThinkPad-T460p:~/esempio_progetto
$ python3 -m venv mio_ambiente
giulio@giulio-ThinkPad-T460p:~/esempio_progetto
$ cd mio_ambiente/
giulio@giulio-ThinkPad-T460p:~/esempio_progetto/mio_ambiente
$ ls
bin  include  lib  lib64  pyvenv.cfg  share
giulio@giulio-ThinkPad-T460p:~/esempio_progetto/mio_ambiente
$ source ./bin/ac
activate       activate.csh   activate.fish  Activate.ps1
giulio@giulio-ThinkPad-T460p:~/esempio_progetto/mio_ambiente
$ source ./bin/activate
(mio_ambiente) giulio@giulio-ThinkPad-T460p:~/esempio_progetto/mio_ambiente
$ python --version
Python 3.8.10
```

Installare moduli con pip

Possiamo aggiungere pacchetti al nostro sistema usando un **gestore di pacchetti** (package manager).

Python fornisce un package manager di default: **pip.** (ma ne esistono altri, e.g. conda).

Per installare una libreria uso: `pip install <library>`

```
$ python -m pip install matplotlib
Collecting matplotlib
  Downloading matplotlib-3.5.1-cp38-cp38-manylinux_2_5_x86_64.manylinux1_x86_64.whl (11.3 MB)
     |                                | 11.3 MB 3.2 MB/s
Collecting cycler>=0.10
  Downloading cycler-0.11.0-py3-none-any.whl (6.4 kB)
Collecting python-dateutil>=2.7
  Using cached python_dateutil-2.8.2-py2.py3-none-any.whl (247 kB)
Collecting kiwisolver>=1.0.1
```

Requirements.txt

Per rendere l'ambiente facilmente riproducibile possiamo salvare le librerie che abbiamo installato nel nostro ambiente di sviluppo nel file **requirements.txt**

- Salviamo con `pip freeze > requirements.txt`
- Usiamo con `pip install -r requirements.txt`

```
cycler==0.11.0
fonttools==4.33.3
kiwisolver==1.4.2
matplotlib==3.5.1
numpy==1.22.3
packaging==21.3
Pillow==9.1.0
pyparsing==3.0.8
python-dateutil==2.8.2
six==1.16.0
```

Esempio di requirements.txt

NumPy

Il primo modulo che vedremo al corso è **NumPy,** serve a fare computazioni numeriche pesanti.

Questo pacchetto fa da base a molte librerie scientifiche del Python, è quindi importante conoscerne i principali elementi.

NumPy offre la struttura dato "array", che può essere usata per rappresentare array n-dimensionali.

Offre inoltre diverse operazioni vettorizzate (i.e., estremamente efficienti). Il codice è implementato in **c.**

Documentazione: https://numpy.org/doc/stable/

NumPy array

A differenza delle liste Python, in NumPy tutti gli elementi di un array devono avere lo stesso tipo.

Quando li dichiaro, fornisco una forma (shape) che viene usata da numpy per allocare in maniera efficiente i miei dati.

```
import numpy as np

# array di 10 elementi posti a zero
v = np.zeros(10)
print(v)
print(type(v), v.dtype, v.shape)
```

```
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
<class 'numpy.ndarray'> float64 (10,)
```

NumPy Array - Shape

L'attributo `shape` definisce la "forma" dell'array (numero di righe, colonne ecc..),

Nel caso bidimensionale, il primo parametro è il numero di righe, il secondo quello di colonne.

Notate che `shape` è sempre restituito come tupla.

È possibile modificare la rehape con il metodo reshape (o con altri metodi specifici, come T per la trasposta).

```
v = np.zeros((2, 10))
print(v)
```

```
[[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]
```

```
print(v.T)
print(v.shape, v.T.shape)
```

```
[[0. 0.]
 [0. 0.]
 [0. 0.]
 [0. 0.]
 [0. 0.]
 [0. 0.]
 [0. 0.]
 [0. 0.]
 [0. 0.]
 [0. 0.]]
(2, 10) (10, 2)
```

NumPy array - Accesso

In caso di array multidimensionali, possiamo accedere a un elemento **mettendo tutti gli indici dentro alle parentesi quadre**

```
a = np.arange(25).reshape((5, 5))
print(a)
print(a[0, 0]) #riga 0 colonna 0
print(a[2, 3]) #riga 2 colonna 3
```

```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]
 [20 21 22 23 24]]
0
13
```

NumPy array - Slice

Possiamo usare le stesse operazioni di slice (già viste per le liste) anche in un array NumPy.

Attenzione: NumPy torna "viste" degli stessi dati, non crea vere e proprie copie con le slice. Se si desidera una copia, bisogna crearla esplicitamente con il metodo `copy()`.

```
print(a[1:2, :])
```

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

```
print(a[::-1, :3])
```

```
[[20 21 22]
 [15 16 17]
 [10 11 12]
 [ 5  6  7]
 [ 0  1  2]]
```

Vectorized operations

Applicare un operazione matematica tra due vettori applica lo stesso operatore elemento per elemento. Questo è implementato in maniera estremamente efficiente.

```
a = np.array([1.0, 2.3, 7, 2.999])
b = np.array([3.2, 1.0, 1, 1.2])

# vectorize operation
print(a + f)
print(a * f)
print(a / f)
print(a ** f)
print(a * 10)
```

```
[ 2.2    4.6   11.5    8.599]
[ 1.2     5.29   31.5    16.7944]
[0.83333333 1.         1.55555556 0.53553571]
[1.00000000e+00 6.79163008e+00 6.35244890e+03 4.68887018e+02]
[10.   23.   70.   29.99]
```

Fancy indexing

Esiste un metodo alternativo di usare gli indici, noto come **fancy indexing o Boolean indexing.**

Usa degli array di Booleani per selezionare solo gli elementi d'interesse. Si può creare usando operatori di confronto con l'array

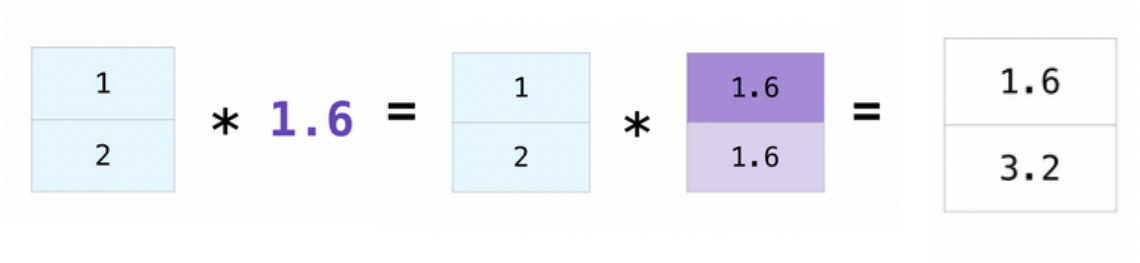
```
# fancy indexing (by value)
a = np.array([-1, -3, 1, 4, -6, 9, 3])
mask = a < 0
print(a[mask])

div3 = a % 3 == 0
print(a[div3])
```

```
[-1 -3 -6]
[-3 -6  9  3]
```

Broadcasting

È possibile applicare operazioni elemento-per-elemento anche in (alcuni) contesti in cui le dimensioni non combaciano. NumPy colma la differenza ripetendo il dato originale più volte. Questa operazione si chiama **broadcasting**.




```
# broadcasting
a =  np.arange(25).reshape(5, 5)
# l'array 1-dimensionale b viene ripetuto più volte
b = np.array([10, 20, 30, 40, 50])
# b si comporta come
# b = [[10, 20, 30, 40, 50],
#      [10, 20, 30, 40, 50],
#      [10, 20, 30, 40, 50],
#      [10, 20, 30, 40, 50],
#      [10, 20, 30, 40, 50]]
print(a + b)
```

```
[[10 21 32 43 54]
 [15 26 37 48 59]
 [20 31 42 53 64]
 [25 36 47 58 69]
 [30 41 52 63 74]]
```