Basi di Python

April 24, 2022

1 Crediti e riferimenti

Documento realizzato da Michele Giacomoli prendendo liberamente spunto da:

- Tutorial di Nicola Cassetta https://ncassetta.altervista.org/Tutorial_Python/index.html
- Tutorial di Python ABC https://pythonitalia.github.io/python-abc/
- Tutoria di Christopher Tao https://towardsdatascience.com/the-simplest-tutorial-for-python-decorator-dadbf8f20b0f

2 Installazione e predisposizione dell'ambiente di lavoro

2.1 Installare Python

Se stai usando il sistema operativo giusto questo capitolo non ti serve. Per tutti gli altri sistemi operativi puoi fare riferimento alla pagina ufficiale di Python https://wiki.python.org/moin/BeginnersGuide/Download

Attenzione alla versione. In molte distribuzioni Python 2, non più supportato, è ancora presente e installato con il comando `python`, mentre Python 3 è eseguibile tramite il comando `python3`. Verificare la versione in uso tramite il comando

```
$ python3 -V
Python 3.8.10
```

2.2 Virtual Environment

2.2.1 Installare il virtual environment

```
$ sudo apt install python3-venv
$ mkdir virtualenv
$ python3 -m venv virtualenv/
```

2.2.2 Usare il virtual environment

```
$ source virtualenv/bin/activate
(virtualenv) $
```

2.3 Gestione moduli: il comando pip

2.3.1 Installare un modulo

```
(django) $ pip install django
Collecting django
  Downloading Django-4.0.4-py3-none-any.whl (8.0 MB)
     |                | 8.0 MB 1.9 MB/s
Collecting asgiref<4,>=3.4.1
  Downloading asgiref-3.5.0-py3-none-any.whl (22 kB)
Collecting backports.zoneinfo; python_version < "3.9"
  Downloading backports.zoneinfo-0.2.1-cp38-cp38-manylinux1_x86_64.whl (74 kB)
     |                | 74 kB 570 kB/s
Collecting sqlparse>=0.2.2
  Downloading sqlparse-0.4.2-py3-none-any.whl (42 kB)
     |                | 42 kB 198 kB/s
Installing collected packages: asgiref, backports.zoneinfo, sqlparse, django
Successfully installed asgiref-3.5.0 backports.zoneinfo-0.2.1 django-4.0.4 sqlparse-0.4.2
(django) $
```

2.3.2 Elencare i moduli presenti

```
(django) $ pip list
Package            Version
------------------ -------
asgiref            3.5.0
backports.zoneinfo 0.2.1
Django             4.0.4
pip                20.0.2
pkg-resources      0.0.0
setuptools         44.0.0
sqlparse           0.4.2
```

2.3.3 Freeze dei moduli attualmente installati

Questa procedura ci permette di generare un file con la descrizione dei moduli installati nel virtual environment e di replicare l'environment in un momento successivo o in un ambiente differente.

```
(django) $ pip freeze > requirements.txt
```

2.3.4 Ripristino dei moduli installati a partire dal file di requirements

```
(django) $ pip install -r requirements.txt
```

2.4 Console interattiva

```
(virtualenv) $ python
Python 3.8.10 (default, Mar 15 2022, 12:22:08)
[GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

[4]:
```
4+6
```

[4]:
```
10
```

[5]:
```
11/2
```

[5]:
```
5.5
```

3 Sintassi

3.1 Espressioni algebriche

Operatore	Significato	Esempio
+	Addizione	`>>> 3 + 5` => 8
-	Sottrazione	`>>> 10 - 15` => -5
	Moltiplicazione	`>>> 6 * 7` => 42
/	Divisione	`>>> 12 / 5` => 2.4
	Elevamento a potenza	`>>> 2 ** 5` => 32
//	Divisione intera	`>>> 13 // 5` => 2
%	Resto della divisione	`>>> 13 % 5` => 3

3.2 Stringhe

- Tra apici

[6]:
```
'Ciao'
```

[6]:
```
'Ciao'
```

- Tra virgolette

[7]:
```
"Io sono Luigi"
```

[7]:
```
'Io sono Luigi'
```

- Tra **tripli** apici

[1]:
```
'''e ho 15 anni'''
```

[1]:
```
'e ho 15 anni'
```

- Tra **triple** virgolette

[19]:
```
"""La penna è sul tavolo"""
```

[19]:
```
'La penna è sul tavolo'
```

La differenza principale è che le ultime due soluzioni permettono di andare a capo nella definizione della stringa

[20]:
```
print('''Questa è una riga
e questa è un'altra riga''')
```

```
Questa è una riga
e questa è un'altra riga
```

Ma questo non mi vincola in alcun modo a definire stringhe con più righe usando apici singoli

[46]:
```
print('So long\nand thanks for all the fish')
```

```
So long
and thanks for all the fish
```

Qualche operazione con le stringhe

[7]:
```
print('mamma' + 'mia')
print('mamma ' * 3)
```

```
mammamia
mamma mamma mamma
```

3.3 Espressioni booleane

[23]:
```
5 > 1
```

[23]: `True`

[24]:
```
5 < 1
```

[24]: `False`

[26]:
```
3 < 5 < 10
```

[26]: `True`

Operatore	Significato	Esempio
<	Minore	`>>> 3 < 5` => True
<=	Minore o uguale	`>>> 10 <= 10` => True
>	Maggiore	`>>> 3 > 5` => False
>=	Maggiore o uguale	`>>> 10 >= 10` => True
==	Uguale	`>>> 3 == 5` => False
!=	Diverso	`>>> 3 != 5` => True

si possono creare espressioni booleane più complesse concatenando espressioni semplici tramite gli operatori `and` `or` `not` (con l'usuale significato dell'algebra booleana)

[25]:
```
(2 < 4) and (3 < 7)
```

[25]:
```
True
```

3.4 Variabili

[27]:
```
b = 3 * 5**2
b
```

[27]:
```
75
```

[31]:
```
a = 3 * 5**2
a = 'Ciao'
a
```

[31]:
```
'Ciao'
```

[1]:
```
a = y + 2
a
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Input In [1], in <cell line: 1>()
----> 1 a = y + 2
      2 a

NameError: name 'y' is not defined
```

In questo caso abbiamo ricevuto un messaggio di errore poiché la variabile y non è ancora stata definita

I nomi delle variabili non sono limitati ad una lettera, ma possono essere lunghi quanto vogliamo in modo da poter creare dei nomi che ci ricordino il contenuto della variabile. Abbiamo delle semplici regole per formare tali nomi:

- Il nome di una variabile può contenere solo lettere, numeri ed il carattere _ (underscore)
- Non può cominciare con un numero
- Non può essere uguale ad una delle parole riservate di Python
- Le lettere maiuscole e minuscole sono considerate differenti (in Inglese si dice che il linguaggio è case sensitive).

3.5 Tipi di dato

In informatica il tipo di un dato determina quali valori esso può assumere e quali operazioni si possono fare su di esso.

Quando noi assegnamo un valore ad una variabile Python ricorda che tipo di dato è conservato in essa

[36]:
```
type("Mamma")
```

[36]:
```
str
```

[37]:
```
a = 1234
type(a)
```

[37]:
```
int
```

[38]:
```
b = 12.34
type(b)
```

[38]:
```
float
```

[39]:
```
c = True
type(c)
```

[39]:
```
bool
```

Alcuni linguaggi di programmazione (ad esempio il C) sono molto rigorosi riguardo ai tipi di dato e richiedono la dichiarazione delle variabili: prima di usare una variabile a è necessario scrivere un'istruzione che dica al linguaggio "Intendo usare una variabile a che conterrà un intero", dopodichè in quella variabile potrò mettere solo numeri interi. Questo permette al linguaggio di sapere in anticipo quanta memoria occuperà la variabile a e quindi di facilitarne la memorizzazione.

Python, come abbiamo visto, non richiede la dichiarazione (bisogna solo ricordarsi di assegnare sempre un valore alla variabile prima di farne qualsiasi altro uso) e permette anche di assegnare tipi diversi alla stessa variabile. E' proprio della "filosofia" di Python cercare di nascondere tutto il lavoro che lui fa nella memoria del computer in modo che l'utente non se ne debba preoccupare.

In Python è possibile convertire il tipo di dato utilizzando le cosiddette funzioni (che vedremo tra pochissimo) `int()`, `float()`, `str()`, ecc...

3.6 Funzioni

Una funzione è una parte di codice che permette di compiere operazioni su dei dati (argomenti) che noi le forniamo. Per chiamare una funzione (Inglese: function call) si usa la seguente sintassi:

```
nomefunzione ( arg1, arg2, ... )
```

[40]:
```
max(1, 4, 7, 3)
```

[40]:
```
7
```

Ribadiamo per prima cosa che per chiamare una funzione dobbiamo:

- scrivere il suo nome (nell'esempio max o min)
- aprire le parentesi tonde
- scrivere gli argomenti della funzione, separati da virgole

- chiudere le parentesi tonde

Possiamo lasciare degli spazi prima e dopo le parentesi e le virgole; la prassi comune, per avere una migliore leggibilità, è di non lasciarne attorno alle parentesi (nè prima nè dopo) e di lasciare uno spazio dopo ogni virgola.

In Matematica sarete probabilmente abituati ad usare funzioni del tipo y = f(x); nei linguaggi di programmazione il concetto di funzione è simile a quello della Matematica (una "macchina" che produce un valore y in base ad un valore di partenza x) ma gode di molta più libertà. Vediamo perchè:

- Nei linguaggi di programmazione le funzioni possono avere più di un argomento: alcune (come max) possono averne addirittura un numero variabile (nell'esempio precedente ne abbiamo messi 4, ma se ne possono inserire una quantità variabile).
- I tipi di dato degli argomenti possono essere qualsiasi e diversi tra di loro: numeri, stringhe, bool … In genere però una funzione richiede di essere chiamata con determinati tipi di dato, altrimenti ci darà un errore.
- Le funzioni generalmente restituiscono un risultato, che, di nuovo, non deve essere per forza un numero ma può essere di qualsiasi tipo. Esistono però funzioni che non restituiscono nessun valore (esse "fanno qualcosa" con gli argomenti che forniamo loro, senza però ricavarne un valore in uscita).
- Infine esistono anche funzioni senza argomenti (esse "fanno sempre la stessa cosa" ogni volta che le chiamiamo). Per chiamarle è necessario scrivere le parentesi aperte e poi subito chiuse.

  `print()` e `type()` visti in precedenza sono due esempi di funzioni

[42]:
```
abs(-10)
```

[42]:
```
10
```

[43]:
```
abs('mouse')
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [43], in <cell line: 1>()
----> 1 abs('mouse')

TypeError: bad operand type for abs(): 'str'
```

[44]:
```
abs()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [44], in <cell line: 1>()
----> 1 abs()

TypeError: abs() takes exactly one argument (0 given)
```

[45]:
```
max(abs(-10), len("zio"), -2)
```

[45]: 10

3.7 Commenti

Tutti i linguaggi di programmazione permettono di inserire dei commenti, cioè delle parti di codice che vengono ignorate dal linguaggio ma che possono essere lette da una persona "umana": così è possibile intercalare alle linee di codice delle linee che spiegano cosa stiamo facendo, documentando il nostro programma. In Python i commenti sono preceduti dal carattere # : quando in una linea Python trova questo carattere, ignora tutto ciò che viene dopo fino alla fine della linea stessa.

[47]:
```
# Questa e' una linea intera di commento
print("Ciao") # Qui il commento inizia dopo l'istruzione
```

```
Ciao
```

3.8 Le istruzioni condizionali - if, else, elif

- Un programma è una serie di linee di codice che vengono eseguite una dopo l'altra
- Molte situazioni della vita reale richiedono di prendere decisioni tra più alternative
- Il programmatore deve prevedere tutte le alternative possibili e scrivere il codice necessario a gestire ognuna di esse
- un programma è suddiviso in blocchi di codice che possono essere eseguiti, saltati o ripetuti molte volte
- L'esecuzione dei blocchi è controllata da apposite istruzioni del linguaggio, che vengono dette istruzioni di controllo di flusso

[10]:
```
a = int(input("Scrivi un numero > 10 "))
if a > 10:
    print("Bravo!")
else:
    print("Hai sbagliato!")

b = int(input("Ora scrivi un altro numero "))
if b > 0:
    print("Il numero e' positivo")
elif b == 0:
    print("Hai scritto zero")
else:
    print("Il numero e' negativo")
print("Ciao")
```

```
Scrivi un numero > 10 5
Hai sbagliato!
Ora scrivi un altro numero 4
Il numero e' positivo
Ciao
```

[12]:
```
num = int(input("Scrivi un numero tra 0 e 20 "))
if num < 10:
    if num % 2 == 0:        # il numero da resto 0 se diviso per 2, quindi e'␣
 ↪pari
        print ("Hai scritto un numero pari minore di 10")
    else:                # il numero e' dispari
        print ("Hai scritto un numero dispari minore di 10")
else:
    if num % 2 == 0:
        print ("Hai scritto un numero pari maggiore di 10")
    else:
        print ("Hai scritto un numero dispari maggiore di 10")
```

```
Scrivi un numero tra 0 e 20 4
Hai scritto un numero pari minore di 10
```

3.9 I cicli - while, break, continue

> Un ciclo è un blocco di istruzioni (sempre riconosciuto da Python per mezzo dell'indentazione) che può essere ripetuto più volte.

Il controllo su quante volte ripetere un ciclo sarà assunto da un'apposita istruzione di controllo di flusso, che determinerà quando si deve uscire dal ciclo

[14]:
```
s = ""                    # stringa vuota: serve per inizializzare la variabile␣
 ↪s
while s != "mamma":               # while con l'espressione di controllo
    s = input("Scrivi 'mamma' ")  # questo è il blocco da ripetere
print ("Bravo!")                  # qui si esce dal ciclo
```

```
Scrivi 'mamma' sad
Scrivi 'mamma' mamma
Bravo!
```

L'istruzione while permette di ripetere un blocco di linee finchè una condizione di controllo (espressione booleana) rimane vera

[15]:
```
from random import *      # verra' spiegata in seguito
seed()                    # idem

num = randrange(1, 21)    # questa istruzione genera un numero casuale da 1 a␣
 ↪20: dovremo indovinarlo
print ("Ho pensato un numero da 1 a 20. Prova ad indovinarlo!")
mio_num = -1              # inizializza mio_num ad un valore sicuramente␣
 ↪diverso da num ...
while mio_num != num:     # ... quindi la prima volta entra sicuramente nel␣
 ↪while
    mio_num = int(input("??? "))    # qui introduciamo il nostro tentativo␣
 ↪(mio_num)
```

```
    if mio_num < num:
        print("Troppo basso")
    elif mio_num > num:
        print("Troppo alto")
print("Hai indovinato!")
```

```
Ho pensato un numero da 1 a 20. Prova ad indovinarlo!
??? 5
Troppo alto
??? 3
Troppo alto
??? 1
Hai indovinato!
```

Quando, durante un ciclo, Python incontra l'istruzione break tutta le linee dal break fino alla fine del ciclo vengono saltate ed il programma esce dal ciclo continuando con la prima istruzione successiva.

[55]:
```
n = int (input("Scrivi un numero "))     # controlliamo se questo numero è primo
k = 2                                    # contatore: parte da 2 ...
while k <= n // 2:                       # ... e arriva a n//2
    if n % k == 0:                       # se questo e' vero, n e' divisibile␣
 ↪per k
        print(n, "e' divisibile per", k)
        break                            # trovato un divisore, usciamo dal ciclo
    k += 1
```

```
Scrivi un numero 4
4 e' divisibile per 2
```

Questa istruzione provoca immediatamente la prossima iterazione: si salta tutta la parte dal continue alla fine del ciclo. E' un'istruzione in effetti non indispensabile (si potrebbe ottenere lo stesso effetto con un if o un else), ma che permette spesso di rendere più leggibile il programma. Usarla o no è questione di gusti del programmatore: tipicamente si usa se vogliamo saltare qualche iterazione nel ciclo.

[2]:
```
salta = int(input("Che numero devo saltare? "))
i = 0
while i < 5:
    i += 1
    if i == salta:
        continue
    print("i =", i)
    print("Il quadrato di i e'", i ** 2)
    print("Il cubo di i e'", i ** 3)
print ("Fine")
```

```
Che numero devo saltare? 3
i = 1
```

```
Il quadrato di i e' 1
Il cubo di i e' 1
i = 2
Il quadrato di i e' 4
Il cubo di i e' 8
i = 4
Il quadrato di i e' 16
Il cubo di i e' 64
i = 5
Il quadrato di i e' 25
Il cubo di i e' 125
Fine
```

3.10 I moduli

In Python possiamo usare migliaia di funzioni. Esse non sono però tutte disponibili immediatamente, per un motivo soprattutto logistico.

Quindi si è scelto di rendere immediatamente disponibili solo poche funzioni (ad esempio max(), print() che abbiamo già visto) mentre le altre sono raggruppate in files esterni, chiamati **moduli**. Ogni modulo contiene funzioni attinenti ad un argomento (matematica, grafica, tempo, interfaccia con il sistema operativo …): quando dobbiamo usare queste funzioni **possiamo dire a Python di importare quel modulo** e così potremo usare le sue funzioni.

[57]:
```
sqrt(4)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Input In [57], in <cell line: 1>()
----> 1 sqrt(4)

NameError: name 'sqrt' is not defined
```

[19]:
```
import math
math.sqrt(4)
```

[19]:
```
2.0
```

3.11 Le liste

Una lista è un insieme di dati (detti elementi) a cui è associato un ordine. Per definire una lista si scrivono i suoi elementi, separati da virgole, tra parentesi quadre.

[20]:
```
citta = ["Roma", "Milano", "Firenze", "Bologna"]  # una lista di 4 string
numeri = [1, 4, 5, 12, 23, 40]                    # una lista di 6 interi
mista = [1, "due", 3, "quattro"]                  # elementi di tipo diverso
numeri_uguali = [1, 4, 1, 6, 6, 4, 7, 7, 7, 1, 6] # elementi ripetuti
vuota = []                                        # lista vuota, senza elementi
```

```
print(citta)
print(numeri)
```

```
['Roma', 'Milano', 'Firenze', 'Bologna']
[1, 4, 5, 12, 23, 40]
```

[60]:
```
print(citta)
print(numeri)
print(citta + numeri)
print(2 * citta)
print(type(mista))
```

```
['Roma', 'Milano', 'Firenze', 'Bologna']
[1, 4, 5, 12, 23, 40]
['Roma', 'Milano', 'Firenze', 'Bologna', 1, 4, 5, 12, 23, 40]
['Roma', 'Milano', 'Firenze', 'Bologna', 'Roma', 'Milano', 'Firenze', 'Bologna']
<class 'list'>
```

3.11.1 Funzioni sulle liste

Sugli oggetti di tipo lista possiamo applicare un gran numero di funzioni. La maggior parte di esse viene chiamata però non con la sintassi che già conosciamo, ma con una sintassi simile a quella che abbiamo già visto per i moduli.

> Se nome_lista è una variabile di tipo lista, possiamo chiamare una funzione nome_funzione applicata ad essa con la sintassi:

```
nome_lista.nome_funzione(argomenti ...)
```

Funzione	Significato
l.append(a)	Aggiunge l'elemento a alla fine della lista l
l.insert(n, a)	Aggiunge l'elemento a al posto n nella lista (i posti sono numerati da 0, vedi sotto)
l.remove(a)	Rimuove dalla lista l l'elemento a. Se a non appartiene alla lista provoca un ValueError
l.sort()	Ordina gli elementi della lista in ordine alfabetico o numerico
l.reverse()	Inverte l'ordine degli elementi della lista
l.count(a)	Restituisce (come int) il numero di occorrenze di a nella lista (cioè quante volte l'elemento a è contenuto in l, 0 se non è presente)
l.pop(a)l.pop()	Rimuove l'elemento a dalla lista e lo restituisce come risultato della funzione. Usata senza parametro rimuove l'ultimo elemento
l.clear()	Elimina tutti gli elementi: la lista l diventa vuota

Funzioni più usate Da notare l'uso della funzione `len(lista)` con la sintassi già vista in precedenza per ottenere la lunghezza della lista

Selezione degli elementi

Per accedere ad un elemento si deve scrivere il nome della lista e subito dopo l'indice dell'elemento tra parentesi quadre, secondo la sintassi nome_lista[indice] sia in lettura che in scrittura

[61]:
```
citta = ["Roma", "Milano", "Venezia", "Firenze", "Bologna"]
print(citta[4])
citta[2] = "Napoli"
print(citta)
print(citta[0] + citta[1])
```

```
Bologna
['Roma', 'Milano', 'Napoli', 'Firenze', 'Bologna']
RomaMilano
```

[65]:
```
print(citta.index("Roma"))
print(citta.index("Firenze"))
print(citta.index("Torino"))
```

```
0
3
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Input In [65], in <cell line: 3>()
      1 print(citta.index("Roma"))
      2 print(citta.index("Firenze"))
----> 3 print(citta.index("Torino"))

ValueError: 'Torino' is not in list
```

[66]:
```
print("Roma" in citta)
print("Catanzaro" in citta)
```

```
True
False
```

Slice notation Python ha la possibilità di dare all'utente la possibilità di selezionare non solo singoli elementi, ma intere sottosequenze della sequenza principale, tramite una serie di regole sintattiche che sono state ribattezzate slice notation

[71]:
```
# Indici Negativi

citta = ["Roma", "Milano", "Napoli", "Firenze", "Bologna"]
```

```
print(citta[-1])
print(citta[-2])

nome = "Massimiliano"
print(nome[-1])
print(nome[-2])
```

```
Bologna
Firenze
o
n
```

[72]:
```
# Sottosequenze

print(citta[1:3])
print(nome[3:9])
print(nome[5:5])
print(nome[5:4])
```

```
['Milano', 'Napoli']
simili
```

[76]:
```
print(citta[-3:-1])    # gli elementi dal terzultimo all'ultimo
print(nome[3:-2])      # le lettere di 'Massimiliano' dalla terza alla penultima
print(citta[:3])        # dalla prima lettera alla terza
print(nome[2:8:2])        # dalla terza lettera alla settima ogni due lettere
print(nome[::-3])         # dall'ultima lettera alla prima ogni tre lettere
ind = 2
print(nome[ind+1:])    # dalla lettera con indice "ind" + 1 fino alla fine
```

```
['Napoli', 'Firenze']
similia
['Roma', 'Milano', 'Napoli']
sii
oims
similiano
```

3.11.2 Tuple

Le tuple sono oggetti molto simili alle liste, la differenza sostanziale da queste ultime è che non è possibile variare l'insieme di elementi dopo che la tupla è stata definita. **La tupla si definisce come la lista, usando parentesi tonde (e) al posto delle parentesi quadre**

3.12 I Dizionari

Un dizionario è un insieme di coppie chiave-dato: il primo elemento della coppia contiene la chiave (cioè il nome del dato), il secondo il suo valore

[92]:
```
persona1 = {"Nome":"Luigi", "Cognome":"Bianchi", "Eta":35}
persona2 = {"Nome":"Pietro", "Cognome":"Verdi", "Eta":38}
print(persona1["Nome"])
print(persona2["Eta"])

persona1["Nome"] = "Mario"
print(persona1["Nome"])
```

```
Luigi
38
Mario
```

3.13 Il ciclo for

E' usata anch'essa per i cicli, in particolare quando serve una variabile contatore. L'istruzione crea tale variabile automaticamente, e la fa variare nell'insieme degli elementi di una lista. Se abbiamo bisogno di una variabile contatore che vari su un certo insieme numerico basta usare la funzione `range()`.

[79]:
```
n = int(input("Quale tabellina vuoi stampare? "))
for i in range(1, 11):          # i numeri da 1 a 10
    print (n, "x", i, "=", n*i)
```

```
Quale tabellina vuoi stampare? 2
2 x 1 = 2
2 x 2 = 4
2 x 3 = 6
2 x 4 = 8
2 x 5 = 10
2 x 6 = 12
2 x 7 = 14
2 x 8 = 16
2 x 9 = 18
2 x 10 = 20
```

3.14 Definizione di funzioni

- Bisogna stabilire il nome della funzione (che servirà poi per chiamarla)
- Bisogna stabilire i suoi argomenti
- Bisogna infine scrivere il codice che sarà eseguito da Python ogni volta che la funzione verrà chiamata

[80]:
```
def firma():
    print("\tFirmato: Il grande Peppe")
```

```
    print("\tImperatore galattico")
    print("\tPadrone dell'Universo")

# L'esecuzione inizia da qui!!!
print ("Tutti i professori devono mettermi 10")
firma()
print ("Tutti i compagni devono passarmi i compiti")
firma()
print ("Il bar della scuola deve farmi i cappuccini gratis")
firma()
```

```
Tutti i professori devono mettermi 10
        Firmato: Il grande Peppe
        Imperatore galattico
        Padrone dell'Universo
Tutti i compagni devono passarmi i compiti
        Firmato: Il grande Peppe
        Imperatore galattico
        Padrone dell'Universo
Il bar della scuola deve farmi i cappuccini gratis
        Firmato: Il grande Peppe
        Imperatore galattico
        Padrone dell'Universo
```

[81]:
```
# Funzione con argomenti

def saluta(nome):
    print ("Ciao", nome + "!")   # nome + "!" non inserisce lo spazio tra il␣
 ↪nome e "!"

# qui inizia il programma principale
saluta("Peppe")
saluta("Ciccio")
n = input("Chi vuoi salutare? ")
saluta(n)
```

```
Ciao Peppe!
Ciao Ciccio!
Chi vuoi salutare? asd
Ciao asd!
```

[85]:
```
# Funzione con argomenti opzionali

def saluta(nome="Peppe"):
    print ("Ciao", nome + "!")

saluta()
```

```
saluta("Ciccio")
n = input("Chi vuoi salutare? ")
saluta(n)
```

```
Ciao Peppe!
Ciao Ciccio!
Chi vuoi salutare?
Ciao !
```

Nella definizione di una funzione possiamo avere sia parametri con valore di default (opzionali) sia parametri ordinari (obbligatori). Gli argomenti opzionali devono sempre seguire quelli obbligatori nella lista dei parametri.

[87]:
```
def compra(cosa, prezzo, quantita=1):
    print("Mi serve", quantita, "etto di", cosa)
    print("Ecco qui")
    print("Quanto costa?")
    print(prezzo * quantita, "euro")

compra("Prosciutto", 1.5, 2)   # compra 2 etti di prosciutto a 1.5 euro l'uno
compra("Prosciutto", 1.5, quantita=3)   # compra 2 etti di prosciutto a 1.5␣
 ↪euro l'uno,
                                        # ma utilizzando un KEYWORD ARGUMENT
compra("Salame", 1.0)          # compra 1 etto (default) di salame a 1.0 euro
compra("Carne")                # ERRORE! un solo argomento (ne servono almeno 2)
```

```
Mi serve 2 etto di Prosciutto
Ecco qui
Quanto costa?
3.0 euro
Mi serve 3 etto di Prosciutto
Ecco qui
Quanto costa?
4.5 euro
Mi serve 1 etto di Salame
Ecco qui
Quanto costa?
1.0 euro
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [87], in <cell line: 11>()
      9                                       # ma utilizzando un KEYWORD␣
 ↪ARGUMENT
     10 compra("Salame", 1.0)          # compra 1 etto (default) di salame a 1.␣
 ↪euro
---> 11 compra("Carne")
```

```
TypeError: compra() missing 1 required positional argument: 'prezzo'
```

3.14.1 L'istruzione return

Serve a far si che la funzione restituisca un risultato al blocco di codice che l'ha chiamata

- `return`: provoca l'immediata terminazione della funzione ed il ritorno al programma chiamante. Come per il `break` può essere usata per avere diversi punti di uscita, anche prima della fine del corpo della funzione.
- `return espressione` provoca anch'essa la terminazione della funzione, inoltre il risultato dell'espressione viene restituito al programma chiamante.

[82]:
```
def cubo(x):
    return x ** 3

def minimo(x, y):
    if x <= y:
        return x
    else:
        return y

a = float(input("Scrivi un numero "))
print("Il cubo di", a, "e'", cubo(a))
b = float(input("Adesso un altro numero "))
print("Il minimo tra", a, "e", b, "e'", minimo(a, b))
```

```
Scrivi un numero 3
Il cubo di 3.0 e' 27.0
Adesso un altro numero 5
Il minimo tra 3.0 e 5.0 e' 3.0
```

3.14.2 Variabili globali e locali

- Le variabili create nel programma principale si dicono variabili globali: esse esistono dal punto nel quale sono definite fino alla fine del programma, ed è sempre possibile leggerle e scriverle nel programma principale
- I parametri delle funzioni e le variabili create all'interno di esse sono invece variabili locali: esse possono essere lette e scritte solo all'interno della funzione e vengono distrutte (cioè cancellate dalla memoria) quando si esce da essa.

[83]:
```
def prova(x):           # x e' il parametro (variabile locale)
    x = x + 1           # cambiamo il valore del parametro
    print ("Dentro la funzione x =", x)
                        # scrivera' "Dentro la funzione x = 11"
                        # all'uscita dalla funzione x viene distrutta

# inizio del programma principale
a = 10                  # creiamo la variabile a
```

```
print ("Prima della chiamata a =", a)
                        # scriverà "Prima della chiamata a = 10"
prova(a)                # passiamo a come parametro alla funzione
print ("Dopo la chiamata a =", a)
                        # scrivera' "Dopo la chiamata a = 10": a non e' cambiata␣
 ↪fuori dalla funzione
```

```
Prima della chiamata a = 10
Dentro la funzione x = 11
Dopo la chiamata a = 10
```

[84]:
```
# Istruzione global

def nuova_a():
    global a
    a = 10      # ora a e' globale
    print ("Nella funzione a vale", a, "e b vale", b)

a = 5           # qui a e' globale
b = 20
print ("Prima della funzione a vale", a, "e b vale", b)
nuova_a()
print ("Dopo la funzione a vale", a, "e b vale", b)
```

```
Prima della funzione a vale 5 e b vale 20
Nella funzione a vale 10 e b vale 20
Dopo la funzione a vale 10 e b vale 20
```

3.15 Creazione di moduli

Possiamo creare dei file separati con le nostre funzioni da richiamare con l'istruzione import che abbiamo visto in precednza. Per fare questo possiamo creare un file `myinput.py` e inserire il seguente contenuto:

```
def input_sn(prompt):
    while True:
        risp = input(prompt)
        if risp in ["s", "n"]:
            return risp
        print("Input scorretto")
```

Nel file principale potremo utilizzare la nostra funzione scrivendo quanto segue

```
import myinput
# codice, codice, codice...
risp = myinput.input_sn("Vuoi continuare? (s/n) ")
```

Oppure

```
from myinput import *
```

```
# codice, codice, codice...
risp = input_sn("Vuoi continuare? (s/n) ")
```

3.16 Gestione delle eccezioni

Python (come tutti i linguaggi più evoluti) permette di gestire gli errori di runtime, facendo in modo che, quando accadono, vengano eseguite delle istruzioni definite dal programmatore (che di solito servono a correggere l'errore) ed il programma possa continuare.

[88]:
```
nomi = ["Gino", "Peppe", "Carlo", "Ciccio"]
try:
    n = int(input("Ho pensato quattro nomi. Quale vuoi sapere? "))
    n -= 1    # ricorda che gli indici vanno da 0 a 3
    print(nomi[n])
except:
    print("Input non valido!")
```

```
Ho pensato quattro nomi. Quale vuoi sapere? a
Input non valido!
```

[89]:
```
# Gestione degli errori in base al tipo di errore

nomi = ["Gino", "Peppe", "Carlo", "Ciccio"]
while True:
    try:
        n = int(input("Ho pensato quattro nomi. Quale vuoi sapere? "))
        n -= 1    # ricorda che gli indici vanno da 0 a 3
        print(nomi[n])
        break
    except ValueError:       # viene eseguita se abbiamo introdotto una stringa␣
 ↪senza senso
        print("Devi introdurre un numero!")
    except IndexError:       # viene eseguita se abbiamo introdotto un numero␣
 ↪sbagliato
        print("Hai introdotto un numero non valido!")
```

```
Ho pensato quattro nomi. Quale vuoi sapere? s
Devi introdurre un numero!
Ho pensato quattro nomi. Quale vuoi sapere? 4
Ciccio
```

L'istruzione raise

[90]:
```
raise NameError("Nome sbagliato")
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Input In [90], in <cell line: 1>()
----> 1 raise NameError("Nome sbagliato")
```

```
NameError: Nome sbagliato
```

3.17 List comprehension e filtri

Una list comprehension è racchiusa tra parentesi quadre; essa è formata da un'espressione, seguita da uno o più for ed (opzionalmente) da uno o più if. Quando incontra una comprehension Python crea automaticamente una lista (che sarà il risultato della comprehension) e vi aggiunge tutti i valori dell'espressione al suo interno calcolati eseguendo il ciclo for

[93]:
```
["aaa" + c for c in "wxyz"]
```

[93]:
```
['aaaw', 'aaax', 'aaay', 'aaaz']
```

[94]:
```
lista_alunni = [
        {"nome":"Mario", "cognome":"Rossi", "classe":"1A",
         "voti":{"Italiano":6, "Matematica":5, "Inglese":7 }},
        {"nome":"Lucia", "cognome":"Bianchi", "classe":"1B",
         "voti":{"Italiano":7, "Matematica":8, "Inglese":8 }},
        {"nome":"Luigi", "cognome":"Neri", "classe":"2A",
         "voti":{"Italiano":5, "Matematica":4, "Inglese":6 }},
        {"nome":"Laura", "cognome":"Verdi", "classe":"1A",
         "voti":{"Italiano":7, "Matematica":8, "Inglese":7 }},
        {"nome":"LPietro", "cognome":"Gialli", "classe":"1A",
         "voti":{"Italiano":7, "Matematica":8, "Inglese":5 }}
         ]
print(
    [(x["nome"], x["cognome"]) for x in lista_alunni if
                x["voti"]["Matematica"] < 6 ]
)
```

```
[('Mario', 'Rossi'), ('Luigi', 'Neri')]
```

3.18 Le Classi

3.18.1 Definizione

[96]:
```
class Human:
    # Un attributo della classe. E' condiviso tra tutte
    # le istanze delle classe
    species = "H. sapiens"

    # Si noti che i doppi underscore iniziali e finali denotano
    # gli oggetti o attributi utilizzati da Python ma che vivono
    # nel namespace controllato dall'utente.
    # Metodi, oggetti o attributi come: __init__, __str__,
    # __repr__, etc. sono chiamati metodi speciali (o talvolta
```

```
    # chiamati "dunder methods").
    # Non dovresti inventare tali nomi da solo.

    def __init__(self, name):
        # Assegna l'argomento all'attributo name dell'istanza
        self.name = name

        # Inizializza una proprietà
        self._age = 0

    # Un metodo dell'istanza. Tutti i metodi prendo "self"
    # come primo argomento
    def say(self, msg):
        print("{name}: {message}".format(name=self.name, message=msg))

    # Un altro metodo dell'istanza
    def sing(self):
        return 'yo... yo... microphone check... one two... '

    # Un metodo della classe è condiviso fra tutte le istanze
    # Sono chiamati con la classe chiamante come primo argomento
    @classmethod
    def get_species(cls):
        return cls.species

    # Un metodo statico è chiamato senza classe o istanza
    # di riferimento
    @staticmethod
    def grunt():
        return "*grunt*"

    # Una property è come un metodo getter.
    # Trasforma il metodo age() in un attributo in sola lettura,
    # che ha lo stesso nome
    @property
    def age(self):
        return self._age

    # Questo metodo permette di modificare una property
    @age.setter
    def age(self, age):
        self._age = age

    # Questo metodo permette di cancellare una property
    @age.deleter
    def age(self):
        del self._age
```

3.18.2 Uso delle classi

[97]:
```
if __name__ == '__main__':
    # Crea un'istanza della classe
    i = Human(name="Ian")
    i.say("hi")                     # "Ian: hi"
    j = Human("Joel")
    j.say("hello")                  # "Joel: hello"
    # i e j sono istanze del tipo Human, o in altre
    # parole sono oggetti Human

    # Chiama un metodo della classe
    i.say(i.get_species())          # "Ian: H. sapiens"
    # Cambia l'attributo condiviso
    Human.species = "H. neanderthalensis"
    i.say(i.get_species())          # => "Ian: H. neanderthalensis"
    j.say(j.get_species())          # => "Joel: H. neanderthalensis"

    # Chiama un metodo statico
    print(Human.grunt())            # => "*grunt*"

    # Non è possibile chiamare il metodo statico con l'istanza dell'oggetto
    # poiché i.grunt() metterà automaticamente "self" (l'oggetto i)
    # come argomento
    print(i.grunt())                # => TypeError: grunt() takes 0 positional␣
↪arguments but 1 was given

    # Aggiorna la property (age) di questa istanza
    i.age = 42
    # Leggi la property
    i.say(i.age)                    # => "Ian: 42"
    j.say(j.age)                   # => "Joel: 0"
    # Cancella la property
    del i.age
    i.age                           # => questo genererà un AttributeError
```

```
Ian: hi
Joel: hello
Ian: H. sapiens
Ian: H. neanderthalensis
Joel: H. neanderthalensis
*grunt*
*grunt*
Ian: 42
Joel: 0
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
```

```
Input In [97], in <cell line: 1>()
     30 # Cancella la property
     31 del i.age
---> 32 i.age

Input In [96], in Human.age(self)
     45 @property
     46 def age(self):
---> 47     return self._age

AttributeError: 'Human' object has no attribute '_age'
```

3.18.3 Ereditarietà

[100]:
```python
class Superhero(Human):
    # Se la classe figlio deve ereditare tutte le definizioni del
    # genitore  senza alcuna modifica, puoi semplicemente usare
    # la parola chiave "pass" (e nient'altro)
    # Le classi figlio possono sovrascrivere gli attributi dei loro genitori
    species = 'Superhuman'

    # Le classi figlie ereditano automaticamente il costruttore della classe
    # genitore, inclusi i suoi argomenti, ma possono anche definire ulteriori
    # argomenti o definizioni e sovrascrivere i suoi metodi (compreso  il
    # costruttore della classe).
    # Questo costruttore eredita l'argomento "nome" dalla classe "Human" e
    # aggiunge gli argomenti "superpowers" e "movie":
    def __init__(self, name, movie=False,
        superpowers=["super strength", "bulletproofing"]):

        # aggiungi ulteriori attributi della classe
        self.fictional = True
        self.movie = movie
        self.superpowers = superpowers

        # La funzione "super" ti consente di accedere ai metodi della classe
        # genitore che sono stati sovrascritti dalla classe figlia,
        # in questo caso il metodo __init__.
        # Il seguente codice esegue il costruttore della classe genitore:
        super().__init__(name)

    # Sovrascrivere il metodo "sing"
    def sing(self):
        return 'Dun, dun, DUN!'

    # Aggiungi un ulteriore metodo dell'istanza
    def boast(self):
```

```
        for power in self.superpowers:
            print("I wield the power of {pow}!".format(pow=power))


if __name__ == '__main__':
    sup = Superhero(name="Tick")

# Controllo del tipo di istanza
if isinstance(sup, Human):
    print('I am human')
if type(sup) is Superhero:
    print('I am a superhero')

# Ottieni il "Method Resolution search Order" usato sia da getattr ()
# che da super (). Questo attributo è dinamico e può essere aggiornato
print(Superhero.__mro__)    # => (<class '__main__.Superhero'>,
                            # => <class 'human.Human'>, <class 'object'>)

# Esegui il metodo principale ma utilizza il proprio attributo di classe
print(sup.get_species())    # => Superhuman

# Esegui un metodo che è stato sovrascritto
print(sup.sing())           # => Dun, dun, DUN!

# Esegui un metodo di Human
sup.say('Spoon')            # => Tick: Spoon

# Esegui un metodo che esiste solo in Superhero
sup.boast()                 # => I wield the power of super strength!
                                # => I wield the power of bulletproofing!

# Attributo di classe ereditato
sup.age = 31
print(sup.age)              # => 31

# Attributo che esiste solo in Superhero
print('Am I Oscar eligible? ' + str(sup.movie))
```

```
I am human
I am a superhero
(<class '__main__.Superhero'>, <class '__main__.Human'>, <class 'object'>)
Superhuman
Dun, dun, DUN!
Tick: Spoon
I wield the power of super strength!
I wield the power of bulletproofing!
31
```

```
Am I Oscar eligible? False
```

3.19 I decorator

I decorator possono essere considerati dei "wrapper" di una funzione o di una classe, il cui codice viene eseguito "prima" della funzione o classe stessa. Molto utile nei seguenti casi: - Aggiungere dei comportamenti inizialmente non previsti a delle funzioni o classi scritte da terzi, così da evitare un fork del progetto esterno e la conseguente modifica del codice - Riutilizzare questi comportamenti in più funzioni o classi

3.19.1 Definizione di un decorator

[105]:
```python
def function_decorator(func):
    def wrapped_func():
        # Fai qualcosa prima che la funzione venga eseguita
        print("Picere, sono il signor")
        # Richiama la funzione decorata
        func()
        # Fai qualcosa dopo l'esecuzione della funzione
        print("Piacere di conoscerla")
    return wrapped_func
```

- `function_decorator` è il nome del decorator
- `wrapped_func` è il nome della funzione annidata, usata solitamente all'interno del decorator per manipolare il comportamento della funzione/classe che stiamo decorando
- `func` è la funzione che abbiamo decorato

3.19.2 Utilizzo di un decorator

[107]:
```python
@function_decorator
def func():
    print("Mario Rossi")

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

```
Picere, sono il signor
Mario Rossi
Piacere di conoscerla
```

3.19.3 Caso più complesso: gli argomenti

[108]:
```python
def function_decorator(func):
    def wrapped_func(*args, **kwargs):
        print("Picere, sono il signor")
        func(*args, **kwargs)
        print("Piacere di conoscerla")
    return wrapped_func
```

```
@function_decorator
def func(nome):
    print(nome)

if __name__ == '__main__':
    func(input("Inserire il nome:"))
```

```
Inserire il nome:Asdf
Picere, sono il signor
Asdf
Piacere di conoscerla
```