Programmare in Python: controllo del flusso

Rights & Credits

Questo notebook è stato creato da Agostino Migliore, traendo e adattando diverse parti da un notebook di:

- Simone Campagna (CINECA)
- Mirko Cestari (CINECA)
- Nicola Spallanzani (CNR-NANO)

Costrutti di controllo del flusso

Indentazione

In Python l'indentazione non è arbitraria, in quanto ha un valore sintattico e il programma non funziona se l'indentazione non è corretta. Per esempio, l'esecuzione ripetuta di blocchi a diversi livelli avviene nel corretto ordine se tali blocchi sono opportunamente indentati. La convenzione adottata da linguaggio di programmazione Python è di aggiungere quattro spazi ad ogni livello di indentazione. Non usare il tasto Tab per indentare.

Costrutto condizionale: if-elif-else

if-elif-else è il costrutto condizionale. Il nome di tale costrutto deriva dalle parole chiave usate in esso: se (*if*), altrimenti se (*else if* in inglese, contratto nella parola chiava *elif*), altrimenti (else).

In [1]:
```
a , b = 1 , 3
```

In [4]:
```
if a == b:
    print(a)
elif a > b:
    print(b)
else:
    print(a + b)
```

4

Il costrutto può anche contenere più istruzioni (*statements*) `elif` nel mezzo. Ogni istruzione è seguita da `:`. In ciascun blocco di istruzioni (per esempio dopo un certo `elif`) tutte le linee di codice devono essere indentate alla stessa maniera e ciò viene fatto

automaticamente nel Jupyter notebook. Se due o più espressioni in un costrutto `if-elif-else` risultano soddisfatte, viene eseguita solo la prima di esse. Per esempio, in

In [5]:
```
if a <= b:
    print(a)
elif a == b/3:
    print(b)
else:
    print(a - b)
    print(a + b)
```

```
1
```

le espressioni di comparazione che seguono `if` ed `elif` sono entrambe soddisfatte; tuttavia, il progrmma esegue solo l'istruzione (oppure il set di istruzioni) dipendente dalla prima condizione che trova soddisfatta, in questo caso `print(a)`.

for loop

Il loop (in italiano: *ciclo continuo*) for consente di iterare comandi e operazioni su oggetti iterabili come liste, tuple, insiemi, range, ecc.

In [26]:
```
for i in range(4):
    print(i,i**2)
```

```
0 0
1 1
2 4
3 9
```

Si noti che, fornendo un solo argomento a `range`, l'iterabile parte di default da 0.

Gli oggetti su cui si esegue il ciclo `for` possonono anche essere eterogenei se le operazioni che si eseguno su di essi lo consentono.

In [27]:
```
t = ('a', 'b', 10, 3.4)
for i in t:
    print(i)
```

```
a
b
10
3.4
```

for loop: esempio 1

In [28]:
```
D = {'a': 0, 'b': 1, 'c': 2}
for key in D.keys():
    print(key)
```

```
a
b
c
```

for loop: esempio 2

In [29]:
```
for val in D.values():
    print(val)
```

```
0
1
2
```

for loop: esempio 3

In [30]:
```
for key, val in D.items():
    print(key, '=', val)
```

```
a = 0
b = 1
c = 2
```

for loop: esempio 4

In [37]:
```
a = 0
for i in range(6):
    a+=3
a
```

Out[37]: 18

Sul formato dei costrutti if e for

Se il corpo del blocco `if` o `for` contiene una sola istruzione, questa può essere scritta sulla stessa linea, come nell'esempio seguente.

In [37]:
```
k = 0
for i in range(7): k += 3
k
```

Out[37]: 21

for, if e range

L'uso di `range` all'interno di un loop `for` ci consente di comprendere la differenza tra contenitori (list, tuple, set e dict) e iterabili. Si consideri l'esempio seguente.

In [38]:
```
for i in range(100000000):
    if i < 4:
        print(i)
```

```
0
1
2
3
```

Se usassimo un numero molto più grande all'interno di `range` e la sequenza numerica prodotta dovesse essere scritta, l'intera memoria del computer sarebbe insufficiente per contenerla.

Quello di sopra è anche un esempio di uso combinato di `for` e `if`. Vediamone un altro.

In [41]:
```
L = {1, 2, 3, 4, 5, 7, 11, 12, 13}
```

In [54]:
```
n = 6
M = []
for i in L:
    if i >= n:
        M = M + [i]
M
```

Out[54]:
```
[7, 11, 12, 13]
```

while

while è un ciclo subordinato a una condizione, cioè il ciclo viene ripetuto fintantoché si verifica la condizione scritta dopo `while`.

In [51]:
```
i = 0
while i < 4:
    print(i)
    i += 1
```

```
0
1
2
3
```

Anche se il ciclo di sopra è sintatticamente corretto e quindi funziona, esso non rappresenta un uso intelligente di `while`. Infatti, abbiamo utilizzato `while` in un caso in cui era più semplice ottenere lo stesso risultato, con due sole righe, usando un ciclo `for`:

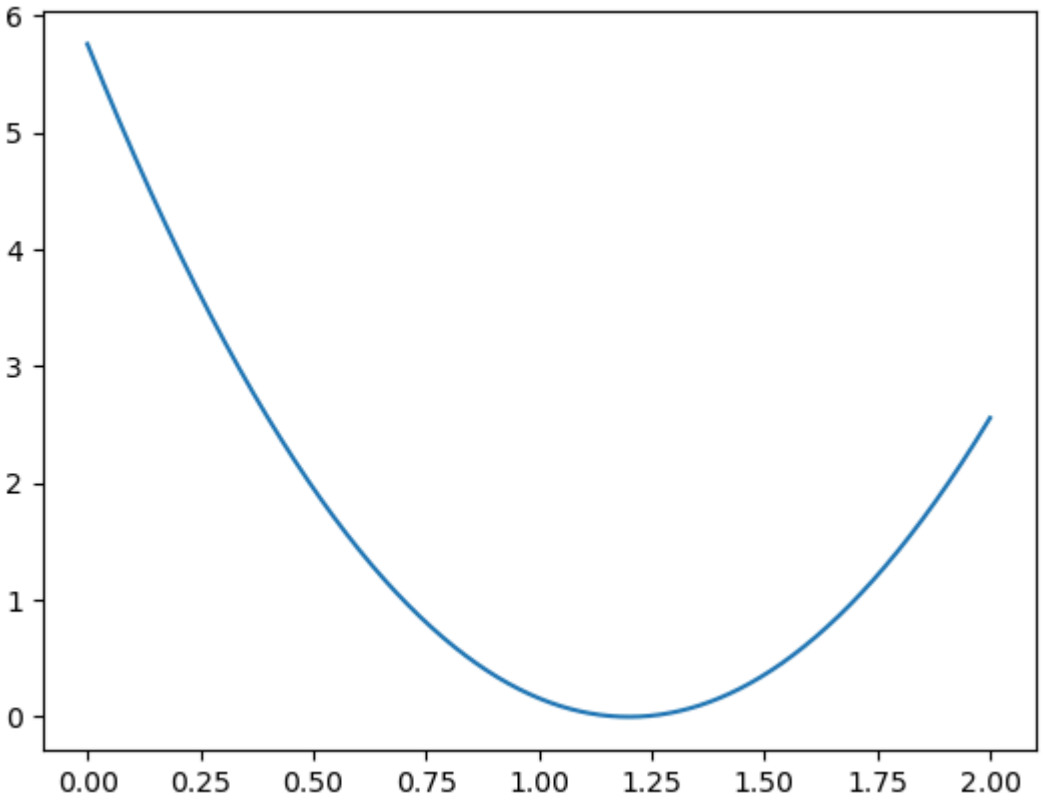
In [53]:
```
for i in range(4):
    print(i)
```

```
0
1
2
3
```

Il ciclo `while` è più appropriato quando c'è una condizione che va verificata in itinere. Si consideri l'esempio seguente, che consente anche di approfondire qualche altro punto.

In [18]:
```
import numpy as np
import matplotlib.pyplot as plt
# %matplotlib qt
# %matplotlib inline
def f(x):
    return 4*(x-1.2)**2
```

In [19]:
```
r = 0.0001
x = np.arange(0, 2, r)
y = f(x)
plt.plot(x,y)
plt.show()
```



In [20]:
```
i = 0
while f(x[i+1]) < f(x[i]):
    xm = x[i]
    ym = f(x[i])
    i += 1
print(xm,ym)
```

```
1.1999 3.9999999999991186e-08
```

In [19]:
```
import matplotlib
matplotlib.get_backend()
```

Out[19]:
```
'module://matplotlib_inline.backend_inline'
```

In [20]:
```
help(matplotlib.get_backend)
```

```
Help on function get_backend in module matplotlib:

get_backend()
    Return the name of the current backend.

    See Also
    --------
    matplotlib.use
```

pass

L'istruzione pass può essere usata come segnaposto (*placeholder*). Per esempio, assumiamo di scrivere un blocco di codice in cui usiamo un ciclo `for` e di volerlo interrompere senza inficiare il resto del codice.

In [24]:
```
l = 0
m = 11
L = []
for i in range(1000):

```

```
  Cell In[24], line 5

    ^
SyntaxError: incomplete input
```

Se eseguiamo un codice in cui sono contenute tali righe, otteniamo il messaggio di errore mostrato sopra. Infatti, abbiamo aperto un ciclo `for` e, nell'indentazione successiva, il programma si aspetta che venga immessa almeno una istruazione. Se, in tale linea, inseriamo `pass`, tale istruzione non fa nulla a parte consentire al programma di proseguire senza alcun (messaggio di) errore:

In [25]:
```
l = 0
m = 11
L = []
for i in range(1000):
    pass
```

Proviamo adesso ad usare `pass` come nel codice seguente:

In [34]:
```
l = 0
m = 11
L = []
for i in range(200000000):
    if l < m:
        L.append(l)
        l += 2
    else: pass
print(L)
```

```
[0, 2, 4, 6, 8, 10]
```

break

L'istruzione break consente di uscire da un ciclo. Se vi sono più cicli, permette di uscire da quello più interno.

In [35]:
```
for i in range(3):
    for j in range(4):
        k=i+j
        print(k)
        if k/2 == 1:
            break
```

```
0
1
2
1
2
2
```

In [36]:
```
l = 0
m = 11
L = []
for i in range(100000000):
    if l < m:
        L.append(l)
        l += 2
    else: break
print(L)
```

```
[0, 2, 4, 6, 8, 10]
```

continue

L'istruzione continue consente di saltare un'interazione in un ciclo e quindi continuare con quella successiva.

In [42]:
```
q = (1, 3, 4, 1, 5, 9, 1, 12)
for i in q:
    if i == 1:
        continue
    print(i)
```

```
3
4
5
9
12
```

Ciclo for con clausola else

La clausola else può apparire anche in un ciclo `for`. La clausola è eseguita soltanto dopo che il ciclo raggiunge la sua iterazione finale.

In [56]:
```
for i in q:
    if i > 4 and i<10:
        print(i)
else:
    print("nessun altro valore con tale proprietà trovato")
```

```
5
9
nessun altro valore con tale proprietà trovato
```

In [57]:
```
for i in q:
    if i > 5 and i<10:
        print(i)
        break
else:
    print("done!")
```

```
9
```

Confronto e operatori logici

Operatori di confronto

Gli operatori di confronto sono `==`, `!=`, `<`, `<=`, `>`, `>=`, `is` e `in`. I risultati delle corrispondenti relazioni di contronto sono uno dei due valori booleani `True` e `False`.

In [59]:
```
1 > 3
```

Out[59]:
```
False
```

Il risultato di una espressione di confronto, essendo un dato di tipo `bool`, può essere assegnato a una variabile.

In [63]:
```
b = 1 <= 4
b
```

Out[63]:
```
True
```

in

L'espressione `A` in `B` produce il valore logico `True` se `A` è contenuto in `B`, il valore `Falso` altrimenti.

In [64]:
```
lista = [3, 5, 7]
2 in lista
```

Out[64]:
```
False
```

is

L'espressione `A is B` restituisce `True` se `A` e `B` fanno riferimento allo stesso oggetto.

In [68]:
```
L1 = [1, 3, 12]
L2 = [1, 3, 12]
L3 = L1
L1 == L2, L1 is L2, L1 is L3
```

Out[68]:
```
(True, False, True)
```

In [70]:
```
id(L1), id(L2), id(L3)
```

Out[70]:
```
(2015904964480, 2015904793472, 2015904964480)
```

Si noti che, sopra, `=` assegna `L1` ad `L3`. In altre parole, `L1` è un riferimento ad un certo oggetto in memoria con identificativo `2015904964480` ed L3 viene assegnato come un altro riferimento allo stesso oggetto.

Operatori logici

Gli operatori logici sono `and` (già usato sopra), `or` e `not`. Essi agiscono tra valori logici, che possono anche essere i risultati di espressioni logiche. L'operatore and resituisce `True` se entrambi i valori logici sono veri, `False` altrimenti:

In [71]:
```
True and True, True and False, False and False
```

Out[71]:
```
(True, False, False)
```

In [73]:
```
1 < 4 and 5 > 2
```

Out[73]:
```
True
```

In [74]:
```
1 < 4 and 2 > 5
```

Out[74]:
```
False
```

L'operatore or dà `True` se almeno uno dei due oggetti è vero, `False` altrimenti:

In [75]:
```
True or True, True or False, False or False, 1 >= 3 or 7 < 12
```

Out[75]:
```
(True, True, False, True)
```

L'operatore not fornisce l'opposto di un certo valore logico:

In [77]:
```
not True, not False, not 3 > 1, not 1 > 3
```

Out[77]:
```
(False, True, False, True)
```

Per esempio, in linguaggio umano, l'ultima espressione logica si legge `non è vero che 1 sia maggiore di 3`, il che è `vero`.

Esercizio

- Scrivere un programma che stampi tutti i divisori di un numero *n* inizializzato nel programma stesso.
- Modificare il programma perché stampi solo i divisori di *n* che sono numeri primi.