Python: Funzioni e Moduli

Rights & Credits

Il notebook è stato creato da Agostino Migliore, sfruttando in varie parti materiali di notebooks di:

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

Funzioni

Una funzione è un programma che compie una specifica sequenza di operazioni (algoritmo) su variabili di input (gli argomenti che noi forniamo come valori dei parametri di ingresso), producendo l'output desiderato. Come anticipato,

- si dichiara una funzione con la parola chiave def, seguita dal nome della funzione, dalla lista di parametri da cui la funzione dipende racchiusa tra parentesi tonde e dal simbolo `:`, dopodiché il corpo della funzione verrà indentato; usando il jupyter notebook, ci accorgiamo immediatamente se ci siamo scordati di aggiungere `:`, perché l'indentazione è automatica quando la sintassi è corretta mentre non viene effettuata in tal caso;
- non occorre dichiarare i tipi delle variabili di ingresso e uscita;
- la funzione si chiude con un'istruzione return, che fornisce, rende, restituisce, ritorna (in inglese, appunto *return*) il risultato dell'algoritmo eseguito.

In [1]:
```
def summa(a, b):
    return a + b  # blocco di istruzioni (in questo caso una sola e quindi deve
                  # contenere "return") che costituisce il corpo della funzione
```

In [2]:
```
summa(3, 4)  # funziona con oggetti dello stesso tipo o di tipo compatibili
```

Out[2]:
```
7
```

In [3]:
```
print(summa(3.2, 4), summa(3.2 + 1.4J, 7), summa("Hello ", "world!"),
      summa([1, 2], [3, 4, 5]), sep="   ")
```

```
7.2   (10.2+1.4j)   Hello world!   [1, 2, 3, 4, 5]
```

In [4]:
```
summa(3, 'world!')  # non funziona con oggetti di tipi incompatibili
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[4], line 1
----> 1 summa(3, 'world!')

Cell In[1], line 2, in summa(a, b)
      1 def summa(a, b):
----> 2     return a + b

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Una funzione può essere ricorsiva:

In [5]:
```
def factorial(n):
    if n < 2:
        return 1
    else:
        return n * factorial(n - 1)
```

In [6]:
```
for i in (0, 3, 5, 130): print(factorial(i))
```

```
1
6
120
64668554892204736725073043955364852531553594478280496089759523229447819611855261655127070472292684529256839692403980271491207400740421058447377477994593100296357809917746129838031509651456000000000000000000000000000000000
```

Ogni funzione produce un valore. Di default, se non viene indicato nulla dopo `return`, tale valore è `None`. In altre parole, `return` da solo equivale a `return None`. Così,

In [7]:
```
def somma(a, b):
    S=a+b
    return
```

produce "niente". Chiedendo di stampare l'output della funzione otteniamo appunto `None`:

In [8]:
```
print(somma(3,4))
```

```
None
```

Esercizio

- Scrivere una funzione che produce tutti i divisori di un numero *n* dato in ingresso, a partire dal codice creato precedentemente.
- Scrivere una funzione che verifica se *n* è un numero primo o meno.

Passaggio (assegnazione) di argomenti a funzioni

Gli argomenti possono essere passati a una funzione per posizione o per nome (con keyword). Ovviamente, l'ordine in cui gli argomenti sono passati non importa se si procede

per nome, dal momento che i nomi chiave distinguono i parametri quale che sia il loro ordine di inserimento. Facciamo un esempio.

In [9]:
```
def count(L, val):
    c = 0
    for k in L:
        if k == val:
            c += 1
    return c
```

In [10]:
```
count([1,2,1,3,2,4,7,12,2,17], 2)  # assegnazione per posizione
```

Out[10]: 3

In [11]:
```
count(2,[1,2,1,3,2,4,7,12,2,17])  # assegnazione per posizione
                                  # con ordine di inserimento errato
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[11], line 1
----> 1 count(2,[1,2,1,3,2,4,7,12,2,17])

Cell In[9], line 3, in count(L, val)
      1 def count(L, val):
      2     c = 0
----> 3     for k in L:
      4         if k == val:
      5             c += 1

TypeError: 'int' object is not iterable
```

In [12]:
```
count(val=2, L=[1,2,1,3,2,4,7,12,2,17])  # assegnazione con keyword
```

Out[12]: 3

Dopo aver passato almeno un argomento per nome, i successivi non possono essere passati per posizione:

In [13]:
```
count([1,2,1,3,2,4,7,12,2,17], val=2)
```

Out[13]: 3

In [14]:
```
count(L=[1,2,1,3,2,4,7,12,2,17], 2)
```

```
  Cell In[14], line 1
    count(L=[1,2,1,3,2,4,7,12,2,17], 2)
                                     ^

SyntaxError: positional argument follows keyword argument
```

Se una funzione è costruita con valori di default per certi argomenti, allora essa può essere invocata omettendo tali argomenti:

In [15]:
```
def count_bis(L, val=3):
    c = 0
    for k in L:
        if k == val:
            c += 1
    return c
```

In [16]:
```
count_bis([1,2,1,3,2,4,7,12,2,17])   # lo stesso che chiamare la funzione
                                     # così: count_bis([1,2,1,3,2,4,7,12,2,17],3)
```

Out[16]: 1

Se una funzione è costruita in modo tale da avere un valore di default, si devono assegnare valori di default pure agli argomenti successivi.

In [17]:
```
def f(a, b = 0, c):
    print(a + b + c)
```

```
  Cell In[17], line 1
    def f(a, b = 0, c):
                    ^
SyntaxError: non-default argument follows default argument
```

Se gli argomenti da passare sono indicati con la notazione * (vedi esemio sotto), essi possono essere passati in numero (e tipo, compatibilmente con l'algoritmo da applicare) arbitrario, assegnandoli in modo posizionale. La funzione arrangerà tali argomenti in una tupla.

In [18]:
```
def summation(*args):
    S = sum(args)
    print(args)
    return S
```

In [19]:
```
summation(1,3,4)
```

```
(1, 3, 4)
```

Out[19]: 8

La cosa rilevante è il simbolo `*`. Nei manuali si fa seguire `*` da `args`, per richiamare che si tratta di argomenti, ma ciò non è necessario:

In [20]:
```
def summation_bis(*a):
    S = sum(a)
    return S
```

In [21]:
```
summation_bis(1,3,4)
```

Out[21]: 8

In [22]:
```
def summation_tris(*args):
    S=''
```

```
    for k in args:
        S += k
    return S
```

In [23]:
```
summation_tris('a','b','c')
```

Out[23]:
```
'abc'
```

In [24]:
```
def summation_tetra(S=None,*args):
    for k in args:
        S += k
    return S
```

In [25]:
```
summation_tetra(1,2,3)
```

Out[25]:
```
6
```

In [26]:
```
summation_tetra('a','b','c')
```

Out[26]:
```
'abc'
```

Si può avere pure una lista arbitraria di argomenti da passare per mezzo di chiave. Per indicare tale lista si usa la notazione **. In questo caso, la funzione collocherà gli argomenti in un `dict`.

In [27]:
```
def my_dict(**kwargs):
    print(kwargs)
```

In [28]:
```
my_dict(x=1,y=3,z=4)
```

```
{'x': 1, 'y': 3, 'z': 4}
```

Se nella lista di parametri di una funzione compare un elemento `/`, tutti gli argomenti precedenti devono per forza essere passati posizionalmente (positional-only arguments):

In [29]:
```
def rectangle_area(length,/,width):
    A = length * width
    return A
```

In [30]:
```
rectangle_area(length = 10,width = 30)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[30], line 1
----> 1 rectangle_area(length = 10,width = 30)

TypeError: rectangle_area() got some positional-only arguments passed as keyword arg
uments: 'length'
```

In [31]:
```
rectangle_area(10,30)
```

Out[31]:
```
300
```

In [32]:
```
rectangle_area(10, width = 30)
```

Out[32]:
```
300
```

Se nella lista di parametri di una funzione compare un elemento `*`, tutti gli argomenti successivi devono per forza essere passati con keyword (keyword-only arguments):

In [33]:
```
def rectangle_area_bis(length,*,width):
    A = length * width
    return A
```

In [34]:
```
rectangle_area_bis(10,30)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[34], line 1
----> 1 rectangle_area_bis(10,30)

TypeError: rectangle_area_bis() takes 1 positional argument but 2 were given
```

In [35]:
```
rectangle_area_bis(10)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[35], line 1
----> 1 rectangle_area_bis(10)

TypeError: rectangle_area_bis() missing 1 required keyword-only argument: 'width'
```

In [36]:
```
rectangle_area_bis(10, width = 30)
```

Out[36]:
```
300
```

Funzioni introspettive

Nel presente contesto di programmazione, introspezione è la capacità di esaminare le caratteristiche degli oggetti. In aggiunta alla capacità introspettiva di Python di rilevare il tipo di ciascun dato in fase di esecuzione, abbiamo già visto alcune funzioni che servono a fare introspezione.

In [37]:
```
z = 33.0 + 10j
```

In [38]:
```
type(z), id(z)
```

Out[38]:
```
(complex, 2115063655120)
```

Inoltre, come per ogni altro oggetto Python, possiamo ottenere informazioni su `z` mediante `help(z)`.

La funzione dir(), applicata a `z`, fornirà una lista degli attributi validi per tale oggetto (inclusi i possibili metodi):

In [39]:
```
dir(z)
```

Out[39]:
```
['__abs__',
 '__add__',
 '__bool__',
 '__class__',
 '__complex__',
 '__delattr__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getnewargs__',
 '__getstate__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__mul__',
 '__ne__',
 '__neg__',
 '__new__',
 '__pos__',
 '__pow__',
 '__radd__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__rmul__',
 '__rpow__',
 '__rsub__',
 '__rtruediv__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__sub__',
 '__subclasshook__',
 '__truediv__',
 'conjugate',
 'imag',
 'real']
```

Per esempio,

In [40]:
```
z.conjugate()
```

Out[40]:
```
(33-10j)
```

Se inseriamo una stringa esplicativa, ovvero di documentazione (doc string or docstring), in una funzione, per esempio

In [41]:
```
def factorial(n):
    "Questa funzione fornisce il fattoriale del numero dato."
    if n < 2:
        return 1
    else:
        return n * factorial(n - 1)
```

poi possiamo ottenere tale informazione sulla funzione tramite `help`:

In [42]:
```
help(factorial)
```

```
Help on function factorial in module __main__:

factorial(n)
    Questa funzione fornisce il fattoriale del numero dato.
```

oppure

In [43]:
```
factorial.__doc__
```

Out[43]:
```
'Questa funzione fornisce il fattoriale del numero dato.'
```

o ancora, per eliminare la virgolettatura,

In [44]:
```
print(factorial.__doc__)
```

```
Questa funzione fornisce il fattoriale del numero dato.
```

Moduli

Un modulo e' un file Python (quindi con estensione `.py`) che può contenere ogni tipo di codice Python, incluse istruzioni eseguibili e una o piu' funzioni. Importando un modulo con l'istruzione

```
import nome_modulo
```

(si noti che l'estensione `.py` non è inclusa nell'istruzione di sopra) si rendono disponibili le funzioni in esso contenute.
Usando un `alias`, si può ridurre la lunghezza del nome da digitare quando s'invoca una funzione del modulo. Per esempio:

```
import nome_modulo as nm
```

Assumiamo che il modulo abbia una struttura del tipo

```
def funzione_1(...):
    blocco di istruzioni 1
    return output_1

def funzione_2(...):
    blocco di istruzioni 2
    return output_2
```

Eseguendo il comando

`!type nome_modulo.py`

in una cella di un Jupyter notebook, il contenuto del modulo viene stampato sotto.
La linea di comando

`dir(nm)`

elenca tutte le proprietà e funzioni del modulo, includendo le funzioni create da noi, così come pure gli attributi e metodi intrinseci automaticamente conferiti al modulo in quanto oggetto Python. Una volta importato il modulo, possiamo eseguire le due funzioni create da noi con le seguenti istruzioni:

`nm.funzione_1(argomenti di input)`

`nm.funzione_2(argomenti di input)`

Se importiamo direttamente tali funzioni dal modulo, possiamo usarle senza bisogno di anteporre il nome (o l'alias) del modulo che le contiene. Per esempio, dopo avere eseguito

`from nome_modulo import funzione_1`

si può eseguire tale funzione sempliemente come segue:

`funzione_1(argomenti di input)`

Le due funzioni (e in generale tutte le funzioni di un modulo, tranne quelle il cui nome inizia con `_`) possono essere importate assieme tramite la riga di comando

`from nome_modulo import *`

Si possono raggruppare moduli in un pacchetto (*package*), che è generalmente costituito da un direttorio e sottodirettori con opportuna struttura gerarchica. Per essere classificato come `package`, un direttorio deve contenere un file di nome `__init__.py`.

Esercizio

- Creare un modulo `divisor_is_prime.py` contenente le due funzioni costruite nell'esercizio precedente e provare le istruzioni riportate sopra.

- Creare un programma che, dopo aver definito in qualche modo (un modo a scelta) una lista di numeri, la scorra sfruttando il modulo `divisor_is_prime.py` per verificare la natura di numero primo di ciascun elemento della lista e fornire in output la lista del sottoinsieme di numeri primi.