INDICE

- Le Funzioni
  - Definizione
  - Esecuzione
  - Passaggio di argomenti a una funzione
    - 1) per posizione
    - 2) per nome
    - 3) per posizione in numero variabile (*args)
    - 4) per nome in numero variabile (**kwargs)**
    - **3+4) funzioni che accettano un numero variabile di argomenti per posizione e per nome (*args**,kwargs)
  - Gli operatori di assegnamento 'splat' (o starred assignment)
  - Modifica del valore di una variabile passata come parametro ad una funzione (passaggio per valore/argomento)
  - Parametri di default
    - I parametri di default sono valutati nel momento in cui la funzione e' definita
    - Attenzione ai parametri di default mutable!!!
  - Il tipo di dato 'funzione'
  - Assegnazione del nome di una funzione ad una variabile
  - Passaggio di una funzione come parametro ad un'altra funzione
  - La ricorsione
  - Annidamento di funzioni
  - Attributi di una funzione
- Gli Scope (chi vede cosa)
  - Uso di variabili locali che hanno lo stesso nome di variabili globali
  - Uso di variabili globali in funzioni
  - Modifica di variabili globali in funzioni (uso della parola riservata global)
  - Uso di variabili nonlocali in funzioni
    - nonlocal
  - Risoluzione dei nomi: la regola LEGB
- Costrutti particolari
  - Funzioni che restituiscono funzioni (le chiusure)
  - Funzioni che accettano come parametri funzioni e restituiscono funzioni (decoratori)
    - Un primo esempio
    - un decoratore più generico
    - Un decoratore parametrizzato ...

Le Funzioni

Definizione

Il blocco di codice in cui si definisce una funzione ha la seguente struttura:

def nome_funzione(input1,input2): #corpo della funzione: qui si fa qualcosa ... ... [return valore_restituito #(anche nulla oppure più valori)]

In pratica:

In [1]:
```
#definisco una funzione che accetta un parametro e ritorna un valore
def quadrato(a):
    return a**2
```

Da notare:

- il blocco di codice inizia con **def**
- dopo **def** segue, sulla stessa riga **tra parentesi tonde, la lista dei parametri separati da virgole**

- segue,sulla stessa riga, il carattere **:**
- dopo i due punti segue **il corpo della funzione** che, in genere, e' a un (1) livello di indentazione superiore
- la funzione termina con l'istruzione **return**, seguita eventualmente dai valori restituiti
- Terminata la funzione si scende di un livello nell indentazione

Esecuzione

definire una funzione non vuol dire eseguirla!!

In [2]:
```
#definisco una funzione che accetta un parametro e ritorna un valore
def quadrato(a):
    return a**2
```

In [3]:
```
#qui eseguo la funzione
#in altre parole la uso
quadrato(3)
```

Out[3]:
```
9
```

Detta così sembra banale, ma la distinzione ha rilevanza specialmente quando messa in relazione ad esempio:

- alla definizione di argomenti con valore predefinito
- alla importazione di moduli

Passaggio di argomenti a una funzione

Abbiamo visto solo una tipologia di passaggio di argomenti a funzioni.
In realta' ne esistono 4 (come vedremo l'ordinamento della seguente lista e' significativo):

I. per posizione
II. per nome
III. per posizione in numero variabile
IV. per nome in numero variabile

In [4]:
```
def funz(a,b,c,d):
    """Funzione per prova passaggio argomenti
"""
    print('-'*15)
    print('a = ',a)
    print('b = ',b)
    print('c = ',c)
    print('d = ',d)
    return
```

1) per posizione

In [5]:
```
#si associa un valore ad ogni parametro in base all'ordine con cui passo gli argomenti:
funz(1,'ciao',3.55,[1,2,3,4])
```

```
---------------
a =  1
```

```
b =  ciao
c =  3.55
d =  [1, 2, 3, 4]
```

Per come e' definita la funzione devo sempre specificare 4 argomenti:

In [6]:
```
funz(1,2,3)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-6-2a2913c3dc54> in <module>()
----> 1 funz(1,2,3)

TypeError: funz() takes exactly 4 arguments (3 given)
```

2) per nome

ripeto la definizione della funzione x comodita':

In [ ]:
```
def funz(a,b,c,d):
    """Funzione per prova passaggio argomenti
"""
    print('-'*15)
    print('a = ',a)
    print('b = ',b)
    print('c = ',c)
    print('d = ',d)
    return
```

Uso la sintassi nomeparametro = argomento
In questo caso gli argomenti passati per nome possono essere ordinati a piacere:

In [ ]:
```
funz(a=33,b=3,c=2,d=1)
funz(d=1,c=2,b=3,a=33)
```

Posso anche mischiare il passaggio per posizione e quello per nome ma solo in modo ordinato: prima quelli per posizione e poi quelli per nome.

In [ ]:
```
#questo lo posso fare, prima per posizione, poi per nome
funz(1, 2, d=4, c=3)
```

In [ ]:
```
funz(1,2,a=4,c=3)
```

di contro non posso passare parametri per nome prima di parametri posizionali

In [ ]:
```
#questo non lo posso fare!!!
funz(a=1,2,3,4)
```

3) per posizione in numero variabile (*args)

Posso definire una funzione in modo che accetti un numero variabile di argomenti passati per posizione:

In [7]:
```
def funz2(*args):
    print('-'*15)
        arg    args:
```

```
    for      in
        print(arg)
```

vediamo un esempio di utilizzo:

In [8]:
```
funz2(1)
funz2(1,2,3)
funz2(1,[2,3],'ciao')
```

```
---------------
1
---------------
1
2
3
---------------
1
[2, 3]
ciao
```

La notazione *args e' uno standard in Python ma l'interprete accetterebbe qualsiasi nome dopo l'asterisco.

In [9]:
```
def funz3(*argomenti):
    print(type(argomenti))
    print('-'*15)
    for arg in argomenti:
        print(arg)

funz3(1,2,3,4,5,'ciao')
```

```
<class 'tuple'>
---------------
1
2
3
4
5
ciao
```

Python interpreta il nome che segue l'asterisco come una toupla!

Non posso passare nessun argomento per nome a una funzione come funz3!!!

In [10]:
```
funz3((2,3))
```

```
<class 'tuple'>
---------------
(2, 3)
```

4) per nome in numero variabile (**kwargs)

Uso il doppio asterisco nella definizione della funzione:

In [11]:
```
def funz4(**kwargs):
    print(type(kwargs))
    print(kwargs)
    return

funz4(a=1,b=2)
```

```
<class 'dict'>
{'a': 1, 'b': 2}
```

Quindi kwargs (o meglio il nome che segue il doppio asterisco) e' **un dizionario che contiene come chiavi i nomi dei parametri e come valori il valori del parametro associato**.

In [12]:
```
def funz4(**kwargs):
    print('-'*15)
    for key in kwargs:
        print(key,'=',kwargs[key])
```

In [13]:
```
funz4(a=1,b=2)
```

```
---------------
a = 1
b = 2
```

Non posso passare alcun argomento per posizione a funz4

In [14]:
```
funz4(1,b=2)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-14-f70b24174889> in <module>()
----> 1 funz4(1,b=2)

TypeError: funz4() takes exactly 0 positional arguments (2 given)
```

3+4) funzioni che accettano un numero variabile di argomenti per posizione e per nome (*args,**kwargs)

combinando le due ultime opzioni e' possibile definire funzioni che accettano un numero qualsiasi di argomenti sia per posizione sia per nome

In [15]:
```
def funz5(*args,**kwargs):
    print('-'*20)
    #itero sugli argomenti posizionali
    for i,arg in enumerate(args):
        print('posizionale n.{0} = {1}'.format(i+1,arg))
    #itero sugli argomenti passati per nome
    for key in kwargs:
        print(key,' = ',kwargs[key])
```

vediamo l'uso:

In [16]:
```
funz5(1)
```

```
--------------------
posizionale n.1 = 1
```

In [17]:
```
funz5(1,2,a=2)
```

```
--------------------
posizionale n.1 = 1
posizionale n.2 = 2
a  =  2
```

```
In [18]: funz5('ciao',1,a=4.3,b=[1,2],c={1:2,'a':3})
```

```
--------------------
posizionale n.1 = ciao
posizionale n.2 = 1
a  =  4.3
c  =  {'a': 3, 1: 2}
b  =  [1, 2]
```

Gli operatori di assegnamento 'splat' (o starred assignment)

In Python 3, la possibilità di lavorare con un numero indefinito di parametri è stata estesa anche all'operatore di assegnamento oltre che agli argomenti delle funzioni. Vediamo come:

```
In [19]: l = range(5)
         a,*b =  l
         print('a=',a)
         print('b=',b)
```

```
a= 0
b= [1, 2, 3, 4]
```

l'espressione di assegnamento sopra è equivalente a:

```
In [20]: a=l[0]
         b=list(l[1:])
         print('a=',a)
         print('b=',b)
```

```
a= 0
b= [1, 2, 3, 4]
```

l'operatore di assegnamento starred può essere usato nello stesso contesto anche nei seguenti modi

```
In [21]: *a,b = range(5)
         print('a=',a)
         print('b=',b)
```

```
a= [0, 1, 2, 3]
b= 4
```

```
In [22]: a,*b,c = range(5)
         print('a=',a)
         print('b=',b)
         print('c=',c)
```

```
a= 0
b= [1, 2, 3]
c= 4
```

```
In [23]: a,*b,c = range(10)
         print('a=',a)
         print('b=',b)
         print('c=',c)
```

```
a= 0
b= [1, 2, 3, 4, 5, 6, 7, 8]
c= 9
```

Modifica del valore di una variabile passata come parametro ad una funzione (passaggio per valore/argomento)

E' possibile modificare il valore di una variabile passata come parametro ad una funzione in modo che tale modifica permanga anche quando la funzione e' terminata?

A tale scopo alcuni linguaggi (come il C++) introducono la distinzione tra passaggio per valore e per riferimento.

Anche in python esiste questa distinzione anche se meno esplicita e più sottile.

Cio' nondimeno esiste una regola chiara riassumibile nei seguenti punti:

- I parametri passati a funzione sono riferimenti a oggetti e non gli oggetti stessi.
- se il parametro e' di tipo NON-MUTABLE la funzione (ma non solo lei) non potra' modificare in-place il valore della variabile (*).
- se il parametro e' di tipo MUTABLE la funzione puo' EVENTUALMENTE modificare in-place il valore della variabile in modo che tale modifica permanga una volta che la funzione e' terminata.

(*) sara' eventualmente possibile assegnare un nuovo oggetto al nome della variabile passata come parametro, ma per vedere questo dovremo riferirci al paragrafo dedicato alla **Modifica di variabili globali in funzioni**

vediamo, al solito, un esempio:

In [24]:
```
def prova_a_modificare(x,y,z):
    #associo una nuovo oggetto al nome x
    x = 23
    #modifico in-place l'oggetto associato al nome y
    y.append(2)
    #associo una nuovo oggetto al nome z
    z = [1,2,3]

    print('-'*20)
    print('x = ',x)
    print('y = ',y)
    print('z = ',z)
    print('-'*20)
```

usiamo la funzione definita sopra

In [25]:
```
a = 1      # int,  non-mutable
b = ['b']  # list, mutable
c = ['c']  # list, mutable

print('prima')
print('a = ',a)
print('b = ',b)
print('c = ',c)
prova_a_modificare(a,b,c)
print('dopo')
print('a = ',a)
print('b = ',b)
print('c = ',c)
```

```
prima
a =  1
b =  ['b']
c =  ['c']
--------------------
```

```
x =  23
y =  ['b', 2]
z =  [1, 2, 3]
--------------------
dopo
a =  1
b =  ['b', 2]
c =  ['c']
```

Parametri di default

I parametri di una funzione, possono anche avere un valore di default, che viene assunto quando non diversamente specificato:

In [26]:
```
def funz6(a=-1):
    print('a = ',a)
    return

#se non passo alcun argomento il parametro assume il valore di default
funz6()
funz6(10)
```

```
a =  -1
a =  10
```

I parametri di default sono valutati nel momento in cui la funzione e' definita

... e non ogni volta che la funzione e' eseguita.

Cio' e' evidente con il seguente esempio:

In [27]:
```
#definisco una variabile myx
myx=10

#il parametro di default viene valutato
#quando myx vale 10
def percinque(x=myx):
    return x*2

print(percinque())
#se adesso assegno un nuovo valore a myx
myx=1000
#
print(percinque())
```

```
20
20
```

Attenzione ai parametri di default mutable!!!

In relazione a quanto detto sopra si puo' capire perche' vale la raccomandazione che ai parametri di default dovrebbero essere assegnati oggetti NON mutable. Vediamo un esempio di cosa puo' succedere altrimenti

definisco una funzione che determina gli elementi positivi in una sequenza e li aggiunge ad una lista

In [28]:
```
def positivi(lista , pos=[]):
    for v in lista:
```

```
        if v>0:
            pos.append(v)
    return pos
```

La prima chiamata a positivi sembra funzionare bene:

In [29]:
```
positivi([-1,3,4,-22,0])
```

Out[29]: [3, 4]

ma la seconda ...

In [30]:
```
positivi([1000,-2,-3,55.2])
```

Out[30]: [3, 4, 1000, 55.2]

Questo accade perche' l'istruzione pos=[ ] non viene eseguita ogni volta che la funzione e' usata ma solo una volta, quando la funzione e' definita!!!

Nel caso illustrato il parametro di default e' una lista (MUTABLE) che pero' non viene svuotata ad ogni chiamata alla funzione. Piuttosto la lista mantiene memoria dei risultati ottenuti in tutte le chiamate precedenti.

Segue una possibile soluzione per risolvere il problema illustrato nel precedente esempio:

In [31]:
```
def positivi(lista , pos=None):
    if pos == None:
        pos =  []
    for v in lista:
        if v>0:
            pos.append(v)
    return pos

pos1 = positivi([-1,3,4,-22,0])
pos2 = positivi([1000,-2,-3,55.2])
pos3 = positivi([121.1,-22,-33,],pos2[:])#cosa succede se tolgo [:]??

print(pos1)
print(pos2)
print(pos3)
```

```
[3, 4]
[1000, 55.2]
[1000, 55.2, 121.1]
```

Il tipo di dato 'funzione'

Dopo la definizione di una funzione il nome della funzione si riferisce ad un' "istanza" della funzione.
In altre parole una funzione e' un tipo di dato predefinito.
Se si usa il nome di una funzione senza le parentesi l'interprete ci dice che ci stiamo riferendo al nome di una funzione

In [32]:
```
def quadrato(a):
    return a**2

#non sto usando la funzione!!!!
#sto semplicemente facendo riferimento al suo nome
quadrato
```

Out[32]:
```
<function __main__.quadrato>
```

In relazione a quanto detto sopra:

In [33]:
```
type(quadrato)
```

Out[33]:
```
builtins.function
```

Che e' diverso da:

In [34]:
```
type(quadrato(2))
```

Out[34]:
```
builtins.int
```

In [35]:
```
a=quadrato
a(2)
```

Out[35]:
```
4
```

Assegnazione del nome di una funzione ad una variabile

Essendo la funzione un tipo di dato, **posso assegnare una funzione (o meglio il suo nome) ad una variabile** e poi usare la variabile (o meglio la funzione alla quale la variabile si riferisce):

In [36]:
```
q = quadrato
```

è come se avessi attribuito un nuovo nome alla funzione:

In [37]:
```
q(7)
```

Out[37]:
```
49
```

in realtà l'attributo __name__ della funzione mantiene il riferimento al nome assegnato in fase di definizione.

In [38]:
```
q.__name__
```

Out[38]:
```
'quadrato'
```

Passaggio di una funzione come parametro ad un'altra funzione

posso passare una funzione come parametro ad un altra funzione:

In [39]:
```
def quadrato(a):
    return a**2


def applica_funzione(funzione, argomento):
    """stampa una stringa con
funzione(argomento)=risultato
restituisce None (nulla)
"""
    print('sto per usare la funzione {0}'.format(funzione.__name__))
    #determino il risultato
    risultato = funzione(argomento)
    #stampo la stringa con il risultato
```

```
    print('{0}({1})={2}'.format(funzione.__name__, argomento, risultato))
    print('fine')
    #termino la funzione
    return

a = applica_funzione(quadrato,6.234)
b = applica_funzione(abs,-4.546)
c = applica_funzione(len,range(5))
```

```
sto per usare la funzione quadrato
quadrato(6.234)=38.862756
fine
sto per usare la funzione abs
abs(-4.546)=4.546
fine
sto per usare la funzione len
len(range(0, 5))=5
fine
```

La ricorsione

In python è molto semplice implementare la ricorsione (una funzione che chiama se stessa).

Vediamo questa tecnica applicata ad un esempio di funzione che implementa l'elevazione a potenza fra numeri interi:

In [40]:
```
def potenzaint(x, y):
    if y == 0:
        return 1
    elif y > 0:
        print('potenzaint({0},{1})'.format(x,y))
        return x*potenzaint(x,y-1)
    else:
        print('potenzaint({0},{1})'.format(x,y))
        return 1/x*potenzaint(x,y+1)
```

In [41]:
```
potenzaint(2,4)
```

```
potenzaint(2,4)
potenzaint(2,3)
potenzaint(2,2)
potenzaint(2,1)
```

Out[41]:
```
16
```

In [42]:
```
potenzaint(2,-4)
```

```
potenzaint(2,-4)
potenzaint(2,-3)
potenzaint(2,-2)
potenzaint(2,-1)
```

Out[42]:
```
0.0625
```

Annidamento di funzioni

Il fatto che in una funzione possano essere utilizzate altre funzioni, definite altrove, lo abbiamo visto bene, anche negli esercizi.

Le funzioni pero' possono essere annidate cioe' definite una dentro un'altra, anche a più livelli.
A prima vista puo' sembrare poco utile, in realta' questa e' una pratica molto diffusa quindi e' bene saperla riconoscere.

In [43]:
```
#inizio definizione f1
def f1():
#####inizio definizione f2
    def f2():
        #questa funzione viene ridefinita ogni
        #volta che f1 viene utilizzata
        print('in f2')
        return
#####fine definizione f2
    print('in f1')
    #in f1 posso usare f2
    #f2 e' utilizzata qui
    #e non nel momento della definizione
    f2()
#fine definizione f1

#uso f1 e quindi anche f2
f1()
```

```
in f1
in f2
```

visto che f2 e' definita in f1 ne consegue che fuori da f1 NON posso usare f2

In [44]:
```
f2()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-44-fdec4c1c071f> in <module>()
----> 1 f2()

NameError: name 'f2' is not defined
```

approfondiremo la questione quando parleremo degli scope (sotto)

Attributi di una funzione

In un precedente esempio abbiamo usato l'attributo funzione.__name__ vediamo se ce ne sono altri"

In [45]:
```
dir(applica_funzione)
```

Out[45]:
```
['__annotations__',
 '__call__',
 '__class__',
 '__closure__',
 '__code__',
 '__defaults__',
 '__delattr__',
 '__dict__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__get__',
 '__getattribute__',
```

```
 '__globals__',
 '__gt__',
 '__hash__',
 '__init__',
 '__kwdefaults__',
 '__le__',
 '__lt__',
 '__module__',
 '__name__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__']
```

Indovinate cosa c'e' in applica_funzione.__doc__:

In [46]:
```
print(applica_funzione.__doc__)
```

```
stampa una stringa con
funzione(argomento)=risultato
restituisce None (nulla)
```

In [47]:
```
help(applica_funzione)
```

```
Help on function applica_funzione in module __main__:

applica_funzione(funzione, argomento)
    stampa una stringa con
    funzione(argomento)=risultato
    restituisce None (nulla)
```

dentro applica_funzione abbiamo usato funzione.__name__ cioe' l'attributo che contiene il nome della funzione:

In [48]:
```
quadrato.__name__
```

Out[48]:
```
'quadrato'
```

Posso anche aggiungere attributi alle funzioni:

In [49]:
```
quadrato.nota='non serve a nulla visto che c\'è l\'operatore unario **'
```

L'attibuto e' stato aggiunto:

In [50]:
```
'nota' in dir(quadrato)
```

Out[50]:
```
True
```

e inoltre lo si ritrova in:

In [51]:
```
quadrato.__dict__
```

Out[51]:
```
{'nota': "non serve a nulla visto che c'è l'operatore unario **"}
```

Tutte le funzioni hanno l'attributo __call__ che significa che possono essere 'chiamate' con la sintassi *nomefunzione(argomentifunzione)*

Gli Scope (chi vede cosa)

Per **'Scope'** di una variabile, si intende **l'ambito in cui la variabile puo' essere usata**

Lo scope di una variabile e' determinato da dove questa viene definita.
In particolare, in relazione ad una generica funzione:

- se una variabile e' definita nella funzione stessa si dice **locale alla funzione**
- se una variabile e' definita in una funzione che racchiude la funzione alla quale ci si riferisce (enclosing def) la funzione si dice **nonlocale alla funzione**
- se una variabile e' definita all'esterno di tutte le funzioni, e' detta variabile **globale**

La questione degli scope ha rilevanza a proposito:

- dove posso usare una variabile
- cosa succede se assegno lo stesso nome a variabili di contesti diversi
- dove Python cerca le variabili in base al loro nome ed al punto di definizione

vediamo cosa significa con degli esempi.

Uso di variabili locali che hanno lo stesso nome di variabili globali

definisco una funzione che, al suo interno definisce una variabile X

In [52]:
```
#
def func():
    #X e' locale alla funzione
    X = 'ciao'
    print('X = ',X)
```

vediamo cosa succede in uno script che definsce una variabile X e usa la funzione func vista sopra

In [53]:
```
#definisco una variabile X globale
X = 10
print('X = ',X)
func()
print('X = ',X)
```

```
X =  10
X =  ciao
X =  10
```

dentro func uso la X locale (X locale nasconde X globale) fuori da func uso X globale

Uso di variabili globali in funzioni

Se in una funzione si usa una variabile (in una espressione, a destra dell'operatore = ad esempio) che non e' stata definita nella funzione stessa l'interprete cerca la definizione della variabile all'esterno della funzione stessa.

In generale una variabile come quella descritta si dice una 'variabile libera' (free variable) nel contesto della funzione.

Nel seguente caso la funzione fun una variabile X che e' definita esternamente a tutte le funzioni.

In [54]:
```python
def fun(Y):
    #X e' globale anche se non esplicitato
    return Y + X

#variabile globale
X = 10
fun(1)#Y assume il valore 1 ed e' locale a fun
```

Out[54]:
```
11
```

Modifica di variabili globali in funzioni (uso della parola riservata global)

Nell'esempio precedente ho usato la variabile globale X.
Ma come fare se io voglio modificare una variabile globale all'interno di una funzione?
Considerando uno degli esempi precedenti, risulta evidente come non posso semplicemente digitare
X = nuovovalore
perche' in tal caso creerei una variabile locale che 'maschera' la variabile globale e non modificherei la variabile globale stessa.
Vediamo un altro esempio a proposito.

In [55]:
```python
#definisco una funzione che, al suo interno definisce una variabile X
def func():
    #X e' locale alla funzione
    X = 'ciao'
    print('X durante la chiamata a funzione = ',X)
```

In [56]:
```python
#definisco una variabile X globale
X = 10
#stampo X globale
print('X prima della chiamata a funzione = ',X)
#uso la funzione che al suo interno stampa la X a lei locale
func()
#stampo X globale
print('X dopo della chiamata a funzione = ',X)
```

```
X prima della chiamata a funzione =  10
X durante la chiamata a funzione =  ciao
X dopo della chiamata a funzione =  10
```

In questi casi si usa la keyword global che specifica esplicitamente che stiamo operando su una variabile globale.
L'esempio precedente e' modificato semplicemente aggiungendo 'global X' all'inizio della funzione.

In [57]:
```python
#definisco una funzione che, al suo interno definisce una variabile X
def func():
    #dichiaro che X e' quella globale!!!!!!!
    global X
    X = 'ciao'
    print('X durante la chiamata a funzione = ',X)
```

In [58]:
```python
#definisco una variabile X globale
X = 10
#stampo X globale
print('X prima della chiamata a funzione = ',X)
#uso la funzione che al suo interno stampa la X a lei locale
func()
```

```
#stampo X globale
print('X dopo della chiamata a funzione = ',X)
```

```
X prima della chiamata a funzione =  10
X durante la chiamata a funzione =  ciao
X dopo della chiamata a funzione =  ciao
```

NOTA: una variabile dichiarata global all'interno di una funzione non puo' essere anche uno dei parametri della funzione stessa.

In [59]:
```
def func(X):
    global X
    X = 'ciao'
    print('X durante la chiamata a funzione = ',X)
```

```
  File "<ipython-input-59-d0552e3e7b18>", line 1
SyntaxError: name 'X' is parameter and global
```

Uso di variabili nonlocali in funzioni

Si è già detto che una variabile non locale è una variabile definita in uno scope esterno alla funzione di riferimento ma comunque non a livello globale.

Il seguente esempio tratta di tre variabili X,Y,Z definite sia a livello globale, sia all'interno di due funzioni annidate.

La prima versione evidenzia la visibilità delle variabili all'interno delle funzioni.

Nella seconda versione dell'esempio si illustra l'effetto della parola riservata nonlocal.

In [60]:
```
#definisco una funzione con una funzione annidata
def funzione_esterna():

    def funzione_interna():
        #(8)
        X = 'X funzione interna'
        #(9),(10)
        print('-3-',X,Y,Z) # terza (in ordine di esecuzione) chiamata a print
        return # fine DEFINIZIONE funzione interna

    #(4)
    X = 'X funzione esterna'
    Y = 'Y funzione esterna'
    #(5),(6)
    print('-2-',X,Y,Z)# seconda chiamata a print
    #(7)
    funzione_interna()

    print('-4-',X,Y,Z)# quarta chiamata a print
    return #fine funzione esterna
```

In [61]:
```
#(1)
X = 'X globale'
Y = 'Y globale'
Z = 'Z globale'
#(2)
print('-1-',X,Y,Z)# prima chiamata a print
#(3)
funzione_esterna()
print('-5-',X,Y,Z)# prima chiamata a print
```

```
-1- X globale Y globale Z globale
-2- X funzione esterna Y funzione esterna Z globale
-3- X funzione interna Y funzione esterna Z globale
-4- X funzione esterna Y funzione esterna Z globale
-5- X globale Y globale Z globale
```

- (1) definisco le 3 variabili globali X,Y,Z
- (2) stampo le 3 variabili globali X,Y,Z (prima chiamata a print)
- (3) chiamo funzione esterna (non ci interessa qui specificare quando la funzione e' stata definita)
- (4) definisco le 2 variabili X e Y locali al contesto di chiamata
- (5) stampo le 3 variabili X,Y,Z (seconda chiamata a print)
- (6) le variabili stampate sono la Z globale e le X e Y locali
- (7) chiamo la funzione interna
- (8) definisco la variabile X locale
- (9) stampo le 3 variabili (terza chiamata a print)
- (10) le variabili stampate sono la Z globale la Y nonlocale (definita in funzione esterna) e la X locale

nonlocal

esiste una parola chiave (nonlocal) analoga a global che permette di dichiarare esplicitamente che si sta operando su una variabile libera non-locale e quindi modificarla.

In [62]:
```
#definisco una funzione con una funzione annidata
def funzione_esterna():

    def funzione_interna():
        #(8)
        nonlocal X
        X = 'X funzione interna' #agisco sulla X della funzione esterna!!!
        #(9),(10)
        print('-3-',X,Y,Z)
        return

    #(4)
    X = 'X funzione esterna'
    Y = 'Y funzione esterna'
    #(5),(6)
    print('-2-',X,Y,Z)
    #(7)
    funzione_interna()
    #(7)
    print('-4-',X,Y,Z)######### genera un risultato diverso dall'esempio precedente
    return
```

sotto si può vedere l'effetto dell'uso di nonlocal nella chiamata -4- a print

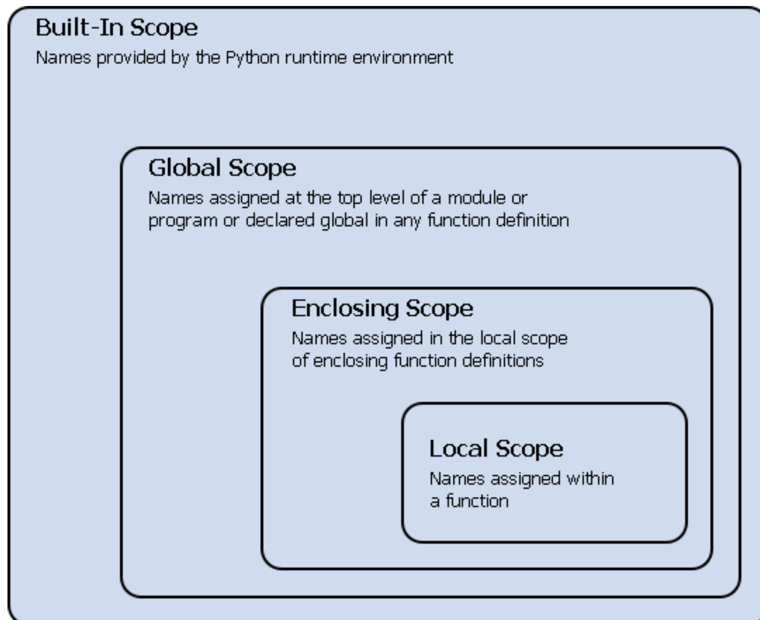
In [63]:
```
#(1)
X = 'X globale'
Y = 'Y globale'
Z = 'Z globale'
#(2)
print('-1-',X,Y,Z)# prima chiamata a print
#(3)
funzione_esterna()
print('-5-',X,Y,Z)# prima chiamata a print
```

```
-1- X globale Y globale Z globale
```

```
-2- X funzione esterna Y funzione esterna Z globale
-3- X funzione interna Y funzione esterna Z globale
-4- X funzione interna Y funzione esterna Z globale
-5- X globale Y globale Z globale
```

Risoluzione dei nomi: la regola LEGB

l'interprete quando incontra un nome ne cerca la definizione secondo il seguente schema, partendo dall'ambito + interno.



NOTA: questo vale anche per le funzioni del modulo builtin che possono essere 'mascherate' da versioni locali come nell'esempio sotto in cui 'maschero' localmente ad una funzione il nome print.

In [64]:
```
import os

def myscope():

    #definisco una funzione che 'maschera la builtin open
    def open(path,mode):
        print('scherzetto! Mi rifiuto di aprire il file {0} in mode
{1}'.format(path,mode))
        return

    #provo ad aprire un file in scrittura
    print('provo ad aprire un file in scrittura (in myscope)')
    open('prova.txt','wt')
    return
```

In [65]:
```
#Lo script parte da qui.
#Uso subito la funzione definita sopra.
myscope()
print('-'*20)
print('riprovo ad aprire un file in scrittura (nel global scope)')
fl=open('prova.txt','wt')
if not fl.closed:
    print('file aperto')
    fl.close()
```

```
provo ad aprire un file in scrittura (in myscope)
```

```
scherzetto! Mi rifiuto di aprire il file prova.txt in mode wt
--------------------
riprovo ad aprire un file in scrittura (nel global scope)
file aperto
```

Costrutti particolari

Finiamo la lezione con alcune note a costrutti che possono apparire 'esoterici' ma che in Python sono in realtà di utilizzo abbastanza comune.

Questi costrutti sfruttano il fatto che le funzioni possono essere passate come argomenti ed assegnate a variabili come un numero, una stringa o qualsiasi altro oggetto (funzioni come 'firs-class' objects).

Funzioni che restituiscono funzioni (le chiusure)

in realtà l'esempio che vi presento è più specifico rispetto al titolo del paragrafo: la seguente è una funzione che definisce e restituisce una funzione dove la funzione interna si riferisce ad una variabile definita nella funzione esterna ....

In [66]:
```
#definisco una funzione ...
def somma_a( x ):

    def sommainterna(y):
        return x+y
    # ... che restituisce una funzione
    return sommainterna
```

Alcune osservazioni

- La funzione (esterna) **somma_a**:
  - accetta come parametro una variabile x
  - restituisce una funzione, definita al suo interno, somma interna
- La funzione (interna) **sommainterna**:
  - accetta come parametro una variabile y
  - si riferisce alla variabile x (non locale) definita nella funzione somma_esterna (livello E di LEGB)
  - restituisce la somma di x + y

ripeto la definizione per comodità:

In [67]:
```
#definisco una funzione ...
def somma_a( x ):

    def sommainterna(y):
        return x+y
    # ... che restituisce una funzione
    return sommainterna
```

Vediamo come può essere utilizzata la nostra somma_a:

In [68]:
```
#definisco delle variabili
a5  = somma_a(5)
a13 = somma_a(13)
```

indovinate di che tipo sono a5 e a 13:

In [69]:
```
#che sono funzioni
type(a5),type(a13)
```

Out[69]:
```
(builtins.function, builtins.function)
```

ma allora posso usare a5 ed a13 come se fosserto funzioni:

In [70]:
```
a5(10),a5(25) , a13(10)
```

Out[70]:
```
(15, 30, 23)
```

Useremo questo tipo di costruzione in un esempio/esercizio più realistico e significativo nella prossima lezione sulle classi.

Funzioni che accettano come parametri funzioni e restituiscono funzioni (decoratori)

..... sempre più esoterico ......

Un primo esempio

Voglio aggiungere un azione prima e dopo la chiamata di una generica funzione che accetta zero argomenti in ingresso e restituisce qualsiasi cosa

In [71]:
```
def verbose(func):

    def wrapper():
        print('-'*20)
        print('prima di ',func.__name__)
        result = func()
        print('dopo ',func.__name__)
        print('*'*20)
        return result

    return wrapper
```

vediamo come posso utilizzare la mia funzione verbose applicandola ad una funzione f1():

In [72]:
```
#funzione bersaglio
def f1():
    print('in f1')
    return 0

#####################
#CAMBIO IL NOME ALLE MIA FUNZIONE f1
#O MEGLIO RIassegno il nome di ciascuna funzione
#  associandolo ad una nuova funzione che è la wrapper
# al cui interno viene richiamata f1

f2 = verbose(f1)
```

In [73]:
```
f1()
#####################
#uso la nuova f1
r1=f2()
```

```
#verifico che la funzione restituisca il valore corretto
print('valore restituito' ,r1)
```

```
in f1
--------------------
prima di  f1
in f1
dopo  f1
********************
valore restituito 0
```

posso applicare la mia verbose a qualsiasi funzione che non accetta parametri, senza replicare il codice:

In [74]:
```
def f2():
    print('in f2')
    return

def f3():
    print('in f3')
    return 'f3'

#'decoro le funzioni originarie
f2 = verbose(f2)
f3 = verbose(f3)

#uso le funzioni decorate
r2=f2()
r3=f3()

#verifico i valori restituiti
print('valori restituiti' ,r2,r3)
```

```
--------------------
prima di  f2
in f2
dopo  f2
********************
--------------------
prima di  f3
in f3
dopo  f3
********************
valori restituiti None f3
```

un decoratore più generico

uso questa tecnica per creare una funzione in grado di misurare il tempo impiegato da una generica funzione (che accetta qualsiasi numero e tipo di argomenti

In [75]:
```
import time

def addtimer( funzionedamisurare ):

    def wrapper(*args,**kwargs):
        t1  = time.clock()
        res = funzionedamisurare(*args,**kwargs)
        t2  = time.clock()
        print('La funzione {0} è stata eseguita in {1:.6f}
microsecondi'.format(funzionedamisurare.__name__,
```

```
(t2-t1)*1e6))
        return res

    return wrapper
```

creo una nuova versione di dir (ma non riassegno il suo nome):

In [76]:
```
#creo una nuova funzione 'decorata'
mydir = addtimer(dir)
```

La nuova dir funziona proprio come quella vecchia, salvo l'aggiunta della temporizzazione:

In [77]:
```
#uso la mia funzione senza argomenti
mydir()
```

```
La funzione dir è stata eseguita in 7.822223 microsecondi
```

Out[77]:
```
['args', 'funzionedamisurare', 'kwargs', 't1']
```

In [78]:
```
#uso la nuova funzione passandogli argomenti
a=1
mydir(a)
```

```
La funzione dir è stata eseguita in 54.476197 microsecondi
```

Out[78]:
```
['__abs__',
 '__add__',
 '__and__',
 '__bool__',
 '__ceil__',
 '__class__',
 '__delattr__',
 '__divmod__',
 '__doc__',
 '__eq__',
 '__float__',
 '__floor__',
 '__floordiv__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getnewargs__',
 '__gt__',
 '__hash__',
 '__index__',
 '__init__',
 '__int__',
 '__invert__',
 '__le__',
 '__lshift__',
 '__lt__',
 '__mod__',
 '__mul__',
 '__ne__',
 '__neg__',
 '__new__',
 '__or__',
 '__pos__',
 '__pow__',
 '__radd__',
```

```
 '__rand__',
 '__rdivmod__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__rfloordiv__',
 '__rlshift__',
 '__rmod__',
 '__rmul__',
 '__ror__',
 '__round__',
 '__rpow__',
 '__rrshift__',
 '__rshift__',
 '__rsub__',
 '__rtruediv__',
 '__rxor__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__sub__',
 '__subclasshook__',
 '__truediv__',
 '__trunc__',
 '__xor__',
 'bit_length',
 'conjugate',
 'denominator',
 'from_bytes',
 'imag',
 'numerator',
 'real',
 'to_bytes']
```

Un nuovo esempio con print

In [79]:
```
myprint = addtimer(print)
```

In [83]:
```
s='provastringa'
myprint(s)
#myprint(1000*s)
```

```
provastringa
La funzione print è stata eseguita in 13.130160 microsecondi
```

Un decoratore parametrizzato ...

... tanto per farsi del male ....

se voglio creare una versione di addtimer che accetta come parametro il numero di volte per cui chiamare la funzione da 'misurare' ...

In [81]:
```
import time

def parametric_timer(n):
    def addtimer( funzionedamisurare ):
        def wrapper(*args,**kwargs):
            t1  = time.clock()
```

```python
            for i in range(n-1):
                funzionedamisurare(*args,**kwargs)
            res = funzionedamisurare(*args,**kwargs)
            t2  = time.clock()
            print('La funzione {0} è stata eseguita per {2} volte in {1:.6f}
microsecondi'.format(funzionedamisurare.__name__,

(t2-t1)*1e6,
                                                                                          n))
            return res # restituito da wrapper

        return wrapper# restituito da addtimer
    return addtimer # restituito da parametric_timer
```

In [82]:
```python
mylen = parametric_timer(20)(len)

mylen([1,2,3,4,5])
```

```
La funzione len è stata eseguita per 20 volte in 6.984128 microsecondi
```

Out[82]:
```
5
```