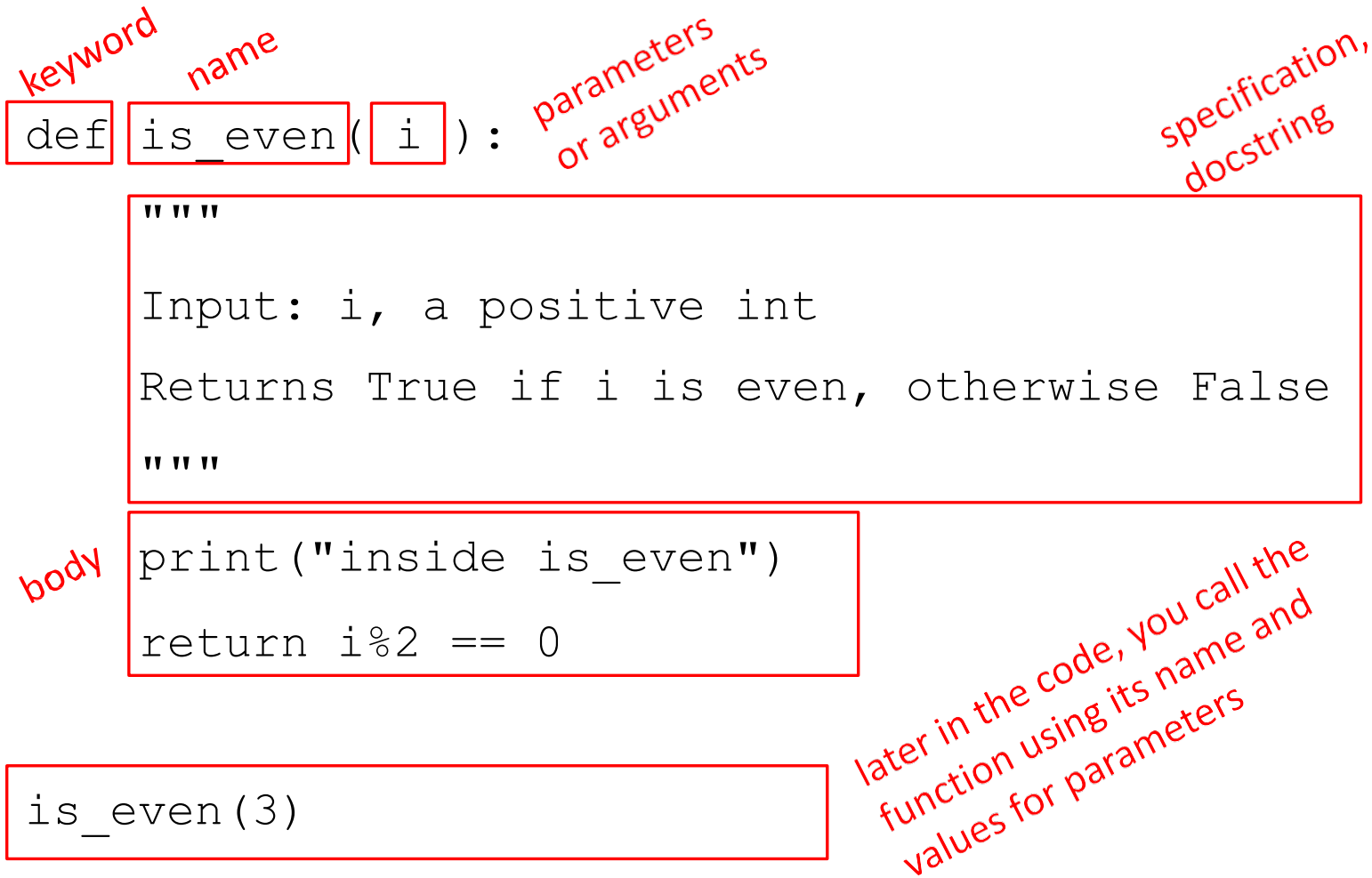
Funzioni

Funzioni in Python

- Le **funzioni** permettono di scrivere codice riusabile
- Le funzioni non vengono eseguite nel programma finché non vengono **chiamate** o **invocate**
- Caratteristiche di una funzione:
  - Ha un **nome**
  - Ha dei **parametri** (0 or più)
  - Ha una **docstring** (opzionale ma raccomandata)
    - La prima istruzione (commento) di una funzione che funge da documentazione
  - Ha un **corpo**
  - **Ritorna** qualcosa

Come scrivere e invocare una funzione in Python



Corpo della funzione

```
def is_even( i ):
    """
    Input: i, a positive int
    Returns True if i is even, otherwise False
    """
    print("inside is_even")
    return i%2 == 0
```

run some commands

keyword

expression to evaluate and return

Visibilità (scope) di una variabile

- **I parametri formali** sono legati al valore di un **parametro effettivo** quando la funzione viene invocata
- Viene creato un nuovo **scope/frame/environment** quando si entra in una funzione
- Lo **scope** è il mapping tra i nomi e gli oggetti

```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x

x = 3
z = f( x )
```

formal parameter

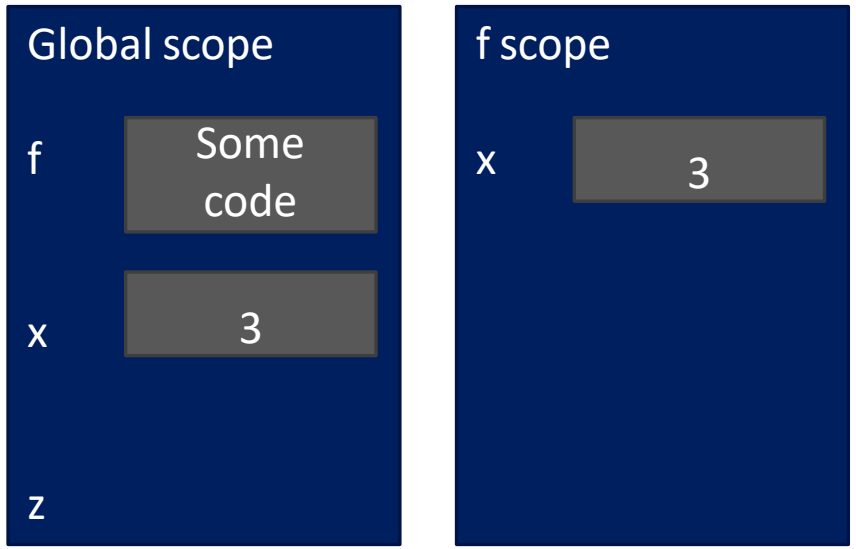
actual parameter

Function definition

Main program code
* initializes a variable x
* makes a function call f(x)
* assigns return of function to variable z

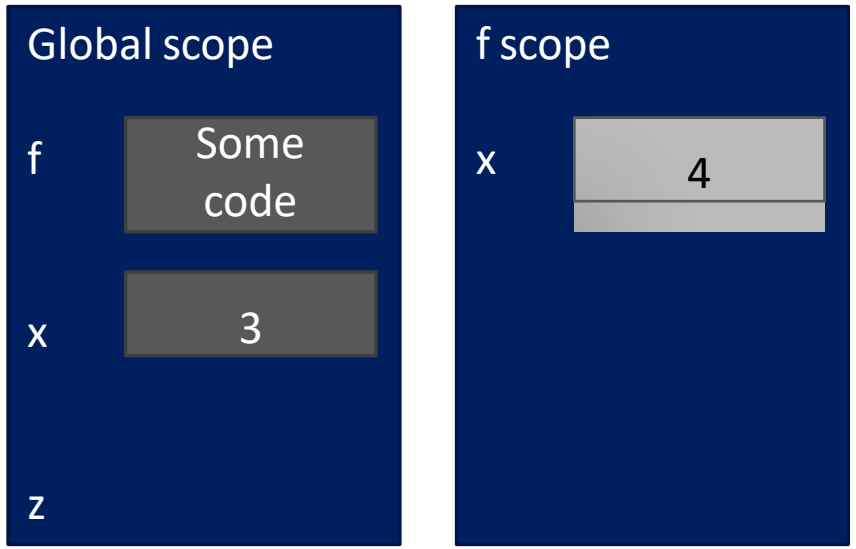
Visibilità (scope) di una variabile

```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x

x = 3
z = f( x )
```



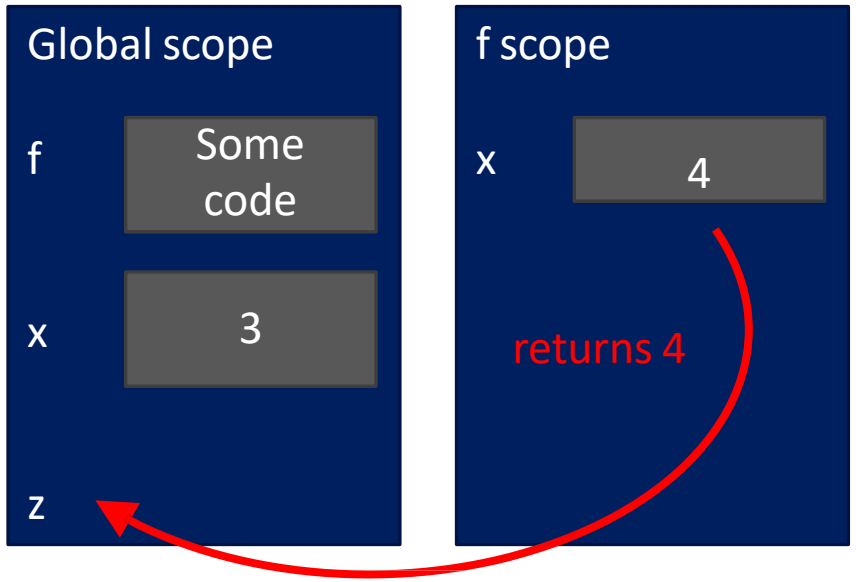
Visibilità (scope) di una variabile

```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x

x = 3
z = f( x )
```



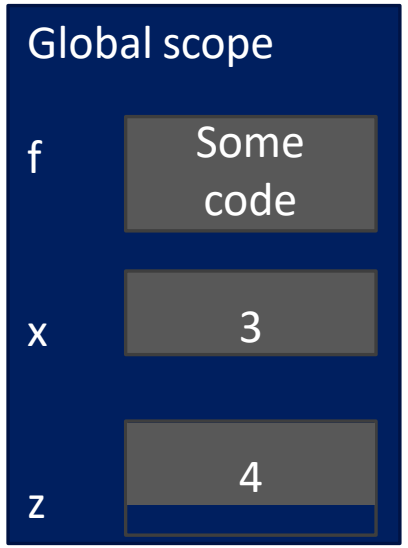
Visibilità (scope) di una variabile

```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x

x = 3
z = f( x )
```



Visibilità (scope) di una variabile

```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x

x = 3
z = f( x )
```



Istruzione `return` e il tipo None

```
def is_even( i ):
    """
    Input: i, a positive int
    Does not return anything
    """
    i%2 == 0
```

without a return statement

- Python ritorna il valore **None, se non si usa l'istruzione `return`!**
- None rappresenta l'assenza di valore

Funzioni come argomenti

- Gli argomenti di funzione possono essere di qualunque tipo, anche funzioni!

```
def func_a():
    print 'inside func_a'
def func_b(y):
    print 'inside func_b'
    return y
def func_c(z):
    print 'inside func_c'
    return z()
print func_a()
print 5 + func_b(2)
print func_c(func_a)
```

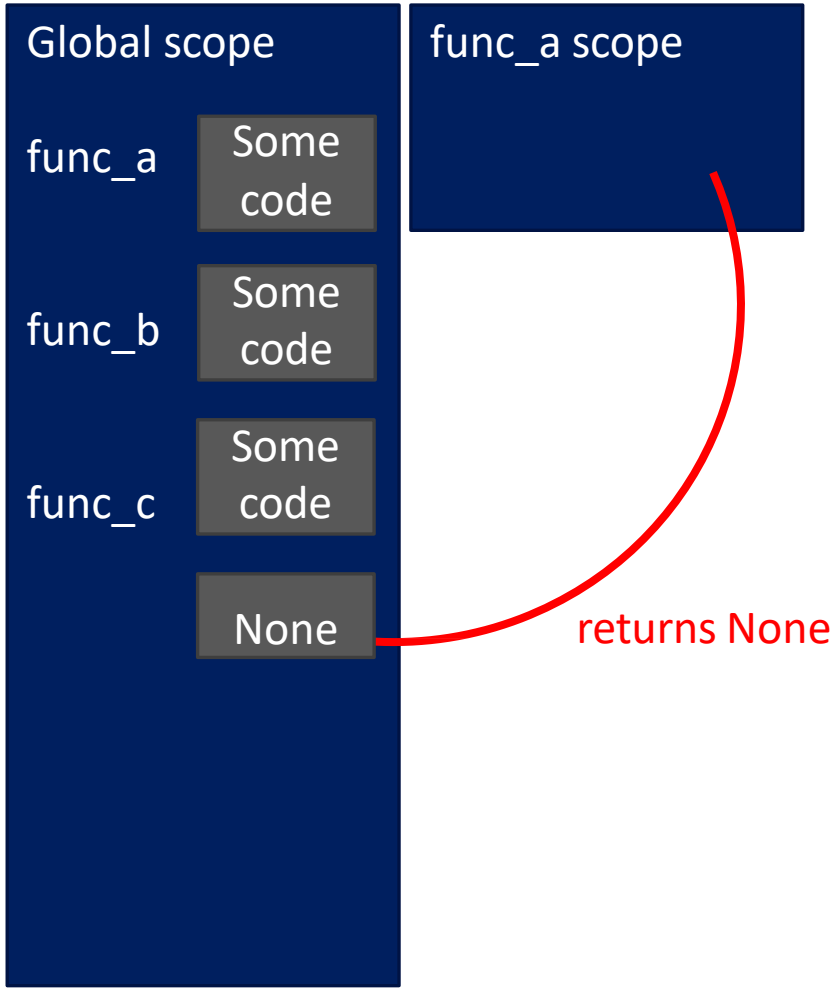
call `func_a`, takes no parameters

call `func_b`, takes one parameter

call `func_c`, takes one parameter, another function

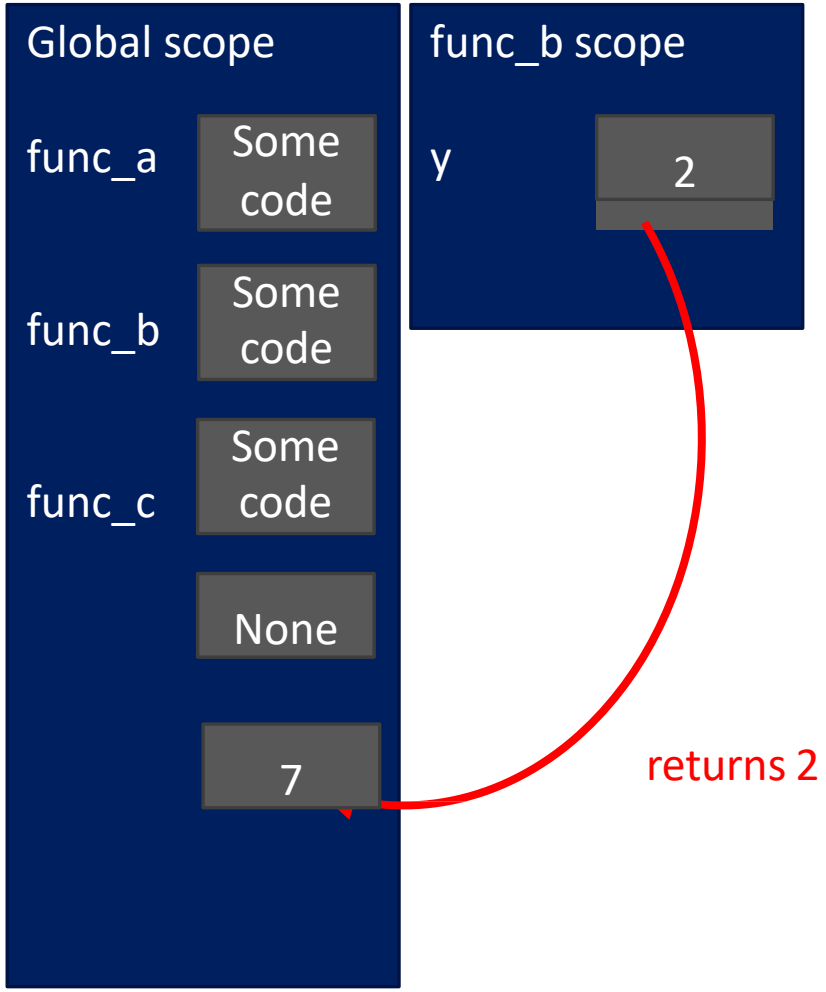
Funzioni come argomenti

```
def func_a():
    print 'inside func_a'
def func_b(y):
    print 'inside func_b'
    return y
def func_c(z):
    print 'inside func_c'
    return z()
print func_a()
print 5 + func_b(2)
print func_c(func_a)
```



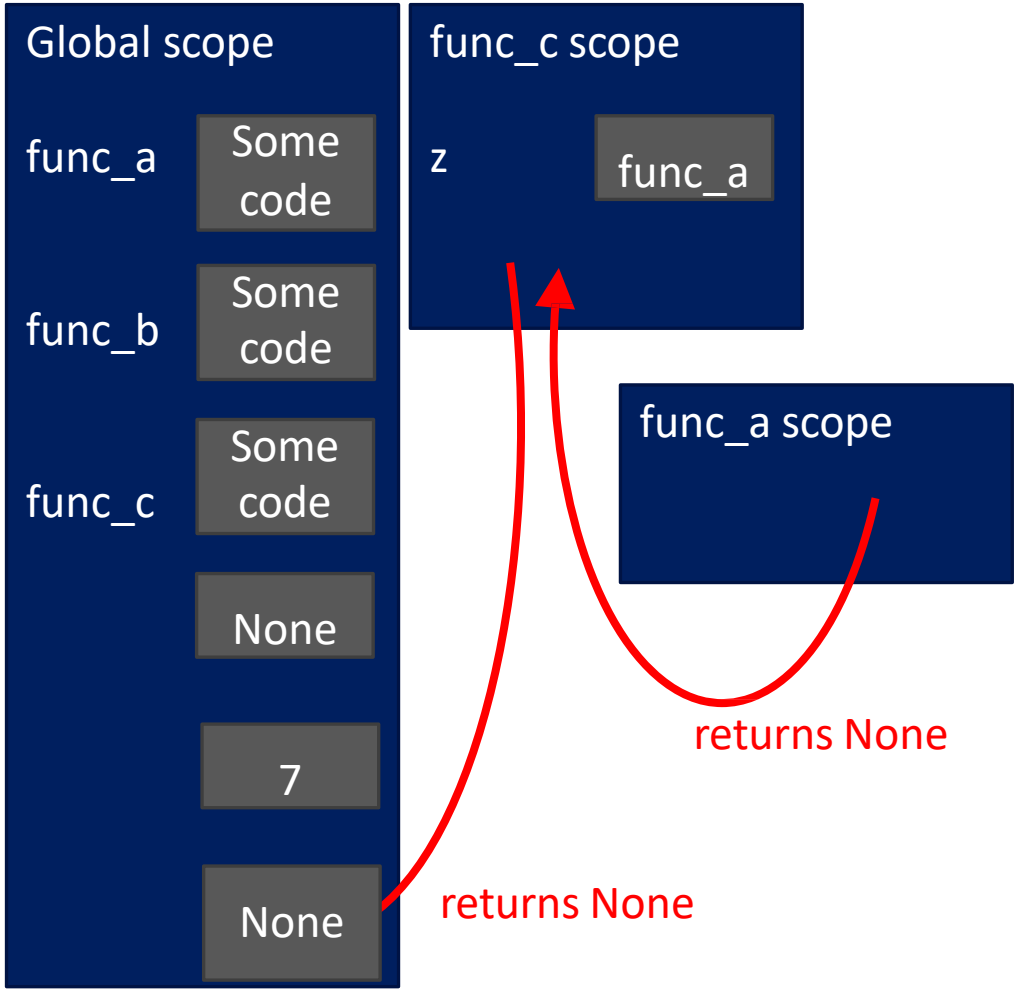
Funzioni come argomenti

```
def func_a():
    print 'inside func_a'
def func_b(y):
    print 'inside func_b'
    return y
def func_c(z):
    print 'inside func_c'

print func_a()
print 5 + func_b(2)
print func_c(func_a)
```



Funzioni come argomenti

```
def func_a():
    print 'inside func_a'
def func_b(y):
    print 'inside func_b'
    return y
def func_c(z):
    print 'inside func_c'
    return z()
print func_a()
print 5 + func_b(2)
print func_c(func_a)
```



Esempio di *scope*

- Dentro una funzione, **possiamo accedere** ad una variabile definita al di fuori della funzione
- Dentro una funzione, **non possiamo modificare** una variabile definita al di fuori della funzione – possiamo usare **variabili globali (direttiva `global` all'interno della funzione)**, ma non è una buona pratica

```
def f(y):
    x = 1
    x += 1
    print(x)

x = 5
f(x)
print(x)
```

`x` is re-defined in scope of `f`

different `x` objects

```
def g(y):
    print(x)
    print(x + 1)

x = 5
g(x)
print(x)
```

`x` from outside `g`

`x` inside `g` is picked up from scope that called function `g`

```
def h(y):
    x += 1

x = 5
h(x)
print(x)
```

UnboundLocalError: local variable 'x' referenced before assignment

Esempio di *scope*

- Dentro una funzione, **possiamo accedere** ad una variabile definita al di fuori della funzione
- Dentro una funzione, **non possiamo modificare** auna variabile definita al di fuori della funzione – possiamo usare **variabili globali**, ma non è una buona pratica

```
def f(y):
    x = 1
    x += 1
    print(x)

x = 5
f(x)
print(x)
```

```
def g(y):
    print(x)
    print(x + 1)

x = 5
g(x)
print(x)
```

```
def h(y):
    x += 1

x = 5
h(x)
print(x)
```

x from global/main program scope

Esempio complesso sullo *scope*

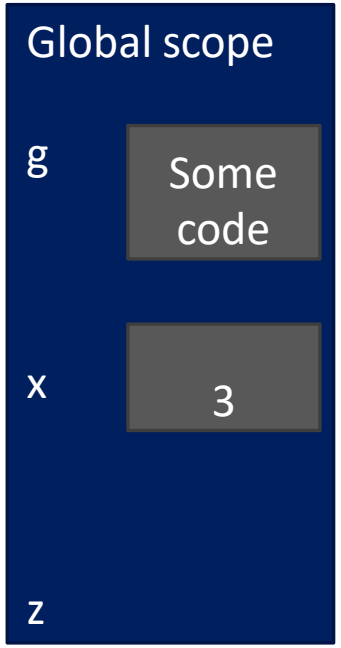
Python Tutor è il tuo miglior amico!

http://www.pythontutor.com/

Esempio complesso sullo *scope*

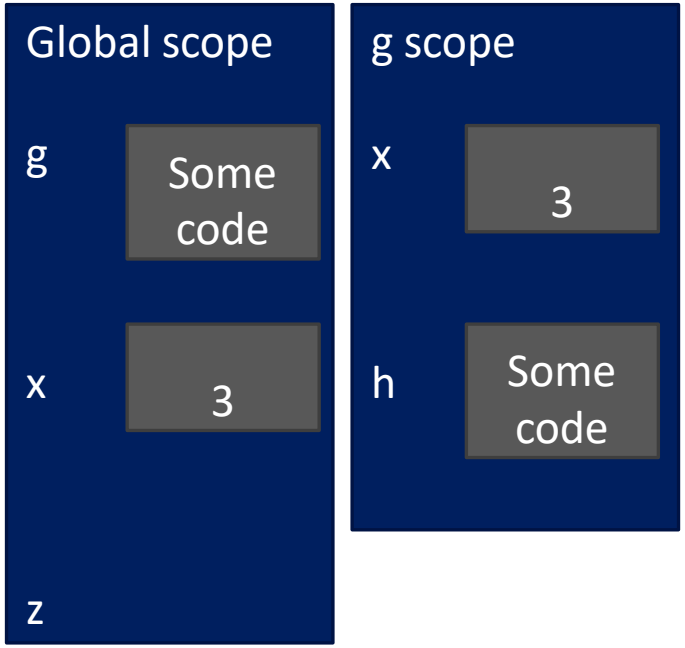
```
def g(x):
    def h():
        x = 'abc'
    x = x + 1
    print('g: x =', x)
    h()
    return x

x = 3
z = g(x)
```

Some code



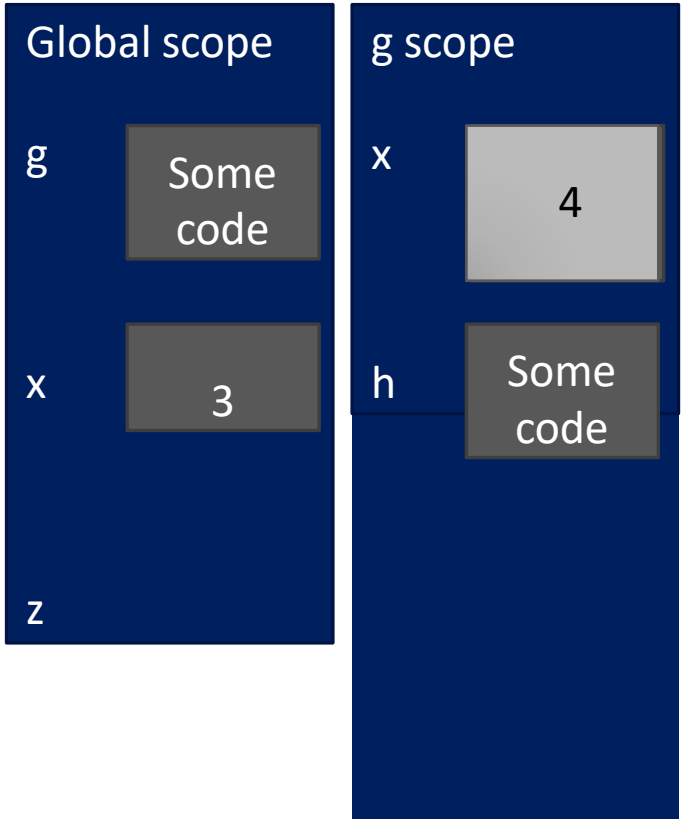
Esempio complesso sullo *scope*

```
def g(x):
    def h():
        x = 'abc'
    x = x + 1
    print('g: x =', x)
    h()
    return x

x = 3
z = g(x)
```



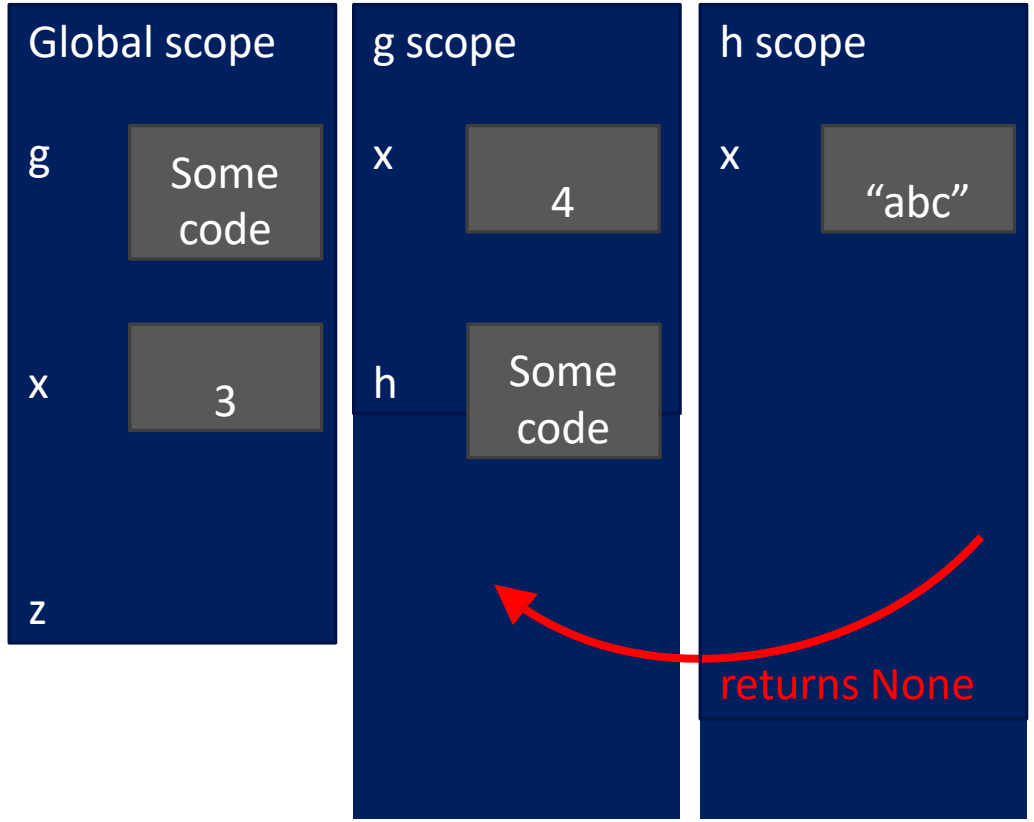
Esempio complesso sullo *scope*

```
def g(x):
    def h():
        x = 'abc'
    x = x + 1
    print('g: x =', x)
    h()
    return x

x = 3
z = g(x)
```



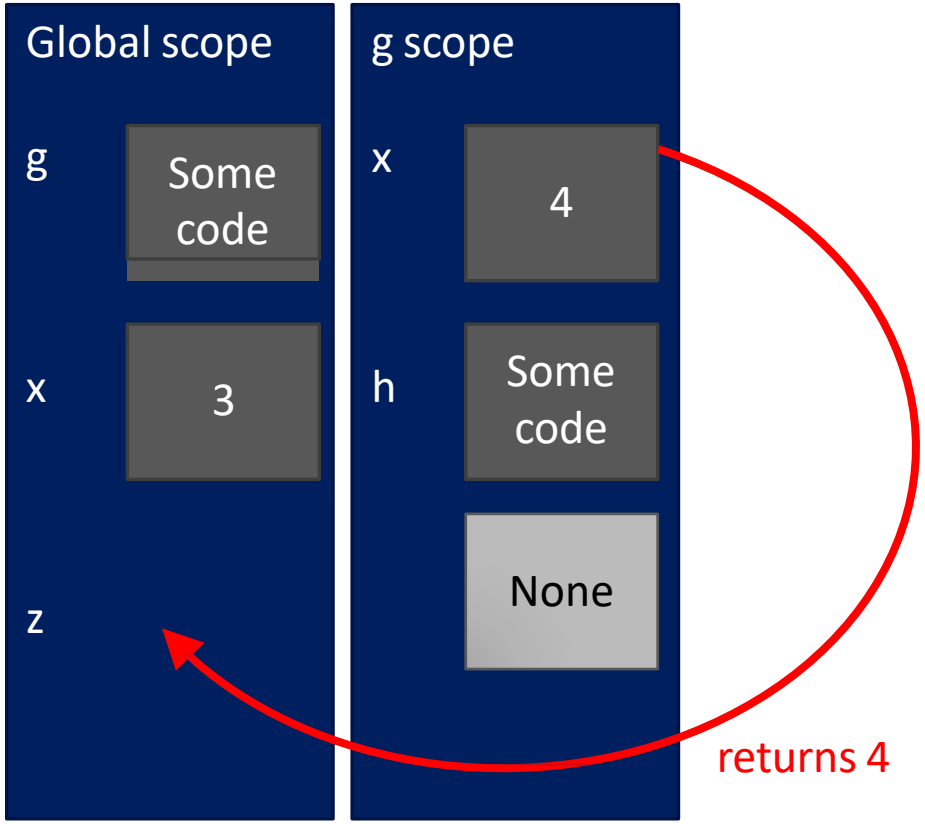
Esempio complesso sullo *scope*

```
def g(x):
    def h():
        x = 'abc'
    x = x + 1
    print('g: x =', x)
    h()
    return x

x = 3
z = g(x)
```



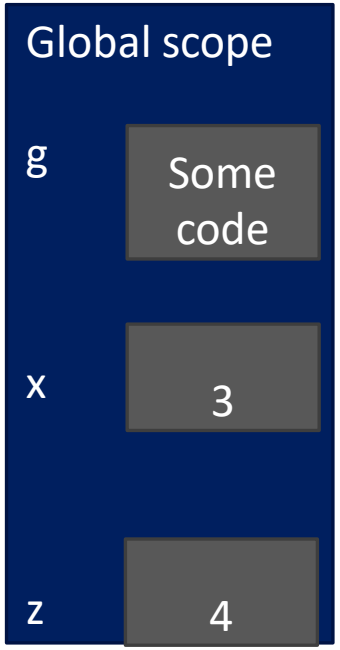
Esempio complesso sullo *scope*

```
def g(x):
    def h():
        x = 'abc'
    x = x + 1
    print('g: x =', x)
    h()
    return x

x = 3
z = g(x)
```



Esempio complesso sullo *scope*

```
def g(x):
    def h():
        x = 'abc'
    x = x + 1
    print('g: x =', x)
    h()
    return x

x = 3
z = g(x)
```



Passaggio dei parametri in Python

- Tutti i parametri (argomenti) in Python sono ***passati per riferimento***
- Ciò significa che se si cambia il valore di un parametro all'interno di una funzione, la **modifica si riflette anche nella funzione chiamante**
- **Nota bene:**
  - Se si ri-assegna la variabile all'interno dello scope locale la **modifica rimane locale**
  - Bisogna usare la direttiva **global** all'interno dello scope locale per far riflettere la modifica

Passaggio dei parametri in Python: esempio

- Se ridefinisco l'oggetto passato come parametro alla funzione, nello scope locale la funzione lavora su un oggetto diverso anche se con lo stesso nome!

```
def fun(x):
        x = x + 1
        print("[fun scope]\t x value: ", x, " x ref: ", hex(id(x))) #
        id è una funzione che stampa la ref. di un oggetto
        return

x = 10
print("Before call [global scope]\t x value: ", x, " x ref: ", hex(id(x)))
fun(x);
print("After call [global scope]\t x value: ", x, " x ref: ", hex(id(x)))
```

Risultato:

```
Before call [global scope]        x value: 10 x ref: 0x10033c210
[fun scope]                       x value: 11 x ref: 0x10033c230
After call [global scope]         x value: 10 x ref: 0x10033c210
```

Passaggio dei parametri in Python: esempio

- Se utilizzo **`global`**, nello scope locale la funzione lavora sullo stesso oggetto (riferimento) dello scope globale

```
def fun():
        global x
        x = x + 1
        print("[fun scope]\t\t\t x value: ", x, " x ref: ", hex(id(x))) # id
        è una funzione che stampa la ref. di un oggetto
        return

x = 10
print("Before call [global scope]\t x value: ", x, " x ref: ", hex(id(x)))
fun();
print("After call [global scope]\t x value: ", x, " x ref: ", hex(id(x)))
```

Risultato:

```
Before call [global scope]  x value:  10  x ref:  0x10033c210
[fun scope]                 x value:  11  x ref:  0x10033c230
After call [global scope]   x value:  11  x ref:  0x10033c230
```