

[FIM] FONDAMENTI DI INFORMATICA per medicina e chirurgia high tech

L03: Funzioni

Dott. Giorgio De Magistris

demagistris@diag.uniroma1.it

CORSO DI LAUREA IN MEDICINA E CHIRURGIA HIGH TECH



SAPIENZA
UNIVERSITÀ DI ROMA

I3S

FACOLTÀ DI INGEGNERIA DELL'INFORMAZIONE, INFORMATICA E STATISTICA

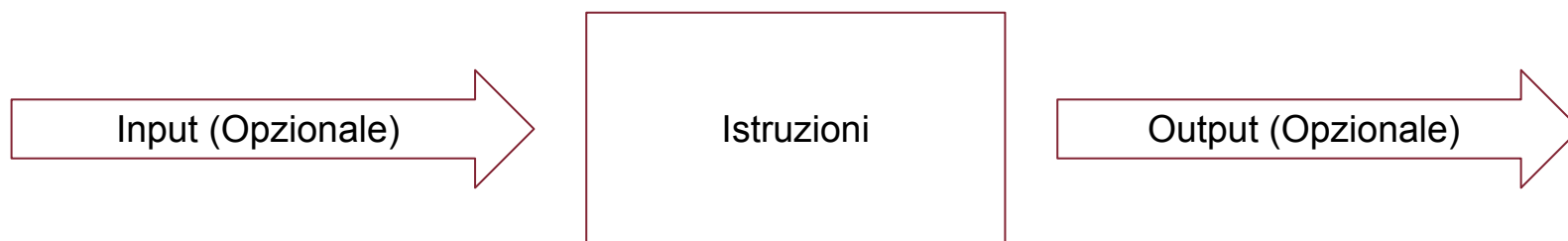
DIAG

DIPARTIMENTO DI INGEGNERIA INFORMATICA, AUTOMATICA E GESTIONALE

TUTTI I DIRITTI RELATIVI AL PRESENTE MATERIALE DIDATTICO ED AL SUO CONTENUTO SONO RISERVATI A SAPIENZA E AI SUOI AUTORI (O DOCENTI CHE LO HANNO PRODOTTO). È CONSENTITO L'USO PERSONALE DELLO STESSO DA PARTE DELLO STUDENTE A FINI DI STUDIO. NE È VIETATA NEL MODO PIÙ ASSOLUTO LA DIFFUSIONE, DUPLICAZIONE, CESSIONE, TRASMISSIONE, DISTRIBUZIONE A TERZI O AL PUBBLICO PENA LE SANZIONI APPLICABILI PER LEGGE

Funzioni

- Gruppo di istruzioni all'interno di un programma che svolgono uno specifico task



- Input e Output

```
def sum(a,b):  
    return a + b
```

- Solo Input (Side-Effect)

```
def insert_list_at(l,idx,sublist)  
    l[idx:idx] = sublist
```

- Nè input nè output

```
def greetings():  
    print("hello")
```

Terminologia

Header

```
def function_name(arg1:type1, arg2:type2,  
arg3=0:int) -> return_type:
```

```
    """
```

```
    this is a function that receives some  
input and returns some output
```

```
    """
```

Documentazione (da
non dimenticare)

```
    instruction
```

```
    instruction
```

```
    instruction
```

```
    ...
```

```
    return return_value
```

Body

Import dei moduli
(vedi dopo)

```
import math  
import random
```

Definizione di
Funzione

```
def mean(l:List[float]) -> float:  
    s = sum(l)  
    m = s / len(l)  
    return m
```

```
def standard_deviation(l:List[float]) -> float:  
    m = mean(l)  
    s = .0  
    for e in l:  
        s += math.pow(m-e,2)  
    return math.sqrt(s/len(l))
```

```
if __name__=="__main__":
```

```
    a = [random.random() for x in range(10)]
```

```
    sd = standard_deviation(a)
```

```
    print(sd)
```

Chiamata di
Funzione

Perchè usare le Funzioni

- **Modularità:** posso usare una funzione semplicemente guardando la segnatura, senza capirne il contenuto
- **Leggibilità:** il codice risulta più leggibile e ben organizzato
- **Riusabilità:** posso invocare una funzione più volte senza dover ripetere il codice
- **Manutenibilità:** posso cambiare l'implementazione di una funzione senza cambiare il resto del programma (a patto di usare la stessa segnatura)

```
import math
import random
```

```
if __name__=="__main__":
```

```
    a = [random.random() for x in range(10)]
```

```
    s = sum(a)
```

```
    mean = sum(a) / len(a)
```

```
    dev = 0.
```

```
    for e in a:
```

```
        dev += (mean-e)**2
```

```
    std_a = math.sqrt(dev/len(a))
```

Stesso
codice
ripetuto

```
    b = [random.random() for x in range(100)]
```

```
    s = sum(b)
```

```
    mean = sum(b) / len(b)
```

```
    dev = 0.
```

```
    for e in b:
```

```
        dev += (mean-e)**2
```

```
    std_b = math.sqrt(dev/len(b))
```

```
print("std_a:{}\nstd_b:{}\n".format(std_a,std_b))
```


Esempio

- Programma che prende in input due interi e stampa la somma dei valori assoluti
- Lo stesso codice può essere scritto con o senza l'utilizzo di funzioni

```
if __name__ == "__main__":  
  
    a = int(input("Inserisci un intero a:"))  
    b = int(input("Inserisci un intero b:"))  
  
    if a < 0:  
        a = -a  
    if b < 0:  
        b = -b  
  
    s = a + b  
    print("|a| + |b| = {}".format(s))
```

```
def abs(a):  
    if a >= 0:  
        return  
    else:  
        return -a  
  
if __name__ == "__main__":  
  
    a = int(input("Inserisci un intero a:"))  
    b = int(input("Inserisci un intero b:"))  
  
    s = abs(a) + abs(b)  
    print("|a| + |b| = {}".format(s))
```

Codice Equivalente



Chiamata di Funzione

- L'esecuzione di un programma parte dalla prima istruzione dello script (riga 12) e prosegue in modo sequenziale
- Quando viene chiamata una funzione l'esecuzione passa al codice della funzione
- Quando la funzione termina, l'esecuzione riparte sequenzialmente dall'istruzione successiva all'invocazione della funzione

```
01) def mean(l:List[float])>float:
02)     s = sum(l)
03)     m = s / len(l)
04)     return m

05) def standard_deviation(l:List[float])>float:
06)     m = mean(l)
07)     s = .0
08)     for e in l:
09)         s += math.pow(m-e,2)
10)     return math.sqrt(s/len(l))

11) if __name__=="__main__":

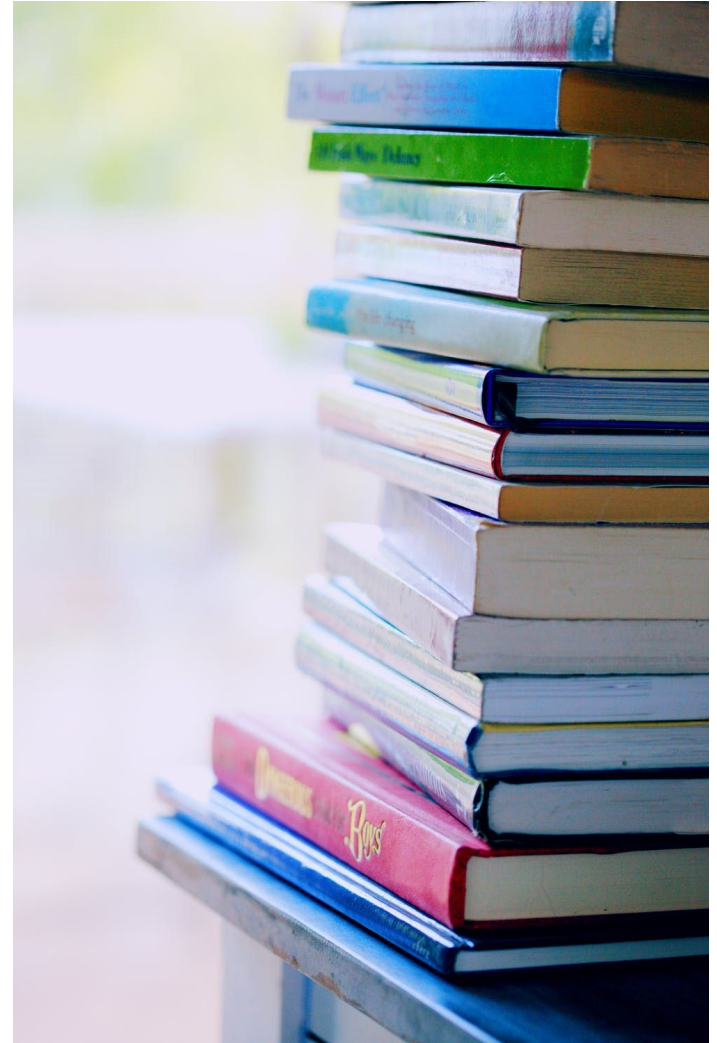
12)     a = [random.random() for x in range(10)]

13)     sd = standard_deviation(a)

14)     print(sd)
```

Call Stack

- Lo stack è una struttura che tiene traccia di tutte le chiamate a funzione
- lo stack contiene un frame per ogni chiamata a funzione
- il frame in cima allo stack è sempre quello della funzione corrente
- quando la funzione corrente invoca un'altra funzione la nuova funzione viene messa in cima allo stack
- quando la funzione corrente ritorna viene rimossa dallo stack e l'esecuzione riprende con la funzione precedente nello stack che diventa la nuova funzione corrente



Passaggio Parametri

- I parametri formali sono quelli che si trovano nell'intestazione (nel nostro esempio l)
- I parametri attuali sono quelli con cui viene invocata la funzione (nell'esempio l1)
- Quando una funzione viene invocata, i parametri formali diventano variabili locali in cui vengono copiati i valori dei parametri attuali (indirizzi in memoria)
- Quando la funzione ritorna, il valore di ritorno viene copiato nella variabile a cui è assegnato (nell'esempio l2)

```
def foo(l:List[int]) -> List[int]:
```

```
    print(id(l))
```

```
    l[0] = 0
```

```
    print(id(l))
```

```
    l = [9,9,9]
```

```
    print(id(l))
```

```
    return l
```

```
if __name__ == "__main__":
```

```
    l1 = [1,2,3]
```

```
    print(id(l1))
```

```
    l2 = foo(l1)
```

```
    print(id(l2))
```

1) Stampa l'indirizzo in memoria di l1 (supponiamo 123)

2) Stampa lo stesso indirizzo 123

3) Stampa ancora 123

4) Stampa un altro indirizzo (supponiamo 456)

5) Stampa ancora 456

Call Stack

```
def foo2(a:int, b:int, c:int)->int:  
    a = a + 2  
    b = b + 2  
    c = c + 2  
    return a + b + c
```

```
def foo1(a:int, b:int)->int:  
    a = a + 1  
    b = b * 2  
    c = foo2(a, b, a + b + 1)  
    return c
```

```
def run():  
    a = 2  
    b = 3  
    r = foo1(2,3)  
    print(r)
```



run()

__main__

Call Stack

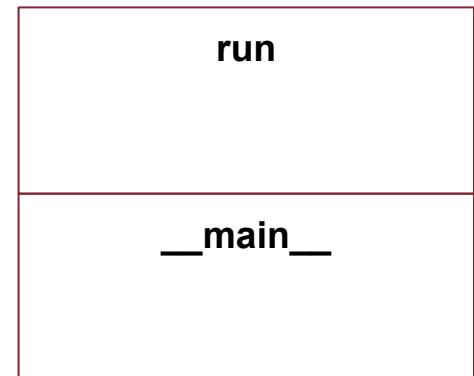
```
def foo2(a:int, b:int, c:int)->int:  
    a = a + 2  
    b = b + 2  
    c = c + 2  
    return a + b + c
```

```
def foo1(a:int, b:int)->int:  
    a = a + 1  
    b = b * 2  
    c = foo2(a, b, a + b + 1)  
    return c
```

```
def run():  
    a = 2  
    b = 3  
    r = foo1(2, 3)  
    print(r)
```



run()



Call Stack

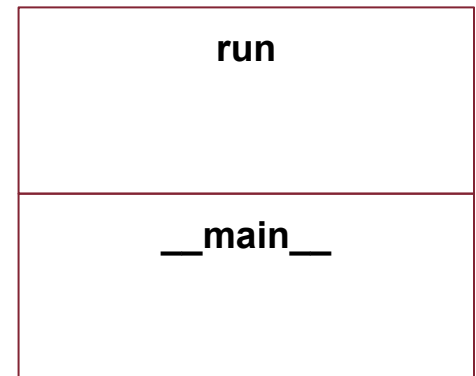
```
def foo2(a:int, b:int, c:int)->int:
    a = a + 2
    b = b + 2
    c = c + 2
    return a + b + c
```

```
def foo1(a:int, b:int)->int:
    a = a + 1
    b = b * 2
    c = foo2(a, b, a + b + 1)
    return c
```



```
def run():
    a = 2
    b = 3
    r = foo1(2, 3)
    print(r)
```

```
run()
```



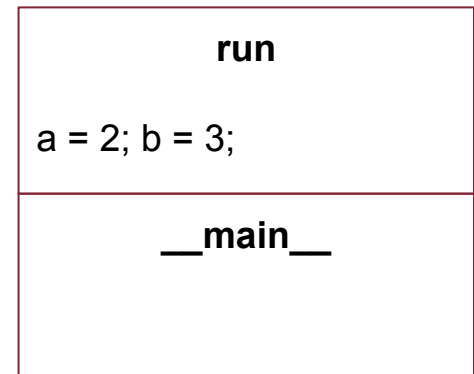
Call Stack

```
def foo2(a:int, b:int, c:int)->int:  
    a = a + 2  
    b = b + 2  
    c = c + 2  
    return a + b + c
```

```
def foo1(a:int, b:int)->int:  
    a = a + 1  
    b = b * 2  
    c = foo2(a, b, a + b + 1)  
    return c
```

```
def run():  
    a = 2  
    b = 3  
    r = foo1(2,3)  
    print(r)
```

```
run()
```



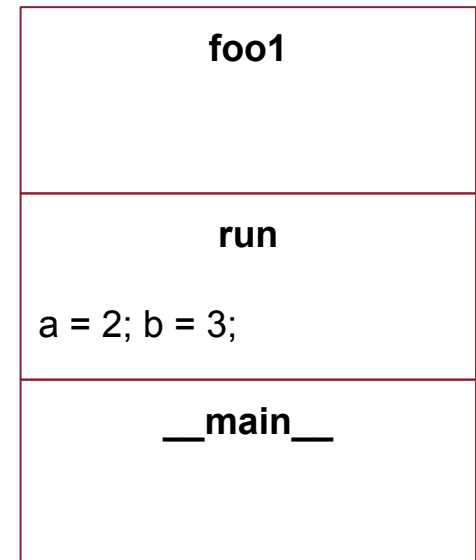
Call Stack

```
def foo2(a:int, b:int, c:int)->int:  
    a = a + 2  
    b = b + 2  
    c = c + 2  
    return a + b + c
```

```
def foo1(a:int, b:int)->int:  
    a = a + 1  
    b = b * 2  
    c = foo2(a, b, a + b + 1)  
    return c
```

```
def run():  
    a = 2  
    b = 3  
    r = foo1(2,3)  
    print(r)
```

```
run()
```



Call Stack

```
def foo2(a:int, b:int, c:int)->int:
    a = a + 2
    b = b + 2
    c = c + 2
    return a + b + c
```



```
def foo1(a:int, b:int)->int:
    a = a + 1
    b = b * 2
    c = foo2(a, b, a + b + 1)
    return c
```

```
def run():
    a = 2
    b = 3
    r = foo1(2, 3)
    print(r)
```

```
run()
```

foo1 a = 2; b = 3;
run a = 2; b = 3;
__main__

Call Stack

```
def foo2(a:int, b:int, c:int)->int:  
    a = a + 2  
    b = b + 2  
    c = c + 2  
    return a + b + c
```

```
def foo1(a:int, b:int)->int:  
    a = a + 1  
    b = b * 2  
    c = foo2(a, b, a + b + 1)  
    return c
```

```
def run():  
    a = 2  
    b = 3  
    r = foo1(2, 3)  
    print(r)
```


```
run()
```



foo1 a = 3; b = 6;
run a = 2; b = 3;
__main__

Call Stack

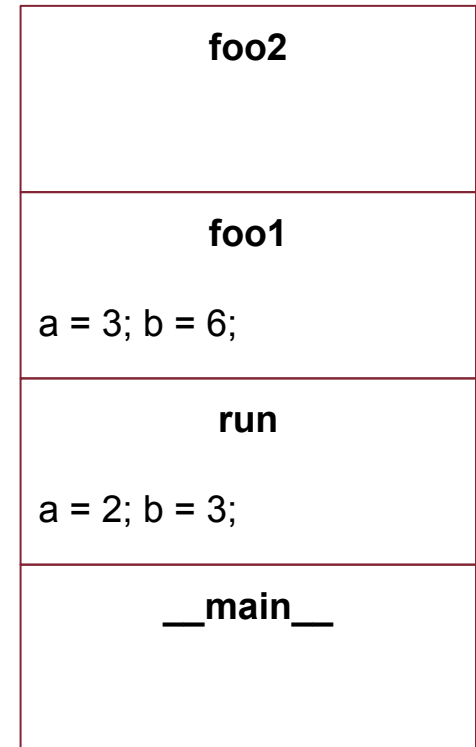
```
def foo2(a:int, b:int, c:int)->int:  
    a = a + 2  
    b = b + 2  
    c = c + 2  
    return a + b + c
```



```
def foo1(a:int, b:int)->int:  
    a = a + 1  
    b = b * 2  
    c = foo2(a, b, a + b + 1)  
    return c
```

```
def run():  
    a = 2  
    b = 3  
    r = foo1(2, 3)  
    print(r)
```

```
run()
```



Call Stack



```
def foo2(a:int, b:int, c:int)->int:
```

```
    a = a + 2
```

```
    b = b + 2
```

```
    c = c + 2
```

```
    return a + b + c
```

```
def foo1(a:int, b:int)->int:
```

```
    a = a + 1
```

```
    b = b * 2
```

```
    c = foo2(a, b, a + b + 1)
```

```
    return c
```

```
def run():
```

```
    a = 2
```

```
    b = 3
```

```
    r = foo1(2, 3)
```

```
    print(r)
```

```
run()
```

foo2 a = 3; b = 6; c = 10
foo1 a = 3; b = 6;
run a = 2; b = 3;
__main__

Call Stack



```
def foo2(a:int, b:int, c:int)->int:
```

```
    a = a + 2
```

```
    b = b + 2
```

```
    c = c + 2
```

```
    return a + b + c
```

```
def foo1(a:int, b:int)->int:
```

```
    a = a + 1
```

```
    b = b * 2
```

```
    c = foo2(a, b, a + b + 1)
```

```
    return c
```

```
def run():
```

```
    a = 2
```

```
    b = 3
```

```
    r = foo1(2, 3)
```

```
    print(r)
```

```
run()
```

foo2 a = 5; b = 8; c = 12
foo1 a = 3; b = 6;
run a = 2; b = 3;
__main__

Call Stack

```
def foo2(a:int, b:int, c:int)->int:  
    a = a + 2  
    b = b + 2  
    c = c + 2  
    return a + b + c
```

```
def foo1(a:int, b:int)->int:  
    a = a + 1  
    b = b * 2  
    c = foo2(a, b, a + b + 1)  
    return c
```

```
def run():  
    a = 2  
    b = 3  
    r = foo1(2,3)  
    print(r)
```

```
run()
```



foo1 a = 3; b = 6; c = 25;
run a = 2; b = 3;
__main__

Call Stack

```
def foo2(a:int, b:int, c:int)->int:
    a = a + 2
    b = b + 2
    c = c + 2
    return a + b + c
```

```
def foo1(a:int, b:int)->int:
    a = a + 1
    b = b * 2
    c = foo2(a, b, a + b + 1)
    return c
```



```
def run():
    a = 2
    b = 3
    r = foo1(2, 3)
    print(r)
```

```
run()
```

foo1 a = 3; b = 6; c = 25;
run a = 2; b = 3;
__main__

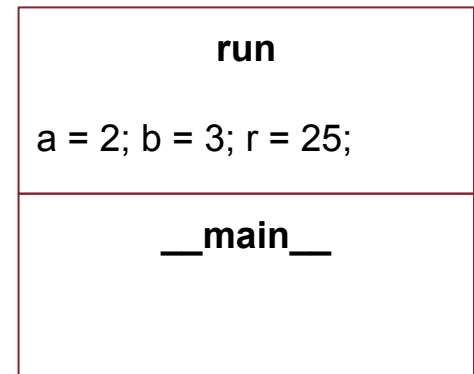
Call Stack

```
def foo2(a:int, b:int, c:int)->int:
    a = a + 2
    b = b + 2
    c = c + 2
    return a + b + c
```

```
def foo1(a:int, b:int)->int:
    a = a + 1
    b = b * 2
    c = foo2(a, b, a + b + 1)
    return c
```

```
def run():
    a = 2
    b = 3
    r = foo1(2,3)
    print(r)
```

```
run()
```



Call Stack

```
def foo2(a:int, b:int, c:int)->int:  
    a = a + 2  
    b = b + 2  
    c = c + 2  
    return a + b + c
```

```
def foo1(a:int, b:int)->int:  
    a = a + 1  
    b = b * 2  
    c = foo2(a, b, a + b + 1)  
    return c
```

```
def run():  
    a = 2  
    b = 3  
    r = foo1(2, 3)  
    print(r)
```



run()

print r = 25
run a = 2; b = 3; r = 25;
__main__

Call Stack

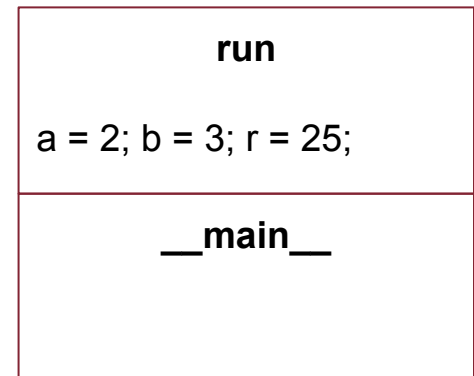
```
def foo2(a:int, b:int, c:int)->int:
    a = a + 2
    b = b + 2
    c = c + 2
    return a + b + c
```

```
def foo1(a:int, b:int)->int:
    a = a + 1
    b = b * 2
    c = foo2(a, b, a + b + 1)
    return c
```

```
def run():
    a = 2
    b = 3
    r = foo1(2, 3)
    print(r)
```



run()



Call Stack

```
def foo2(a:int, b:int, c:int)->int:  
    a = a + 2  
    b = b + 2  
    c = c + 2  
    return a + b + c
```

```
def foo1(a:int, b:int)->int:  
    a = a + 1  
    b = b * 2  
    c = foo2(a, b, a + b + 1)  
    return c
```

```
def run():  
    a = 2  
    b = 3  
    r = foo1(2,3)  
    print(r)
```

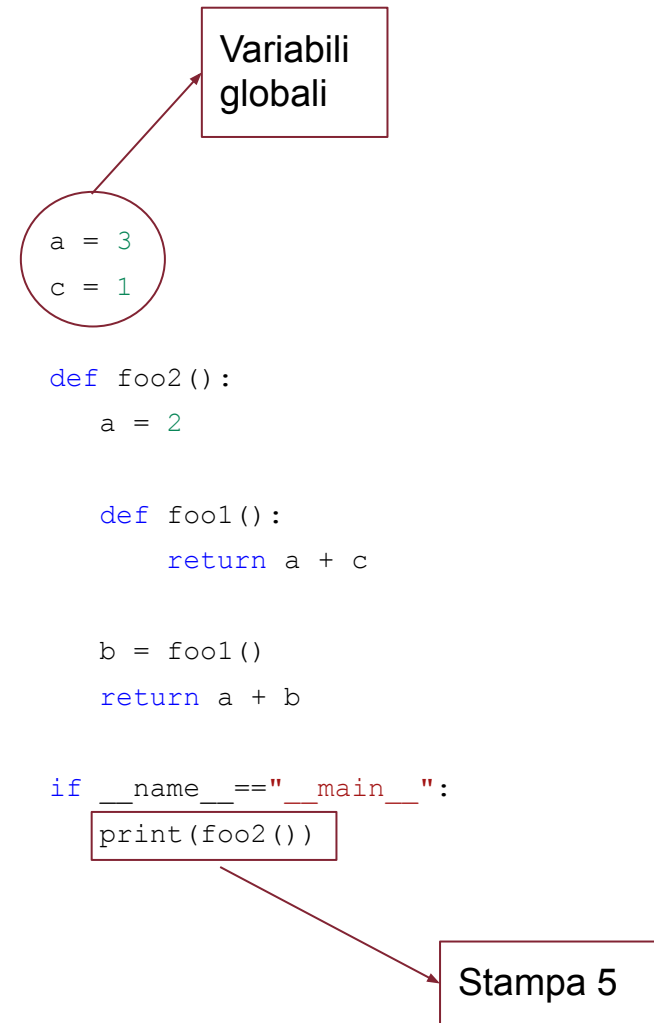


run()

__main__

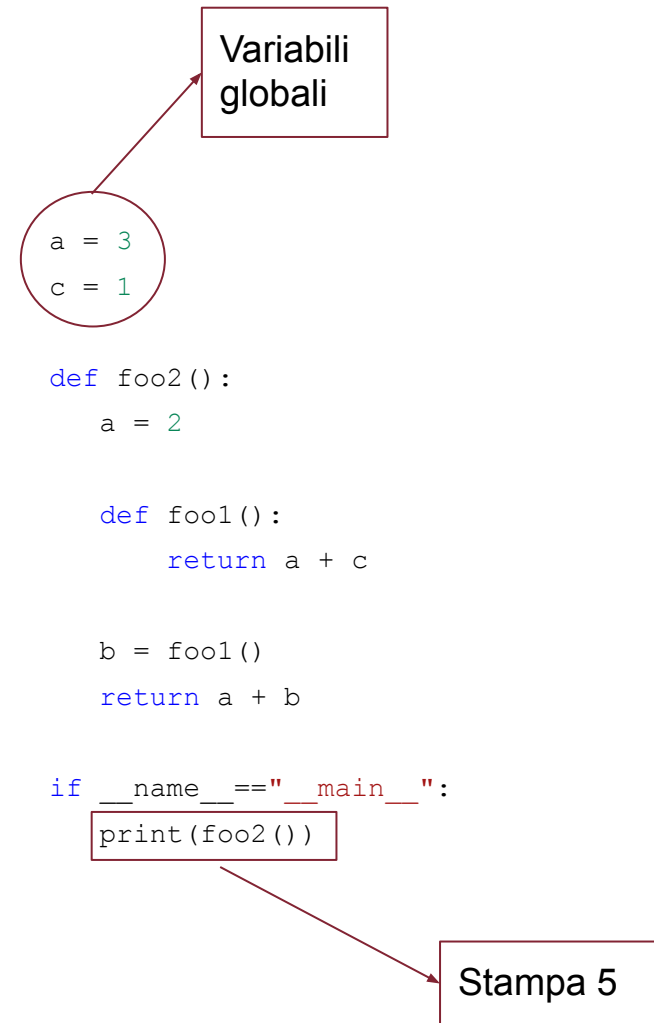
Passaggio Parametri

- Quando la funzione usa una variabile, essa viene cercata:
 - nella tabella delle variabili locali
 - se non viene trovata viene cercata nelle tabelle delle funzioni annidate più esterne
 - se ancora non viene trovata viene cercata nella tabella delle variabili globali



Scope

- La regione in cui una variabile è visibile si chiama scope
- Una variabile creata all'interno di una funzione è visibile solo all'interno della funzione o nelle funzioni annidate (dopo che è stata creata)
- Una variabile globale (definita al di fuori di qualsiasi funzione) è visibile anche all'interno delle funzioni nello stesso file



Esempio 1

```
def foo(a,b):  
    a = 2 * a  
    b = 2 * b  
    return a + b  
  
if __name__=="__main__":  
    a = 2  
    b = 5  
    c = foo(a,b)
```

```
print("c:{}\na:{}\nb:{}".format(c,a,b))
```

Cosa stampa?

Esempio 1

```
def foo(a,b):  
    a = 2 * a  
    b = 2 * b  
    return a + b  
  
if __name__=="__main__":  
    a = 2  
    b = 5  
    c = foo(a,b)
```

```
print("c:{}\na:{}\nb:{}".format(c,a,b))
```

Stampa “c:14 a:2 b:5”. Il valore delle variabili a e b non viene modificato dalla funzione.

Esempio 2

```
def foo1(l:List[int]):  
    for i in range(len(l)):  
        l[i] = 1  
  
def foo2(l:List[int]):  
    l = [2,2,2]  
  
def foo3(l:List[int]):  
    l = l.copy()  
    for i in range(len(l)):  
        l[i] = 3  
    return l  
  
if __name__=="__main__":  
    l = [1,2,3]  
    foo1(l)  
    print(l)  
    foo2(l)  
    print(l)  
    l3 = foo3(l)  
    print(l)  
    print(l3)
```

Cosa stampa?

Esempio 2

```
def foo1(l:List[int]):  
    for i in range(len(l)):  
        l[i] = 1  
  
def foo2(l:List[int]):  
    l = [2,2,2]  
  
def foo3(l:List[int]):  
    l = l.copy()  
    for i in range(len(l)):  
        l[i] = 3  
    return l  
  
if __name__=="__main__":  
    l = [1,2,3]  
    foo1(l)  
    print(l)  
    foo2(l)  
    print(l)  
    l3 = foo3(l)  
    print(l)  
    print(l3)
```

Una lista in Python è un oggetto mutabile, ovvero il contenuto può cambiare pur rimanendo lo stesso oggetto

foo2 aggiorna la variabile locale **l**, che ora contiene il riferimento di un nuovo oggetto lista

foo3 fa una copia della lista referenziata da **l** e la assegna alla variabile locale **l**. Dopo la copia **l** contiene il riferimento ad un altro oggetto.

Stampa

[1, 1, 1]
[1, 1, 1]
[1, 1, 1]
[3, 3, 3]

Moduli

- Per aumentare la modularità, un programma può essere suddiviso in moduli
- I moduli racchiudono delle unità logiche del programma
- Anche la [libreria standard di Python](#) è suddivisa in moduli
- Per utilizzare le funzionalità racchiuse in un modulo bisogna usare la parola chiave **import**

```
import random
```

```
a = random.randint(1,6)  
print(a)
```

Importa il modulo random. Da adesso tutte le funzionalità del modulo possono essere utilizzate specificando il nome del modulo

```
from random import randint
```

```
a = randint(1,6)  
print(a)
```

Importa una funzione specifica dal modulo random. La funzione può essere invocata senza specificare il nome del modulo

Moduli

- Per creare un modulo basta creare un file **nome_modulo.py**
- Una volta importato, tutte le funzioni, classi e variabili definite nel modulo possono essere utilizzate (precedute dal nome del modulo)
- se all'interno del modulo è presente codice eseguibile esso viene eseguito solo una volta al momento dell'import

mio_modulo.py

```
def sum(a:float,b:float)->float:
    return a+b

def pow(base:float,exp:int)->float:
    res = 1.
    for i in range(exp):
        res *= base
    return res

print("2^3 + 3^2 = {}".format(sum(pow(2,3),pow(3,2))))
```

main.py

```
import mio_modulo

print("2^4 + 2^2 = {}".format(
    mio_modulo.sum(
        mio_modulo.pow(2,4),
        mio_modulo.pow(3,2))))
```

Se lancio il programma con
\$ python main.py
il programma stampa:
 $2^3+3^2 = 17.0$
 $2^4+2^2 = 25.0$

Nome Modulo

- Ogni modulo ha una variabile globale chiamata **`__name__`** che contiene il nome del modulo
- Quando il modulo è eseguito come script (**`$ python nome_modulo.py`**), la variabile **`__name__`** viene settata a **`__main__`**

main.py

```
import mio_modulo2

print("il nome del modulo  
importato è:")
mio_modulo2.print_module_name()
```

mio_modulo2.py

```
def print_module_name()->None:
    print(__name__)

if __name__=="__main__":
    print("testing")
```

```
$ python main.py
```

```
$ python mio_modulo2.py
```

?

Nome Modulo

- Ogni modulo ha una variabile globale chiamata **`__name__`** che contiene il nome del modulo
- Quando il modulo è eseguito come script (**`$ python nome_modulo.py`**), la variabile **`__name__`** viene settata a **`"__main__"`**

main.py

```
import mio_modulo2

print("il nome del modulo
importato è:")
mio_modulo2.print_module_name()
```

mio_modulo2.py

```
def print_module_name()->None:
    print(__name__)

if __name__=="__main__":
    print("testing")
```

`$ python main.py`

"il nome del modulo importato è
mio_modulo2"

`$ python mio_modulo2.py`

"testing"

Packaging

- Regole che ci permettono di strutturare un'applicazione con più moduli
- Ecco un esempio di struttura di package

```
sound/                                Top-level package
  __init__.py                          Initialize the sound package
  formats/                             Subpackage for file format conversions
    __init__.py
    wavread.py
    wavwrite.py
    aiffread.py
    aiffwrite.py
    auread.py
    auwrite.py
    ...
  effects/                             Subpackage for sound effects
    __init__.py
    echo.py
    surround.py
    reverse.py
    ...
  filters/                             Subpackage for filters
    __init__.py
    equalizer.py
    vocoder.py
    karaoke.py
    ...
```

Import

- Dato l'esempio precedente, posso importare individualmente moduli dal package:

```
import sound.effects.echo
```

```
sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

oppure

```
from sound.effects import echo
```

```
echo.echofilter(input, output, delay=0.7, atten=4)
```

oppure posso importare direttamente la funzione o variabile che devo utilizzare

```
from sound.effects.echo import echofilter
```

```
echofilter(input, output, delay=0.7, atten=4)
```

Relative / Absolute Import

- All'interno di un modulo di un package posso fare riferimento ad altri moduli utilizzando il percorso assoluto, partendo dalla root directory. Ad esempio per importare echo all'interno del modulo vocoder posso usare l'istruzione

```
from sound.effects import echo
```

- Oppure posso fare riferimento ad altri moduli utilizzando il path relativo al modulo corrente

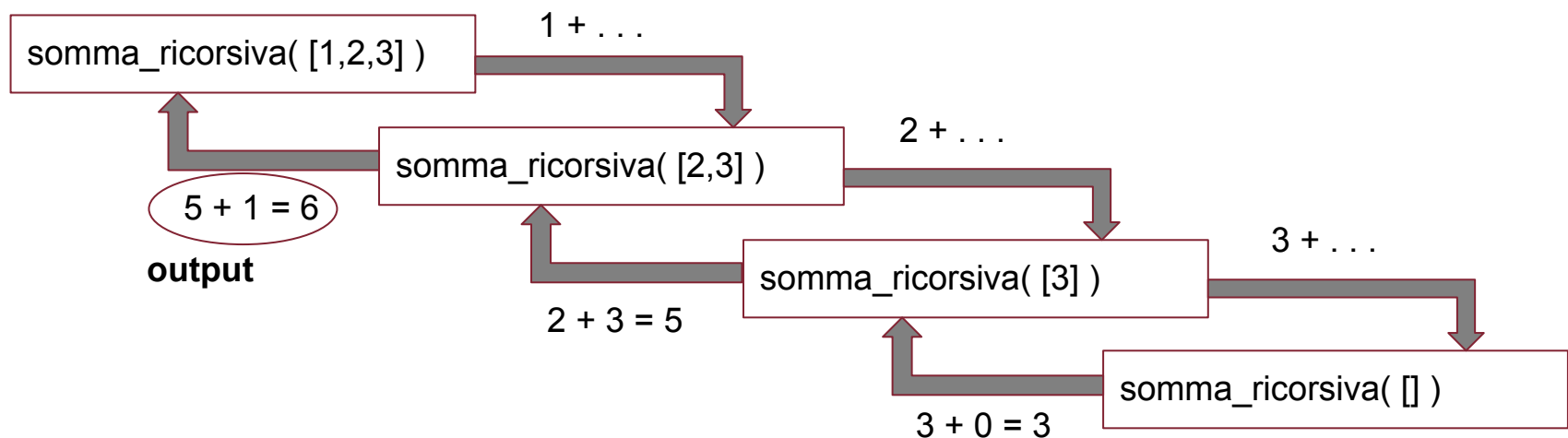
```
from . import echo  
from .. import formats  
from ..filters import equalizer
```

- Attenzione, se utilizzo un modulo come “main module” devo usare solo import assoluti in quanto l'import relativo è basato sul nome del modulo, che nel caso venga utilizzato come “main module” è sempre “__main__”

Ricorsione

- Potente tecnica di programmazione basata sul principio di induzione
- Pensare al problema da risolvere come n sotto-problemi
- Specificare la soluzione per il passo base
- Specificare come risolvere il problema al passo n utilizzando la soluzione del problema al passo n-1

```
def somma_ricorsiva(l:List[int])>int:  
    if len(l)==0:  
        return 0  
    else:  
        return l[0] + somma_ricorsiva(l[1:])
```



Exercises

- Refactor the exercises of the previous lecture using functions
- Write a recursive function to compute the n th power of a number
- Write a function that implements binary search in a list using recursion

Slides distribuite con Licenza Creative Commons (CC BY-NC-ND 4.0) Attribuzione - Non commerciale - Non opere derivate 4.0 Internazionale

PUOI CONDIVIDERLE ALLE SEGUENTI CONDIZIONI

(riprodurre, distribuire, comunicare o esporre in pubblico, rappresentare, eseguire e recitare questo materiale con qualsiasi mezzo e formato)

Attribuzione*

Devi riconoscere una menzione di paternità adeguata, fornire un link alla licenza e indicare se sono state effettuate delle modifiche. Puoi fare ciò in qualsiasi maniera ragionevole possibile, ma non con modalità tali da suggerire che il licenziante avalli te o il tuo utilizzo del materiale.

Non Commerciale

Non puoi utilizzare il materiale per scopi commerciali.

Non opere derivate

Se remixi, trasformi il materiale o ti basi su di esso, non puoi distribuire il materiale così modificato.

Divieto di restrizioni aggiuntive

Non puoi applicare termini legali o misure tecnologiche che impongano ad altri soggetti dei vincoli giuridici a questa licenza