



Politecnico  
di Torino

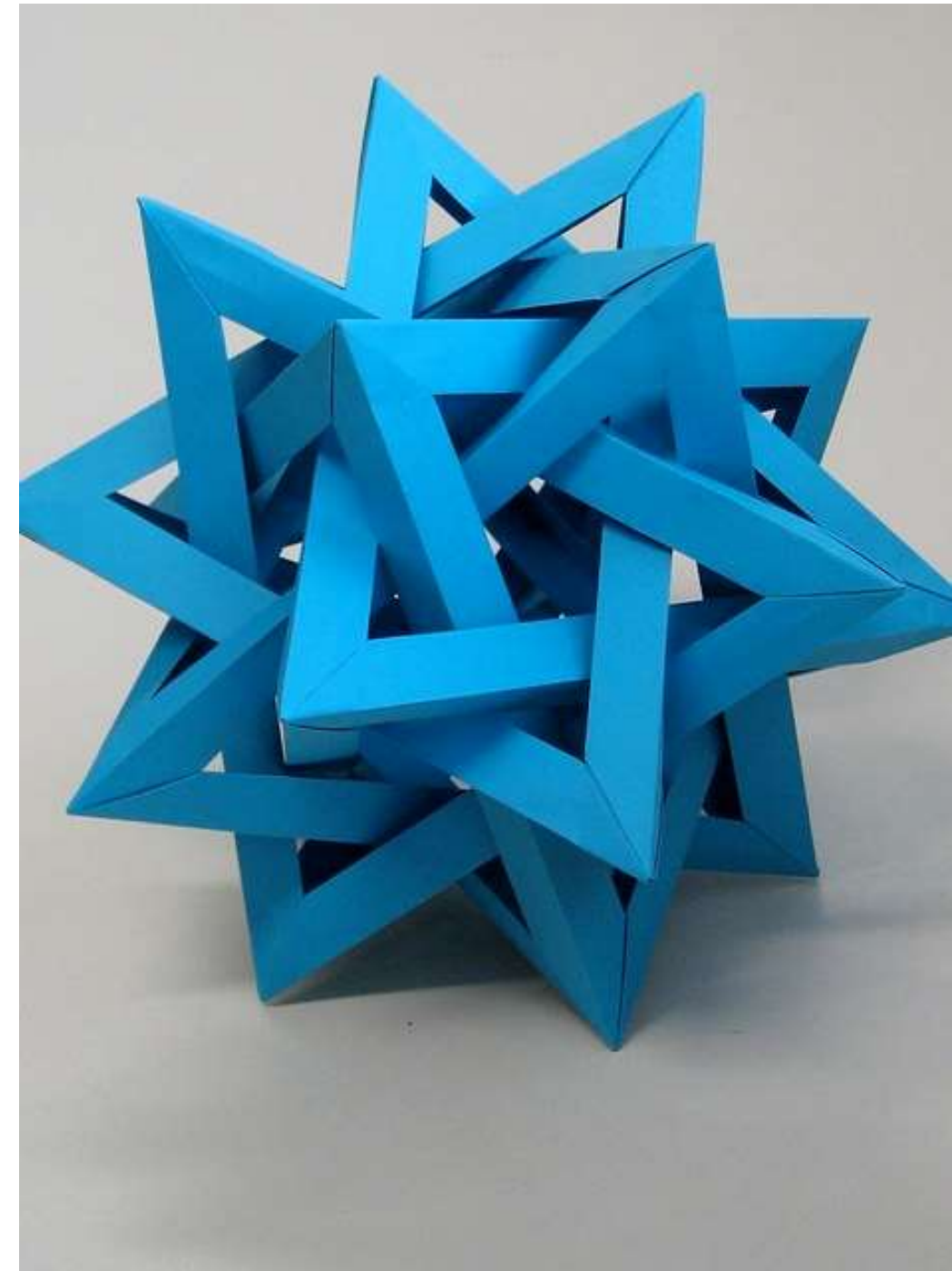
Dipartimento  
di Automatica e Informatica

# Unità P5: Funzioni

STRUTTURARE IL CODICE, FUNZIONI,  
PARAMETRI, RESTITUZIONE DI VALORI



Capitolo 5



[This Photo](#) by Unknown Author is licensed under [CC BY-SA](#)

# Obiettivi dell'Unità

- Realizzare **funzioni**
- Acquisire familiarità con il concetto di **passaggio di parametri**
- Sviluppare strategie per **scomporre** problemi complessi in problemi più semplici
- Saper determinare l'ambito di **visibilità** di una variabile

*In questa Unità, imparerete a progettare e realizzare **le vostre funzioni**.*

*Usando un processo di raffinamenti successivi, sarete in grado di **scomporre problemi complessi in un insieme di funzioni cooperanti***

# Funzioni come scatole nere

---



5.1

# Funzioni come scatole nere

- Una **funzione** è una sequenza di **istruzioni** a cui viene dato un **nome**
- Per esempio, la funzione **round** contiene le istruzioni per arrotondare un numero a virgola mobile a un valore con un determinato numero di cifre decimali

```
round(number[, ndigits])
```

Return *number* rounded to *ndigits* precision after the decimal point. If *ndigits* is omitted or is `None`, it returns the nearest integer to its input.

For the built-in types supporting `round()`, values are rounded to the closest multiple of 10 to the power minus *ndigits*; if two multiples are equally close, rounding is done toward the even choice (so, for example, both `round(0.5)` and `round(-0.5)` are 0, and `round(1.5)` is 2). Any integer value is valid for *ndigits* (positive, zero, or negative). The return value is an integer if *ndigits* is omitted or `None`. Otherwise the return value has the same type as *number*.

<https://docs.python.org/3/library/functions.html#round>

# Invocare funzioni

- Si **invoca** (o **chiama**) una funzione per eseguire le sue istruzioni  
`price = round(6.8275, 2) # Assegna 6.83 a price`
- Usando l'espressione `round(6.8275, 2)`, il programma **invoca la funzione round**, chiedendole di arrotondare 6.8275 a due cifre decimali

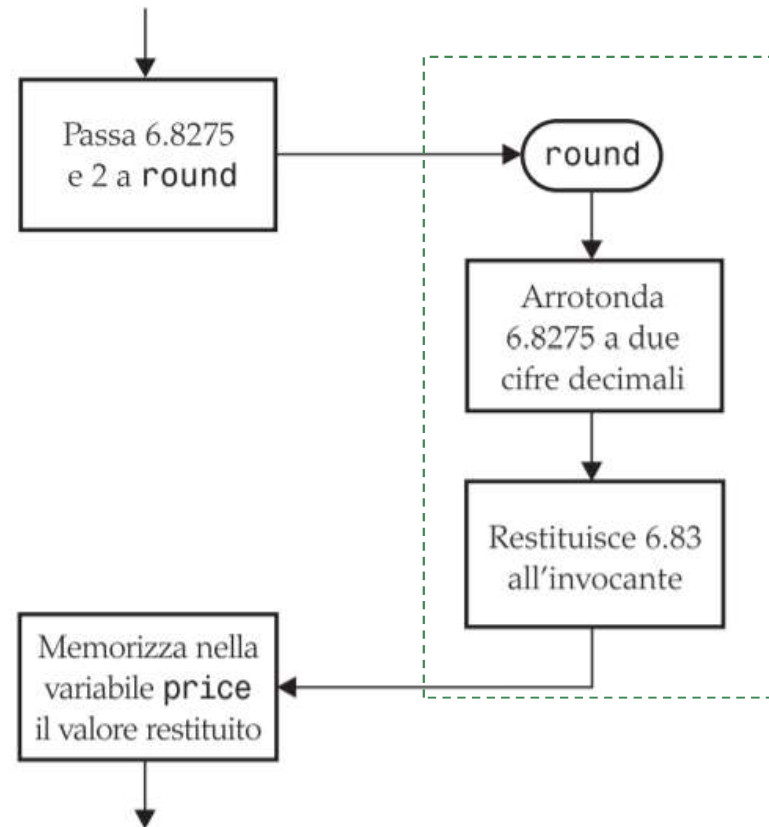
# Invocare funzioni (2)

- Si **invoca** (o **chiama**) una funzione per eseguire le sue istruzioni  
`price = round(6.8275, 2) # Assegna 6.83 a price`
- Usando l'espressione `round(6.8275, 2)`, il programma **invoca la funzione round**, chiedendole di arrotondare 6.8275 a due cifre decimali
- Quando una funzione termina, il **risultato** ottenuto è **restituito** dalla funzione e può essere **usato** in un'espressione (e.g., assegnato a `price`)
- Dopo che il valore è stato utilizzato, il programma riprende l'esecuzione

# Invocare funzioni (3)

```
price = round(6.8275, 2) # Assegna 6.83 a price
```

**Figura 1**  
Flusso di esecuzione  
dell'invocazione  
di una funzione



Non fa parte del mio codice. La funzione è definita nella libreria standard.

# Gli argomenti delle funzioni

- Quando un'altra funzione invoca la funzione round, **le fornisce gli "input"**, ossia i valori 6.8275 e 2 nell'invocazione round(6.8275, 2)
- Questi valori sono detti **argomenti** dell'invocazione di una funzione
  - Notare che non sono necessariamente input forniti dall'utente umano
  - Sono i valori per i quali si vuole che la funzione calcoli il proprio risultato
- Le funzioni possono ricevere diversi argomenti
- È anche possibile avere funzioni senza argomenti



# Le funzioni restituiscono valori

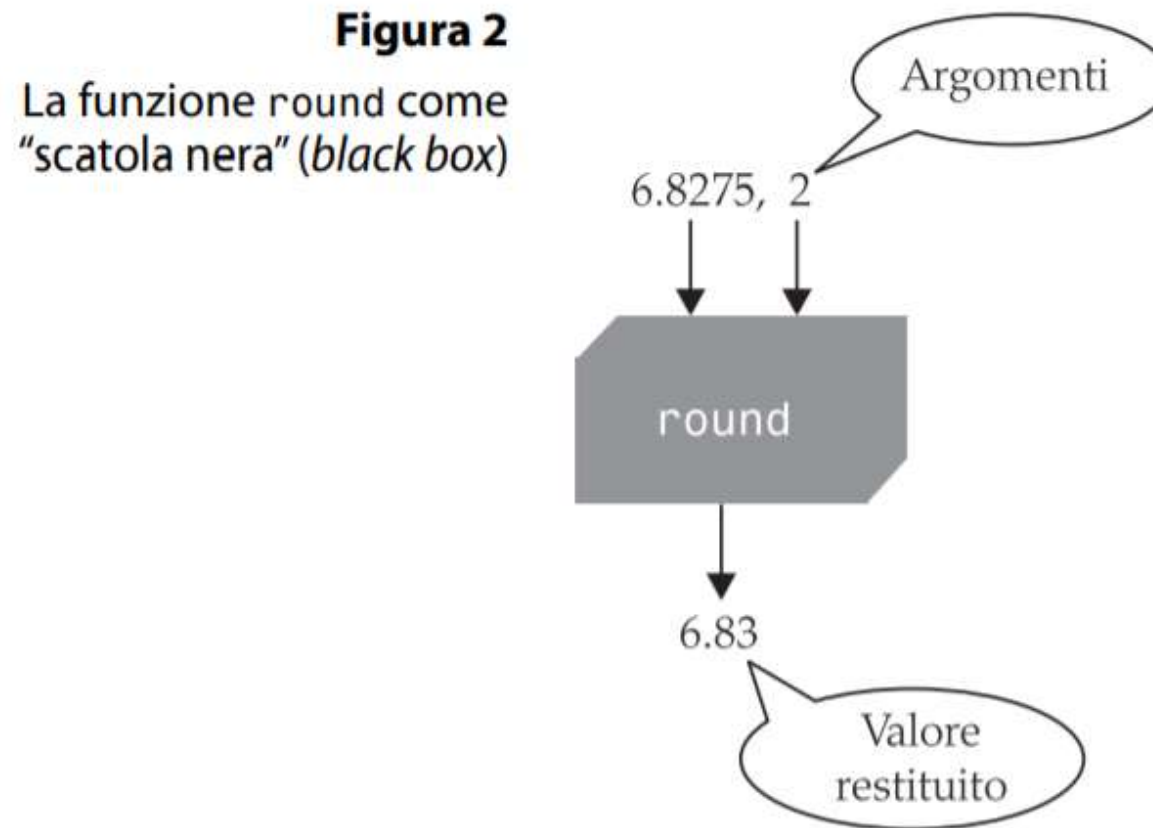
- L' “**output**” che la funzione `round` calcola è detto **valore restituito**
- La funzione restituisce **un solo valore**
  - Se volessi restituire più valori, potrei aggirare la limitazione restituendo una lista o una tupla (impareremo più avanti...)
  - Alcune funzioni non hanno bisogno di restituire alcun valore
- Il **valore restituito** dalla funzione è passato al punto del programma dove la funzione è stata invocata  
`price = round(6.8275, 2)`
  - Quando la funzione restituisce il suo risultato, il valore è memorizzato nella variabile ‘`price`’

# Analogia con una scatola nera

- Un termostato è una ‘scatola nera’
  - Si imposta la temperatura desiderata
  - Si seleziona il riscaldamento o il raffrescamento
  - Non è necessario sapere come funziona davvero!
    - Come conosce la temperatura corrente?
    - Che segnale/comando invia al termosifone o al condizionatore?
- Le funzioni sono come ‘scatole nere’
  - Si passa alla funzione ciò che le serve per eseguire il suo compito
  - Si ottiene il risultato

# La funzione `round` come una scatola nera

- Si passano alla funzione `round` gli argomenti necessari (6.8275 & 2) ed essa calcola il risultato (6.83)



# La funzione `round` come una scatola nera

- Vi potrete chiedere... come fa la funzione `round` a svolgere il suo compito?
- Come utilizzatori della funzione, *non vi serve sapere come* la funzione sia stata implementata
- Vi serve solo conoscere le **specifiche** della funzione:
  - Se si forniscono gli argomenti `x` e `n`, la funzione restituisce `x` arrotondato a `n` cifre decimali
- **Quando si progetta una funzione, la si progetta come se fosse una scatola nera**
  - Anche se siete le uniche persone a lavorare su un programma, nel futuro le utilizzerete come semplici scatole nere e farete in modo che altri programmatori facciano lo stesso
  - Vale sia per le funzioni di libreria, che per le funzioni che progetterete voi

# Trovare le funzioni nelle librerie

- Funzioni **Built-In** (**predefinite**) nella libreria standard
  - <https://docs.python.org/3/library/functions.html>

Built-in Functions				
abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	dir()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	

Importante  
(già usata)

Utile  
(dare un'occhiata)

Verrà usata poi  
(con liste, dizionari...)

# Trovare le funzioni nelle librerie (2)

- Dentro i **moduli** nella libreria standard
  - <https://docs.python.org/3/library/>
  - <https://docs.python.org/3/py-modindex.html>
  - Più di 200 moduli, con varie funzioni in ciascuno
  - Moduli interessanti: string, math, random, statistics, os.path, csv, json, ...
- Ricordare le istruzioni
  - **import module**
  - **from module import function**

Vedi Unità P2

Importare i moduli

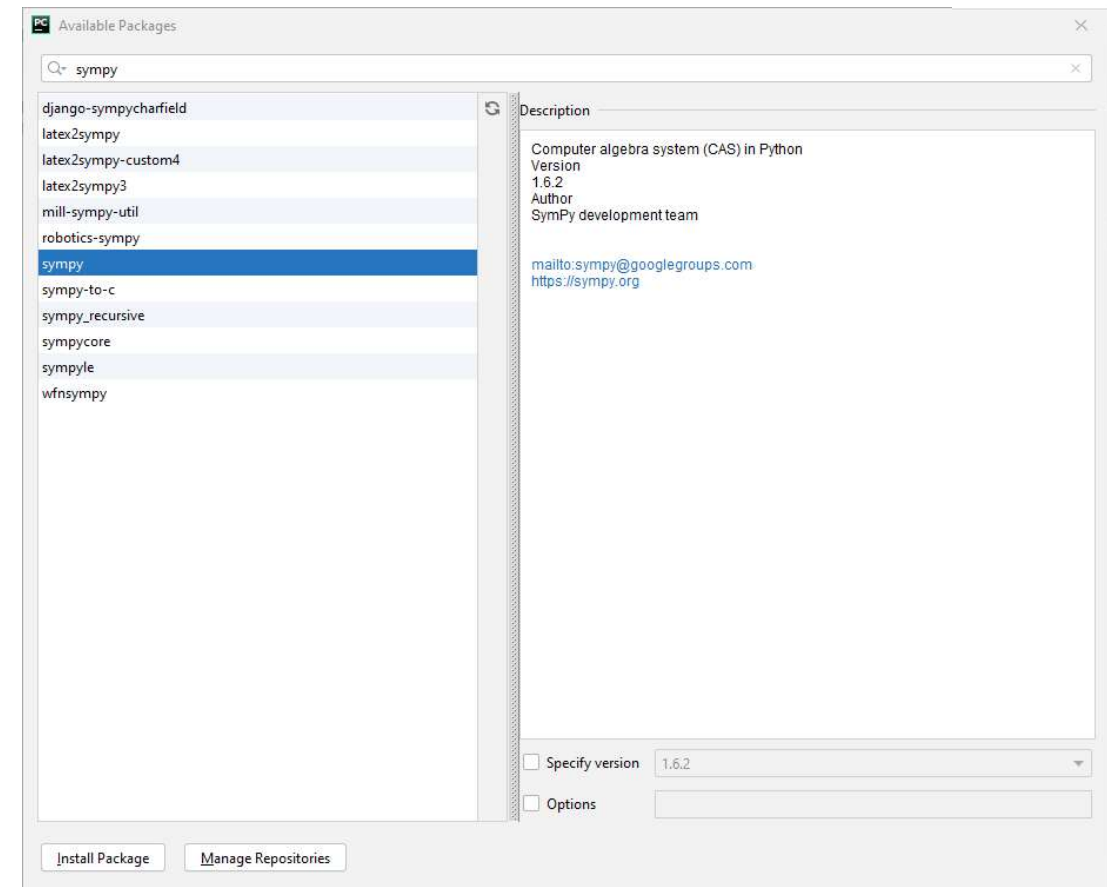
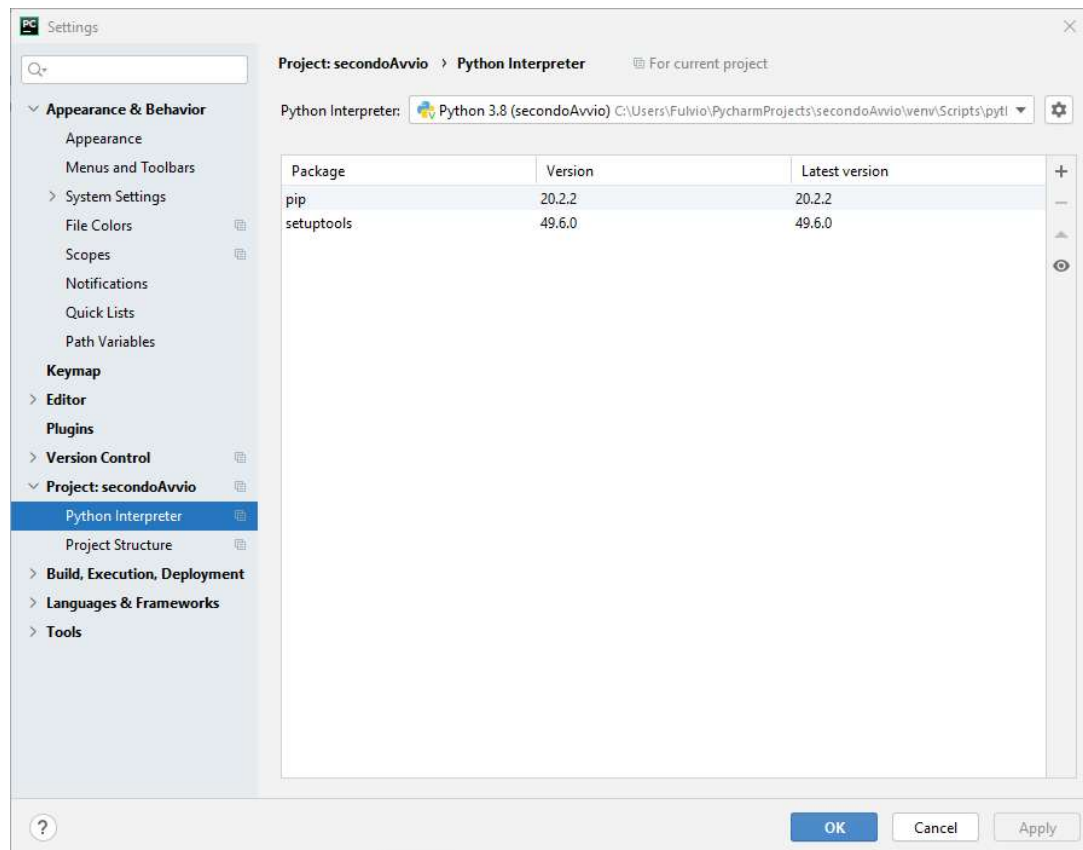
- Tre modi per importare le funzioni dai moduli:
  - `from math import sqrt, sin, cos`  
# imports listed functions
  - `from math import *`  
# imports all functions from the module
  - `import math`  
# imports the module and gives access to all functions
- Se si usa il terzo metodo, bisogna aggiungere il nome del modulo e un "." prima di ogni invocazione di funzione
  - `import math`
  - `y = math.sqrt(x)`

# Trovare le funzioni nelle librerie (3)

- Disponibile per il **download** dalla repository Python Package Index (PyPI)
  - <https://pypi.org/> - oltre 200k moduli disponibili
  - Installare un nuovo modulo nel proprio progetto:
    - con `'pip install module'`
    - da PyCharm Project Settings
    - ...
- Ricordare:
  - `import module`
  - `from module import function`

# Installare moduli su PyCharm

Argomento  
avanzato





# Realizzazione e collaudo di funzioni



5.2

# Realizzare e collaudare le funzioni

- Esempio: Funzione che calcola il **volume di un cubo**
- Cosa le serve per il suo compito?
- Con che cosa risponderà?

# Definire la funzione

- Per scrivere ('definire') questa funzione
  - Scegliere un **nome** per la funzione (**cubeVolume**)
  - Dichiarare una **variabile** per ogni **argomento**
    - (**sideLength**) – lista di variabili parametro
  - Mettere assieme tutte queste informazioni con la parola riservata **def** per formare la prima riga della definizione della funzione:

```
def cubeVolume(sideLength):
```

Nome della  
funzione

Variabile  
parametro

Questa riga è detta **intestazione**  
(**header**) della funzione

# Implementare la funzione

- L'istruzione **def** apre un nuovo blocco (compound statement), all'interno del quale scriveremo le istruzioni che compongono il **corpo** della funzione

```
def cubeVolume(sideLength) :  
    volume = sideLength ** 3  
    return volume
```

Corpo della  
funzione

- All'interno del corpo si possono utilizzare le variabili parametro, come se fossero normali variabili

# Sintassi: definizione di funzione

## *Sintassi*

```
def nomeDiFunzione(nomeParametro1, nomeParametro2, . . .) :  
    enunciati
```

## *Esempio*

Intestazione della funzione.

Nome della funzione.

Nome della variabile parametro.

Corpo della funzione,  
eseguito quando la funzione  
viene invocata.

```
[ def cubeVolume(sideLength) :  
  [ volume = sideLength ** 3  
    return volume
```

Enunciato `return`: termina la funzione  
e restituisce il risultato.

# Collaudare una funzione

- Se si esegue un programma contenente solamente la definizione della funzione, non accade nulla
  - Dopotutto, nessuno sta **invocando** la funzione
- Per collaudare la funzione, il programma deve contenere:
  - La **definizione** della funzione (*corpo della funzione*)
    - In cui si calcola il valore restituito (*return value*), con l'enunciato `return`
  - Le istruzioni che **chiamano** la funzione e **visualizzano** il risultato

# Invocare/Collaudare la funzione `cubeVolume`

- Implementare la funzione (definizione della funzione):

```
def cubeVolume(sideLength) :  
    volume = sideLength ** 3  
    return volume
```

- Invocare/collaudare la funzione:

```
result1 = cubeVolume(2)  
result2 = cubeVolume(10)  
print("Un cubo con lato di lunghezza 2 ha volume ", result1)  
print("Un cubo con lato di lunghezza 10 ha volume ", result2)
```

# Invocazione di una funzione

Codice chiamante

```
result1 = cubeVolume(2)
```

1) Il valore dell'argomento viene usato per inizializzare la variabile parametro  
(come se scrivessi `sideLength = 2`)

Funzione chiamata

```
def cubeVolume(sideLength) :  
    volume = sideLength ** 3  
    return volume
```

2) Il corpo della funzione viene eseguito, con il valore corrente della variabile parametro

3) Il risultato della funzione viene restituito ed usato nel punto di chiamata



# Passaggio di parametri

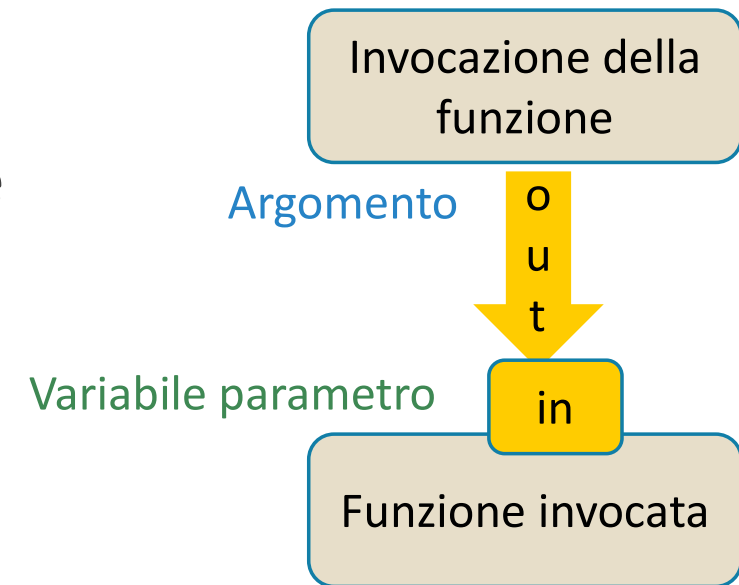
---



5.3

# Passaggio di parametri

- Le **variabili parametro** (parametri formali) ricevono gli **argomenti** (parametri effettivi o parametri attuali) dell'invocazione della funzione
- Gli **argomenti** possono essere:
  - Il valore corrente di una variabile o espressione
  - Un valore 'letterale' : 2, 3.14, 'hello'
  - Sono detti '*parametri attuali*' o **argomento**
- La **variabile parametro** è:
  - Dichiarata nella funzione invocata
  - *Inizializzata* con il valore dell' **argomento**
  - *Usata come una variabile* all'interno della funzione invocata
  - Sono detti '*parametri formali*'



# Fasi del passaggio di parametri

```
result1 = cubeVolume(2)
```

result1 = 8

```
def cubeVolume(sideLength):  
    volume = sideLength ** 3  
    return volume
```

sideLength = 2

volume = 8

# Argomenti nella chiamata

Codice chiamante

```
result1 = cubeVolume(2)
```

Argomento: costante  
sideLength = 2

Codice chiamante

```
result1 = cubeVolume(a**2+b**2)
```

Argomento: espressione  
sideLength = a\*\*2+b\*\*2

Codice chiamante

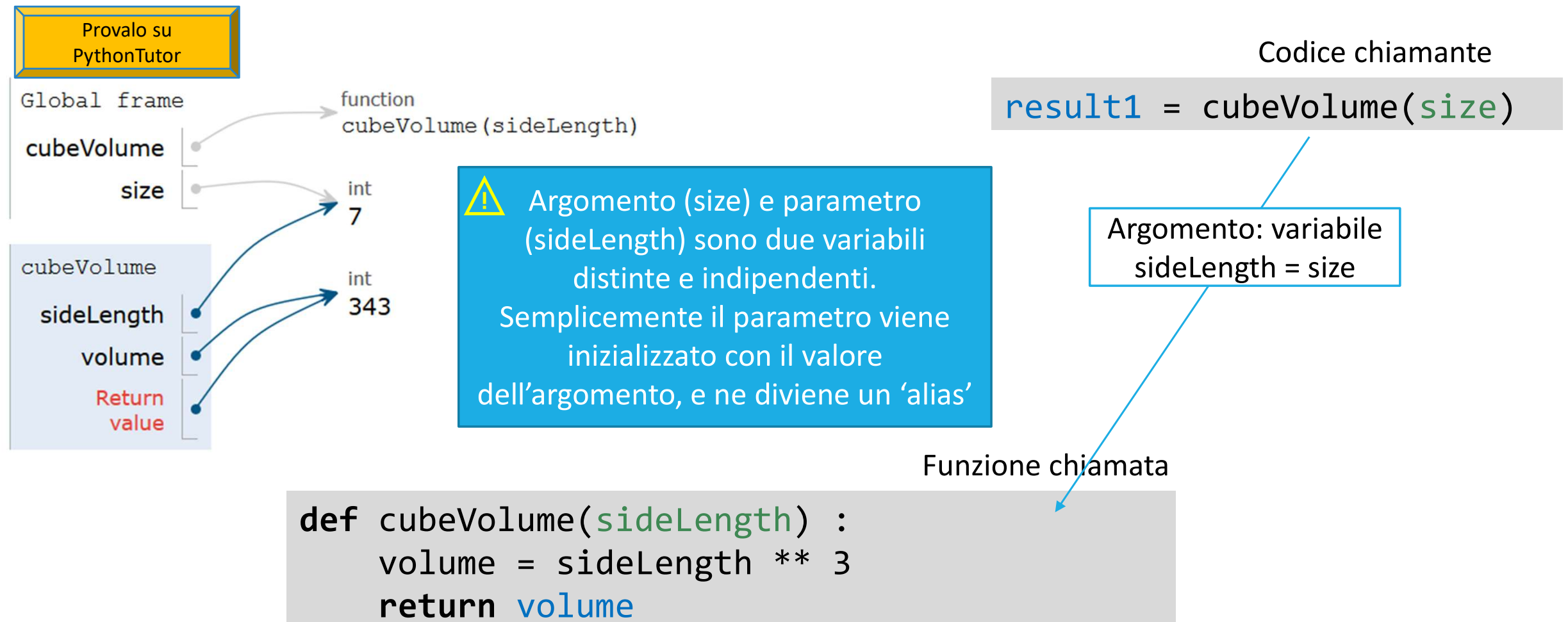
```
result1 = cubeVolume(size)
```

Argomento: variabile  
sideLength = size

Funzione chiamata

```
def cubeVolume(sideLength) :  
    volume = sideLength ** 3  
    return volume
```

# Argomenti di tipo 'variabile'



# Errori comuni

- Tentare di **modificare gli argomenti (parametri attuali)**
- Viene passata solo una copia degli **argomenti** (viene passato il **valore**)
  - La funzione invocata (addTax) può modificare solo la copia locale (**price**)
  - La variabile **total** non è **modificata** nella funzione
  - **total == 10** dopo l'invocazione della funzione

```
total = 10
newPrice = addTax(total, 7.5)
```

Copy  
value

total  
10.0

```
def addTax(price, rate):
    tax = price * rate / 100
    # No effect outside the function
    price = price + tax
    return price
```

price  
10.75

# Suggerimenti

- Non modificare le variabili parametro
  - Molti programmatori trovano che ciò possa generare confusione

```
def totalCents(dollars, cents) :  
    cents = dollars * 100 + cents # Modifica la variabile parametro.  
    return cents
```

- Per evitare la confusione, va semplicemente introdotta una variabile diversa:

```
def totalCents(dollars, cents) :  
    result = dollars * 100 + cents  
    return result
```

# Suggerimento: commentare le funzioni

- Quando si scrive una funzione, bisognerebbe **commentare** il suo comportamento
- Ricordare: i commenti sono per i lettori **umani**, non per il compilatore

```
## Calcola il volume di un cubo.  
# @param sideLength la lunghezza di un lato del cubo  
# @return il volume del cubo  
#  
def cubeVolume(sideLength) :  
    volume = sideLength ** 3  
    return volume
```

*I commenti nella funzione ne spiegano lo **scopo**,  
il significato dei **parametri** e del valore **restituito**,  
così come requisiti specifici*

*Ci sono degli standard per convertire  
automaticamente in documentazione i  
commenti “strutturati”.  
Vedere: <https://realpython.com/documenting-python-code/>*



# Parametri posizionali o nominali

- In Python le variabili parametro di funzione sono indicate dal loro nome specificato nella definizione della funzione.
- Nella chiamata a funzione, gli argomenti possono venire associati alle variabili parametro in due modi
  - **Positional parameters**: associa la posizione del dato all'argomento
    - Il primo argomento passato andrà ad inizializzare la prima variabile parametro, il secondo la seconda, e così via (comportamento di default)
  - **Named parameters**: uso il nome della variabile parametro per specificare a chi assegnare l'argomento

Es: funzione `def complex(real, imag):`

`x = complex(3, 5) → 3 + 5j → (positional parameters)`

`x = complex(real = 3, imag = 5) → 3 + 5j (named parameters)`

# Positional and named parameters: regole

- I **positional** parameters **devono precedere** i named parameters.
- I **named** parameters possono essere inseriti in un qualsiasi ordine

Es: funzione `def complex(real, imag)`

`x = complex(3, 5) → 3 + 5j → (positional parameters)`

`x = complex(real = 3, imag = 5) → 3 + 5j (named parameters)`

`x = complex(imag = 5, real = 3) → 3 + 5j (named parameters)`

# Valori di default

- Le variabili parametro possono avere dei **valori di default**, che vengono usati se, durante la chiamata, nessun argomento è assegnato a tale parametro

Es: funzione `def complex(real = 0.0, imag = 0.0)`

`x = complex(0) → 0.0`

`x = complex(imag = 5) → 5j (named parameters)`

`x = complex(real = 3) → 3 (named parameters)`

# Liste di argomenti variabili

- Eccezione: la sintassi speciale `*args` nella definizione di funzione
- Viene utilizzata per permettere di passare un numero variabile di argomenti a una funzione (e non è associata ad un nome). Una lista di parametri di questo genere è sempre positional

**\*objects = numero variabile di argomenti  
(positional parameter, «unnamed»)**

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

**named parameters**

# Quindi...

Argomento  
avanzato

```
1 # versioni valide (prima positional e poi named)
2 print(10, 20, sep = ":", end = " -- ")
3 print(10, 20, end = " -- ", sep = ":")
4 # errore: i positional non possono seguire i named
5 print(end = " -- ", sep = ":", 10, 20)
6
```

# Valori restituiti

---



5.4

# Valori restituiti

- Le funzioni possono (opzionalmente) restituire un valore
  - Aggiungere l'istruzione `return` che restituisce il valore
  - L'istruzione `return` fa due cose:
    - Termina immediatamente la funzione
    - Passa il `valore di ritorno` direttamente alla funzione che l'ha `invocata`

```
def cubeVolume (sideLength):  
    volume = sideLength ** 3  
    return volume
```

Istruzione return

*Il valore restituito può essere un valore, una variabile o il risultato di un calcolo*

# Restituire più valori

- Da una funzione può essere restituito solo un valore
- Se è necessario restituirne più d'uno, si utilizza una *tupla*, contenente i valori da restituire
- Esempio:
  - `return (x, y)`
  - Costruire la tupla `(x, y)`
  - Restituirla



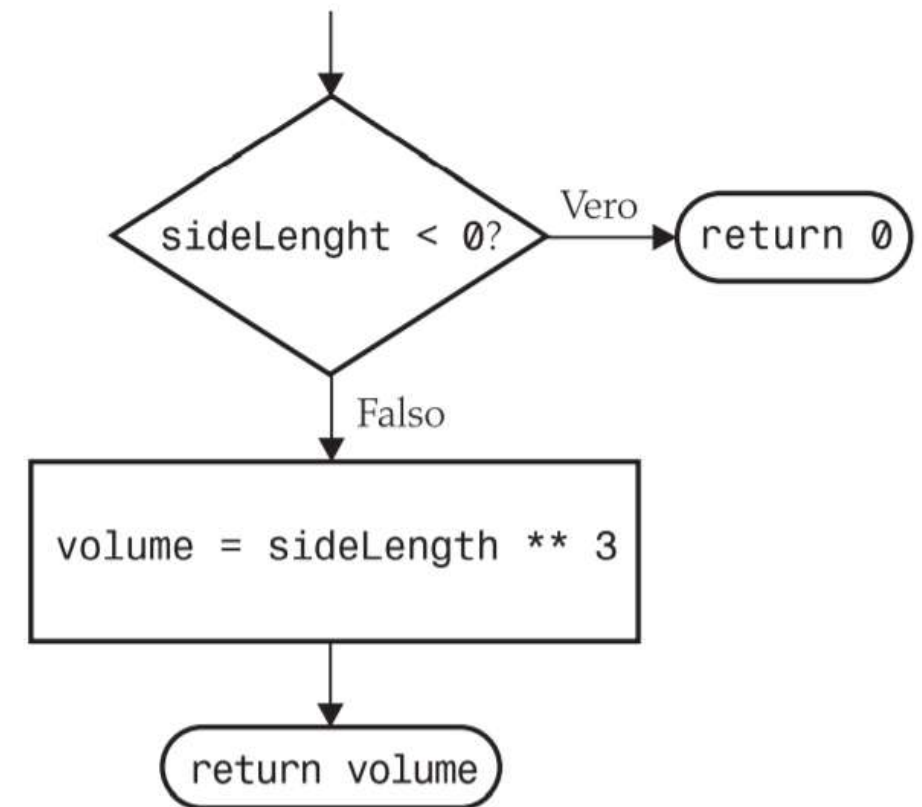
# Enunciati `return` multipli

- Una funzione può utilizzare **molte istruzioni `return`**
  - **Tutte le diramazioni** del flusso di esecuzione di una funzione devono restituire un valore, portando la funzione a incontrare un enunciato `return`

```
def cubeVolume(sideLength):  
    if (sideLength < 0):  
        return 0  
    else:  
        return sideLength ** 3
```

**Figura 4**

Un enunciato `return`  
termina la funzione  
immediatamente



# Enunciati `return` multipli (2)

- Alternativa ai `return` multipli (e.g., uno per ogni diramazione):
  - Si possono evitare i `return` multipli in questo modo:
    - Memorizzare il risultato in una variabile (anche in più punti della funzione)
    - Restituire il valore della variabile nell'ultima istruzione della funzione
  - Per esempio:

```
def cubeVolume(sideLength) :  
    if sideLength >= 0:  
        volume = sideLength ** 3  
    else :  
        volume = 0  
    return volume
```

# Assicurarsi che `return` gestisca tutti i casi

- Istruzione `return` mancante

- Assicurarsi che vengano gestite *tutte le condizioni verificabili*
- In questo caso, `sideLength` potrebbe essere minore di 0
  - Non serve l'istruzione `return` in questo caso
- Al compilatore non crea problemi se una diramazione non ha l'istruzione `return`
- Potrebbe far sorgere un errore di tempo di esecuzione siccome Python restituisce il valore speciale `None` quando ci si dimentica di far restituire un valore

```
def cubeVolume(sideLength) :  
    if sideLength >= 0 :  
        return sideLength ** 3  
    # Error—no return value if sideLength < 0
```

# Assicurarsi che `return` gestisca tutti i casi (2)

- Implementazione corretta:

```
def cubeVolume(sideLength) :  
    if sideLength >= 0 :  
        return sideLength ** 3  
    else :  
        return 0
```

# Implementare una funzione: passi

- Descrivere cosa deve fare la funzione
  - Fornire una semplice descrizione di cosa fa la funzione
  - “Calcola il volume di una piramide a base quadrata”
- Indicare una lista di tutti gli input della funzione
  - Fare una lista di tutti i parametri che possono cambiare
  - Non essere troppo specifici
- Determinare i tipi delle variabili parametro e del valore restituito

# Implementare una funzione: passi (2)

- Scrivere lo pseudocodice per ottenere il risultato voluto
  - Esprimere come formule matematiche, le diramazioni e i loop in pseudocodice
- Implementare il corpo della funzione

```
def pyramidVolume(height, baseLength) :  
    baseArea = baseLength * baseLength  
    return height * baseArea / 3
```

# Implementare una funzione: passi (3)

- Testare la funzione come elemento isolato (unit test)
  - Pensare ai valori di prova e al codice

```
Volume: 300  
Expected: 300  
Volume: 0  
Expected: 0
```

# Pyramids.py

- Aprire il file pyramids.py
- Vedere come è impostata la funzione principale per invocare la funzione pyramidVolume e visualizzare il risultato desiderato



# Funzioni che non restituiscono un valore



5.5

# Non restituire valori

- Se non serve restituire alcun valore, la funzione termina senza trasmettere un valore

`return     # nessun valore specificato`

- Se l'enunciato `return` non viene incontrato durante l'esecuzione della funzione, è equivalente ad avere un `return vuoto dopo l'ultima istruzione` della funzione

# Funzioni che non restituiscono un valore

- Le funzioni **non sono obbligate** a restituire un
  - Non è richiesto alcun'istruzione return
  - La funzione può generare un output (e.g., visualizzare con print) anche se non restituisce un valore

```
...  
boxString("Hello")  
...
```

```
-----  
!Hello!  
-----
```

```
def boxString(contents) :  
    n = len(contents)  
    print("-" * (n + 2))  
    print("!" + contents + "!")  
    print("-" * (n + 2))
```

# Usare `return` senza un valore

- Si può usare l'istruzione `return` senza un valore
  - La funzione `terminerà` immediatamente!

```
def boxString(contents) :  
    n = len(contents)  
    if n == 0 :  
        return # Termina immediatamente  
    print("-" * (n + 2))  
    print("!" + contents + "!")  
    print("-" * (n + 2))
```

# La funzione `main` e le funzioni nei programmi completi



5.2

# La funzione `main`

- Quando si definiscono e usano funzioni in Python, è bene **che tutte le istruzioni del programma si trovino all'interno di funzioni**, indicandone una come **punto di partenza** dell'esecuzione,
- Qualsiasi nome può essere per il punto di inizio, ma si sceglie per convenzione **'main'** siccome è un nome usato in altri linguaggi molto diffusi
- Ovviamente, serve **un'istruzione** nel programma che invochi la funzione **main**

# Sintassi: la funzione `main`

## *Esempio*

Per convenzione, `main` è il punto di partenza del programma.

```
def main() :  
    result = cubeVolume(2)  
    print("A cube with side length 2 has volume", result)
```

La funzione `cubeVolume` è definita più in basso.

```
def cubeVolume(sideLength) :  
    volume = sideLength ** 3  
    return volume
```

```
main()
```

Questo enunciato è al di fuori di tutte le definizioni

# Cubes.py con i commenti

```
1  ##
2  # Questo programma calcola i volumi di due cubi.
3  #
4
5  def main() :
6      result1 = cubeVolume(2)
7      result2 = cubeVolume(10)
8      print("A cube with side length 2 has volume", result1)
9      print("A cube with side length 10 has volume", result2)
10
11  ## Calcola il volume di un cubo.
12  # @param sideLength la lunghezza di un lato del cubo
13  # @return il volume del cubo
14  #
15  def cubeVolume(sideLength) :
16      volume = sideLength ** 3
17      return volume
18
19  # Inizio del programma.
20  main()
```

## Esecuzione del programma

```
A cube with side length 2 has volume 8
A cube with side length 10 has volume 1000
```



# Note

- In generale, l'invocazione della funzione `main`:
  - Dovrebbe essere eseguita se il programma è in esecuzione in modalità standalone
  - Non dovrebbe essere eseguita se il programma è importato, come modulo, in un programma più ampio
    - Bisogna controllare la variabile speciale `__name__`, che contiene il nome del modulo (o la stringa `'__main__'` se in modalità standalone)
    - Spesso si vede questo codice:

```
if __name__ == '__main__':  
    # call the main function if we are running in  
    # standalone mode  
    # don't call it if we are imported as a module  
    main()
```

# Note

- Molte variabili o funzioni interne di Python hanno nomi speciali e non sono (o non dovrebbero essere) normalmente utilizzate
  - I programmatori alle prime armi devono evitare di definire e usare variabili con nomi che inizino con ‘\_’
- Per evitare confusione, le variabili di sistema hanno nomi che iniziano e finiscono con doppio underscore
  - `_ _ n a m e _ _`
- Sono detti nomi “**dunder**” (che sta per **d**ouble-**u**nder**s**core)
  - `__name__` viene letto come *dunder-name*

# Usare le funzioni: ordine (1)

- È importante **definire** qualsiasi funzione prima di **invocarla**
- Per esempio, la seguente espressione genererà un errore di tempo di compilazione:

```
print(cubeVolume(10))
```

```
def cubeVolume(sideLength) :  
    volume = sideLength ** 3  
    return volume
```

- Il compilatore non sa che la funzione cubeVolume verrà definita **successivamente** nel programma
  - Non sa che funzione **invocare**
  - **NameError: name 'cubeVolume' is not defined**

# Usare le funzioni: ordine (2)

- Comunque, una funzione può essere invocata **dall'interno di un'altra funzione** prima di essere stata definita

- Il seguente esempio è corretto:

```
def main() :  
    result = cubeVolume(2) # 1  
    print("A cube with side length 2 has volume",  
          result)
```

```
def cubeVolume(sideLength) :  
    volume = sideLength ** 3  
    return volume
```

```
main() # 2
```

- In #1, la funzione `main` è appena stata **definita** (non ancora **eseguita**). Sarà invocata in #2, che è **dopo** la definizione di `cubeVolume`.

# Funzioni riutilizzabili

---



5.6

# Scrivere una funzione 'parametrizzata'

- Vediamo un esempio di **ripetizione di codice**
  - Può avere diversi valori, ma la stessa logica

0 - 23

```
hours = int(input("Enter a value between 0 and 23: "))  
while hours < 0 or hours > 23 :  
    print("Error: value out of range.")  
    hours = int(input("Enter a value between 0 and 23: "))
```

0 - 59

```
minutes = int(input("Enter a value between 0 and 59: "))  
while minutes < 0 or minutes > 59 :  
    print("Error: value out of range.")  
    minutes = int(input("Enter a value between 0 and 59: "))
```

# Write a 'Parameterized' Function

```
## Chiede all'utente di inserire un valore fino a un dato valore massimo, #  
ripetutamente finché non viene introdotto un valore valido.  
# @param high un numero intero, il valore massimo accettabile  
# @return il numero fornito dall'utente (tra 0 e high, compresi)  
#  
def readIntUpTo(high) :  
    value = int(input("Enter a value between 0 and " + str(high) + ": "))  
    while value < 0 or value > high :  
        print("Error: value out of range.")  
        value = int(input("Enter a value between 0 and " + str(high) + ": "))  
  
    return value
```

# Readtime.py

- Aprire il file readtime.py
- Provare il programma con diversi input
  - Come modifichereste il progetto per usare la funzione readInBetween?



# Ambito di visibilità delle variabili

---



5.8

# Ambito di visibilità delle variabili

- Le variabili possono essere dichiarate:

- Dentro una funzione

- Conosciute come ‘variabili locali’
    - Disponibili solo all’interno della funzione
    - Gli argomenti sono come variabili locali

- Fuori dalla funzione

- Talvolta detta ‘variabile globale’
    - Può essere usata (e modificata) in qualsiasi funzione

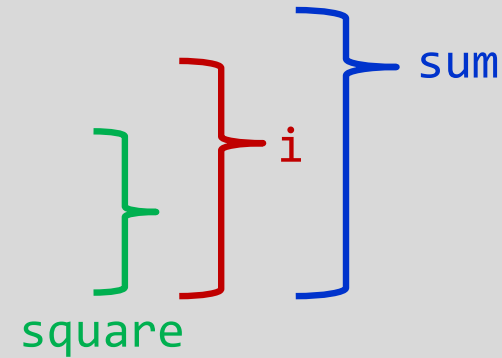
- Quale scegliete?

***L’ambito di visibilità di una variabile è la porzione del programma dove essa è **visible*****

# Esempi di ambiti di visibilità

- `sum`, `square` & `i` sono variabili locali in `main`

```
def main() :  
    sum = 0  
    for i in range(11) :  
        square = i * i  
        sum = sum + square  
    print(square, sum)
```



# Variabili locali delle funzioni

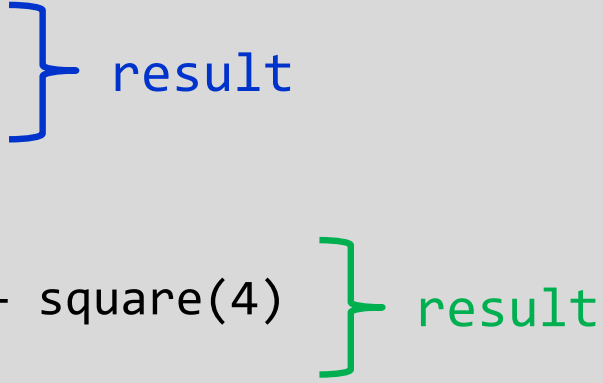
- Variabili dichiarate all'interno di una funzione e che non sono visibili da altre funzioni
  - `sideLength` è locale in `main`
  - Utilizzarla fuori dalla funzione causerà un errore di compilazione

```
def main():  
    sideLength = 10  
    result = cubeVolume()  
    print(result)  
  
def cubeVolume():  
    return sideLength * sideLength * sideLength # ERRORE
```

# Riutilizzare nomi per le variabili locali

- Le variabili dichiarate all'interno di una funzione non sono visibili da altre funzioni
  - `result` è la variabile locale di `square` e `result` è variabile locale di `main`
  - Sono **due variabili diverse** e non si sovrappongono
  - Questo può generare confusione

```
def square(n):  
    result = n * n  
    return result  
  
def main():  
    result = square(3) + square(4)  
    print(result)
```



# Variabili globali

- Sono variabili che sono definite all'esterno delle funzioni
- Una variabile globale è visibile a tutte le funzioni
- Comunque, qualsiasi funzione che voglia aggiornare una variabile globale deve includere una dichiarazione `global`

Non è una buona  
idea, meglio evitarlo

# Esempio di utilizzo di una variabile globale

Argomento  
avanzato

- Se si omette la dichiarazione `global`, la variabile `balance` utilizzata all'interno della funzione `withdraw` è considerata una variabile locale

```
balance = 10000    # A global variable

def withdraw(amount) :
    # Questa funzione aggiorna la variabile
    # globale balance
    global balance
    if balance >= amount :
        balance = balance - amount
```

Non è una buona  
idea, meglio evitarlo

# Suggestimenti

Argomento  
avanzato

- Ci sono alcuni casi in cui le variabili globali sono richieste (come `pi` definita nel modulo `math`), ma sono molto rari
- I programmi con le variabili globali sono difficili da gestire ed estendere, siccome non si può vedere ogni funzione come ‘scatola nera’ che riceve argomenti e restituisce un risultato
- Invece di usare variabili globali, usare, nelle funzioni, le variabili parametro e il valore restituito, che consentono di trasferire informazioni da un punto del programma a un altro



# Raffinamenti successivi

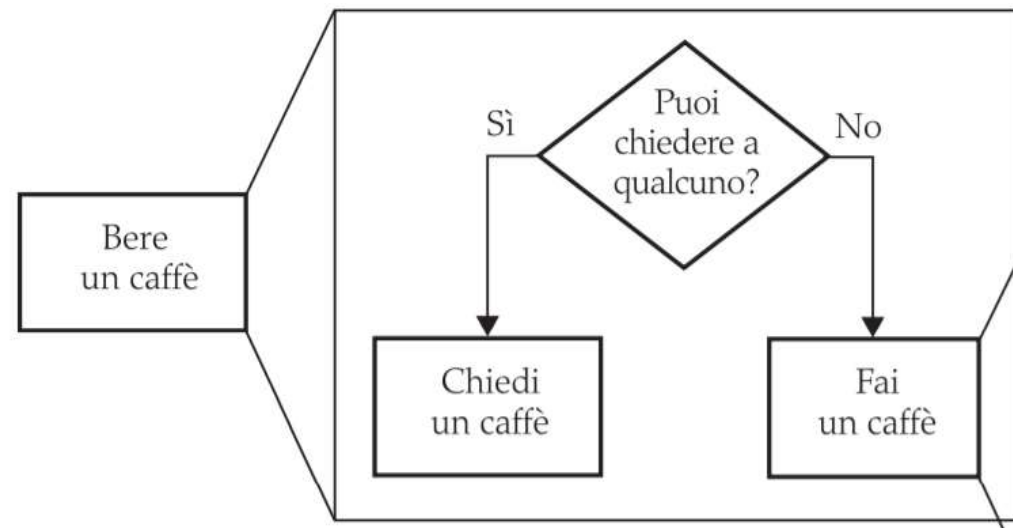
---



5.7

# Miglioramenti successivi

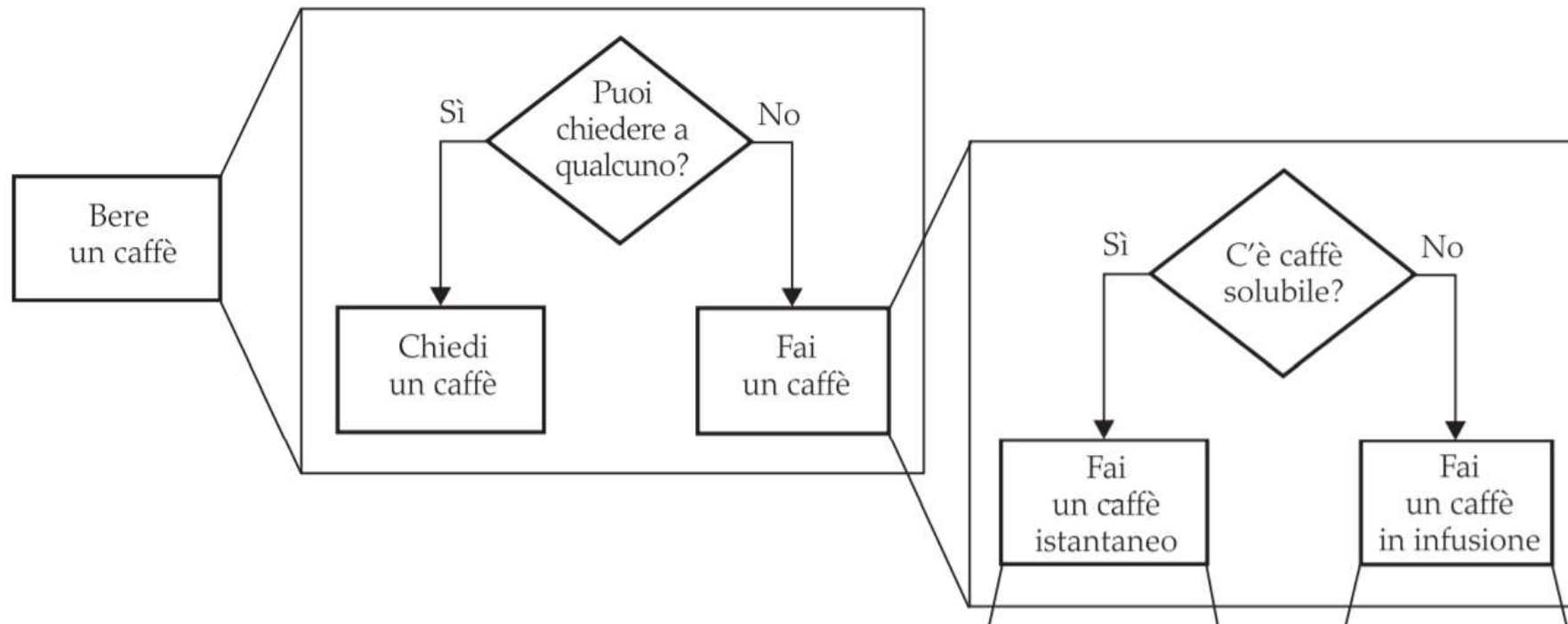
- Per risolvere un problema complesso, esso va scomposto in problemi più semplici
- Si continua a scomporre i problemi così generati in problemi ancora più semplici, finché si ottengono dei problemi che si sappiano risolvere



# Prendere un caffè

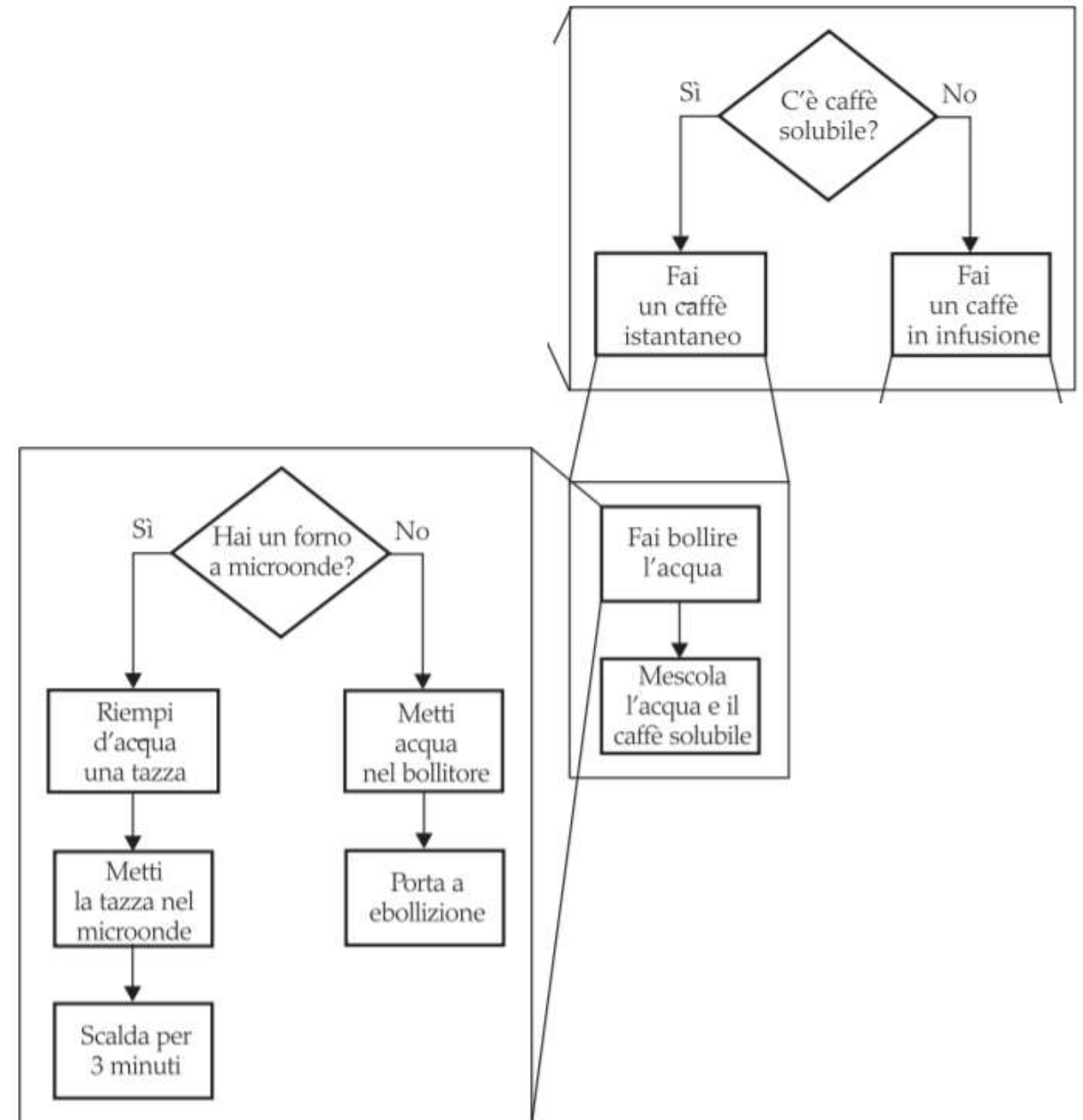
- Se si vuole fare un caffè ci sono due vie:
  - Fare un caffè istantaneo
  - Fare un caffè in infusione

Chiediamo scusa perché gli autori (americani) del libro di testo non hanno la minima idea di come si faccia un **\*vero\*** caffè



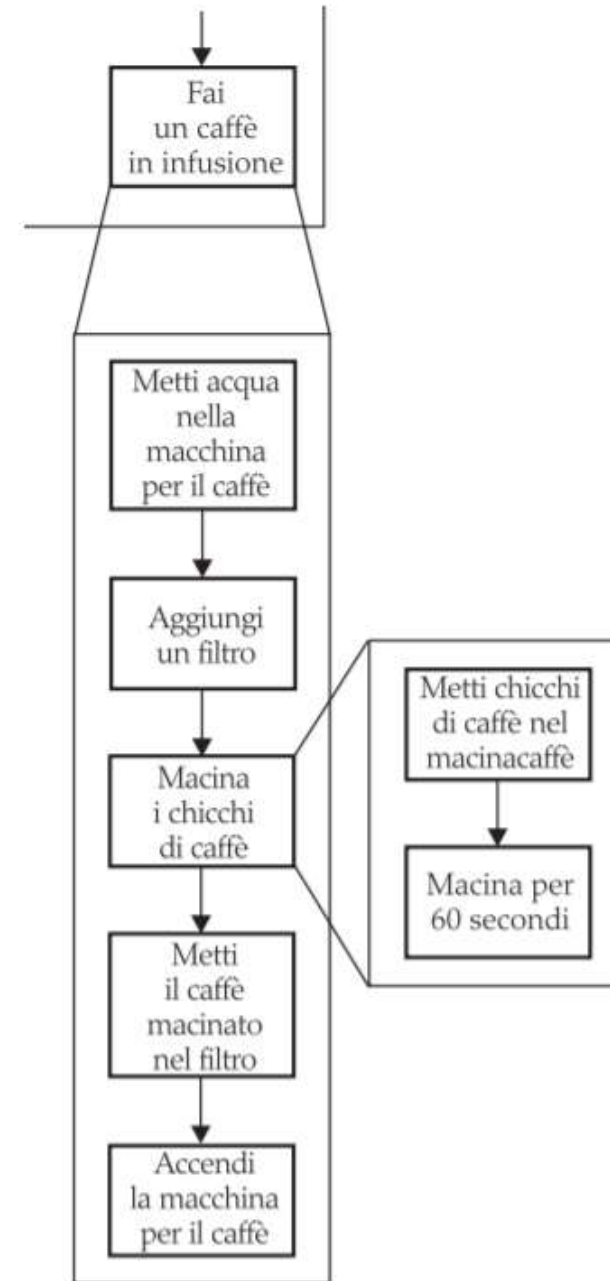
# Caffè istantaneo

- Due modi per bollire l'acqua
  - 1) Microonde
  - 2) Bollitore



# Caffè in infusione

- Ipotizzare l'utilizzo della macchinetta
  - Aggiungere acqua
  - Aggiungere il filtro
  - Macinare i chicchi
    - Aggiungere caffè macinato nel macinacaffè
    - Macina per 60 secondi
  - Mettere il caffè nel filtro
  - Accendere la macchinetta
- I passi sono facilmente eseguibili



# Esempio di miglioramenti successivi

- Quando si stampa un assegno bancario, solitamente si scrive l'importo sia in forma numerica (ad esempio, \$ 274.15) sia in parole ('duecentosettantaquattro dollari e quindici centesimi')
- Scrivere un programma per trasformare un numero in parole (in inglese)
  - Wow, sembra difficile!
  - Scomponetelo
  - Considerate la cifra 274 pensate ad un metodo
  - Prendete un intero tra 0 e 999
  - Convertitelo in stringa
  - Ancora abbastanza difficile...

# Esempio di miglioramenti successivi

- Prendere ogni cifra (2, 7, 4) – da sinistra a destra
- Maneggiare le prime cifre (le centinaia)
  - Se vuoto, si passa oltre
  - Prendere la prima cifra (un intero tra 1 – 9)
  - Prendere nome della cifra (“one”, “two”, “three”...)
  - Aggiungere le parola “hundred”
  - Sembra facile!
- Seconda cifra (le decine)
  - Prendere la seconda cifra (un intero tra 1 – 9)
  - Se 0, si passa alla terza cifra
  - Se 1, ... può essere undici, dodici... non facile!
  - Si considera ogni possibilità rimasta (2x-9x)...

# Esempio di miglioramenti successivi

- Se la seconda cifra è 0
  - Prendere la terza cifra (un intero tra 0 – 9)
  - Prendere il nome della cifra (“”, “one”, “two”...) ... come prima
  - Sembra facile!
- Se la seconda cifra è un 1
  - Prendere la terza cifra (un intero tra 0 – 9)
  - Restituire una stringa (“ten”, “eleven”, “twelve”...)
- Se la seconda cifra è 2-9
  - La stringa inizia con “twenty”, “thirty”, “forty”...
  - Prendere la terza cifra (un intero tra 0 – 9)
  - Prendere il nome della cifra (“”, “one”, “two”...) ... come prima
  - Sembra facile!



# Nominare i sotto-problemi

- **digitName**
  - Prende un intero tra 0 – 9
  - Restituisce una stringa (“”, “one”, “two”...)
- **tensName (seconda cifra  $\geq 20$ )**
  - Prende un intero tra 0 – 9
  - Restituisce una stringa(“twenty”, “thirty”...) plus
    - digitName(terza cifra)
- **teenName**
  - Prende un intero tra 0 – 9
  - Restituisce una stringa(“ten”, “eleven”...)

# Scrivere lo pseudocodice

part = number (la porzione che dobbiamo ancora convertire)  
name = "" (il nome del numero)

Se part  $\geq$  100  
    name = nome delle centinaia in part + " hundred"  
    Elimina le centinaia da part.

Se part  $\geq$  20  
    Aggiungi tensName(part) a destra di name.  
    Elimina le decine da part.  
Altrimenti se part  $\geq$  10  
    Aggiungi teenName(part) a destra di name.  
    part = 0

Se part  $>$  0  
    Aggiungi digitName(part) a destra di name.

***Identificare le funzioni che  
possono essere usate o  
riutilizzate per fare il lavoro!***

# Preparare le funzioni

- Scegliere un nome, i parametri, i tipi e il tipo di valore restituito
- `def intName (numero):`
  - Converte un numero nel suo nome in inglese
  - Restituisce una stringa che è la descrizione in inglese del numero (e.g., “seven hundred twenty nine”)
- `def digitName (cifra):`
  - Restituisce una stringa (“”, “one”, “two”...)
- `def tensName (numero):`
  - Restituisce una stringa(“twenty”, “thirty”...) plus
    - Return da `digitName(thirdDigit)`
- `def teenName (numero):`
  - Restituisce una stringa (“ten”, “eleven”...)

# Convertire in Python: funzione intName

- Aprire il file intname.py
- main invoca intName
  - Fa tutto il lavoro
  - Restituisce una stringa
- Utilizza le funzioni:
  - tensName
  - teenName
  - digitName

```
5 def main() :  
6     value = int(input("Please enter a positive integer < 1000: "))  
7     print(intName(value))  
8
```

# intName

```
13 def intName(number) :
14     part = number    # La parte che deve ancora essere convertita.
15     name = ""        # Il nome del numero.
16
17     if part >= 100 :
18         name = digitName(part // 100) + " hundred"
19         part = part % 100
20
21     if part >= 20 :
22         name = name + " " + tensName(part)
23         part = part % 10
24     elif part >= 10 :
25         name = name + " " + teenName(part)
26         part = 0
27
28     if part > 0 :
29         name = name + " " + digitName(part)
30
31     return name
```

# digitName

```
37 def digitName(digit) :  
38     if digit == 1 : return "one"  
39     if digit == 2 : return "two"  
40     if digit == 3 : return "three"  
41     if digit == 4 : return "four"  
42     if digit == 5 : return "five"  
43     if digit == 6 : return "six"  
44     if digit == 7 : return "seven"  
45     if digit == 8 : return "eight"  
46     if digit == 9 : return "nine"  
47     return ""
```

# teenName

```
53 def teenName(number) :  
54     if number == 10 : return "ten"  
55     if number == 11 : return "eleven"  
56     if number == 12 : return "twelve"  
57     if number == 13 : return "thirteen"  
58     if number == 14 : return "fourteen"  
59     if number == 15 : return "fifteen"  
60     if number == 16 : return "sixteen"  
61     if number == 17 : return "seventeen"  
62     if number == 18 : return "eighteen"  
63     if number == 19 : return "nineteen"  
64     return ""
```



# tensName

```
70 def tensName(number) :  
71     if number >= 90 : return "ninety"  
72     if number >= 80 : return "eighty"  
73     if number >= 70 : return "seventy"  
74     if number >= 60 : return "sixty"  
75     if number >= 50 : return "fifty"  
76     if number >= 40 : return "forty"  
77     if number >= 30 : return "thirty"  
78     if number >= 20 : return "twenty"  
79     return ""
```



# 💡 Suggerimenti

- Realizzare funzioni brevi
  - Se occupano più di uno schermo, spezzarle in 'sotto' funzioni
- Tenere traccia dell'esecuzione delle funzioni
  - Una linea per ogni step
  - Le colonne per le variabili
- Usare gli *stub* se si scrivono programmi ampi
  - Una funzione che restituisce un valore calcolato in modo semplice (anche sbagliato, o sempre lo stesso), è sufficiente per collaudare la funzione il prima possibile, nel contesto in cui viene chiamata

<i>intName(number = 416)</i>	
<i>part</i>	<i>name</i>
<del>416</del>	<del>---</del>
<del>16</del>	<del>"four hundred"</del>
0	"four hundred sixteen"

# Sommario

---

# Sommario: funzioni

- Una funzione è una sequenza di istruzioni a cui viene dato un nome
- Gli argomenti sono forniti quando la funzione è invocata
- Il valore restituito è il risultato calcolato dalla funzione
- Quando si dichiara una funzione le si forniscono un nome e una variabile per ogni argomento
- I commenti nelle funzioni spiegano lo scopo della funzione, il significato dei parametri e del valore restituito, così come requisiti speciali
- Le variabili parametro contengono gli argomenti forniti nell'invocazione della funzione

# Sommario: il return

- L'istruzione `return` termina l'invocazione della funzione e fornisce il suo risultato
- Utilizzare il processo per miglioramenti successivi per scomporre un problema complesso in problemi semplici
  - Quando ci serve una nuova funzione, scrivere una descrizione delle variabili parametro e dei valori restituiti
  - Una funzione può richiedere altre funzioni semplici per portare a termine il suo compito

# Sommario: l'ambito di visibilità

- L'ambito di visibilità di una variabile è la porzione di programma dove essa è visibile
  - Due variabili o parametri locali possono avere lo stesso nome, purché i loro ambiti di visibilità non si sovrappongano
  - Si può usare la stessa variabile con nomi diversi in funzioni diverse dato che i loro ambiti di visibilità non si sovrappongono
  - Le variabili locali dichiarate dentro una funzione non sono visibili al codice all'interno di altre funzioni