

---

Controllo del  
flusso in



# Blocchi di codice

---

- In Python i blocchi di codice non sono racchiusi tra parentesi graffe come in C o Java, oppure termina con end come in Scilab
- In Python per definire i blocchi di codice o il contenuto dei cicli si utilizza **l'indentazione**
  - Ciò migliora la leggibilità del codice, ma all'inizio può confondere il programmatore

# Spazi o tab

---

- Il metodo preferito è indentare utilizzando spazi (di norma 4)
- Il tab può essere diverso tra editor differenti

Stile per Codice Python

<https://www.python.org/dev/peps/pep-0008/#tabs-or-spaces>

# if elif ... else

---

```
if first_condition:
    first_body
elif second_condition:
    second_body
elif third_condition:
    third_body
else:
    fourth_body
```

:

indicano l'inizio del blocco (codice indentato)

elif ed else sono opzionali

```
if x < y and x < z:
    print('x è il minimo')
elif y < z:
    print('y è il minimo')
else:
    print('z è il minimo')
```

Se il blocco è costituito da una sola istruzione, allora può andare subito dopo i due punti

# Esempi

---

```
print('inizio')
if 5>3:
    print(1)
    print(2)
    print(3)
    print(4)
else:
    print(5)
    print(6)
    print(7)
print('fine')
```



```
inizio
1
2
3
4
fine
```

```
print('inizio')
if 5>3:
    print(1)
    print(2)
    print(3)
    print(4)
else:
    print(5)
    print(6)
    print(7)
print('fine')
```

**ERRORE**

# while

---

**while** *condition:*  
    *body*

```
j=0
while j < len(data) and data[j] != x:
    j += 1
```

```
while a<b:
    print(a)
    a=a+1
```

# for ... in

---

`for element in iterable:`

`body`

`# body may refer to 'element' as an identifier`

- “iterable” può essere:
  - Una sequenza:
    - Liste
    - Tuple
    - Stringhe
    - Dizionari
    - Classi definite dall'utente

```
total = 0
for val in data:
    total += val
```

```
biggest = 0
for val in data:
    if val > biggest:
        biggest = val
```

# range()

---

- range(n) genera una lista di interi compresi tra 0 ed n-1
  - range(start, stop, step)
- Utile quando vogliamo iterare in una sequenza di dati utilizzando un indice
  - **for** i **in** range(n)

```
>>> list(range(1,10,3))  
[1, 4, 7]
```

```
big_index = 0  
for j in range(len(data)):  
    if data[j] > data[big_index]:  
        big_index = j
```

```
for i in range(0, -10, -2): print(i)
```

```
0  
-2  
-4  
-6  
-8
```



# Esempi

---

```
# Stampa la lunghezza delle  
# parole in una lista  
words = ['cat', 'window', 'defenestrare']  
for w in words:  
    print(w, len(w))
```

```
cat 3  
window 6  
defenestrare 12
```

```
for _ in range(1,6):  
    print('ciao')
```

```
ciao  
ciao  
ciao  
ciao  
ciao
```

# Esempi

---

```
# Cicla su una copia della lista
for w in words[:]:
    if len(w) > 6:
        words.insert(0, w)
print(words)
```



```
['defenestrate', 'cat', 'window', 'defenestrate']
```

```
# Cicla su sulla stessa lista
for w in words:
    if len(w) > 6:
        words.insert(0, w)
print(words)
```

Crea una lista infinita

# break e continue

---

- **break** termina immediatamente un ciclo **for** o **while**, l'esecuzione continua dall'istruzione successiva al **while/for**
- **continue** interrompe la corrente iterazione di un ciclo **for** o **while** e continua verificando la condizione del ciclo

```
found = False
for item in data:
    if item == target:
        found = True
        break
```

# Clausola **else** e cicli

---

- Utilizzata con cicli che prevedono un **break**
- La clausola **else** è eseguita quando si esce dal ciclo ma **non a causa del break**

```
n=3
for x in [4, 5, 7, 8, 10]:
    if x % n == 0:
        print(x, ' è un multiplo di ', n)
        break
else:
    print('non ci sono multipli di', n , 'nella lista')
```

non ci sono multipli di 3 nella lista

Con n=2 l'output è

4 è un multiplo di 2

# Python: if *abbreviato*

---

- In C/Java/C++ esiste la forma abbreviata dell'if  
massimo = a > b ? a : b
- Anche Python supporta questa forma, ma la sintassi è differente

massimo = a if (a > b) else b

# List Comprehension

---



- *Comprensione di lista*
- Costrutto sintattico di Python che agevola il programmatore nella creazione di una lista a partire dall'elaborazione di un'altra lista
  - Si possono generare tramite comprehension anche
    - Insiemi
    - Dizionari

```
[ expression for value in iterable if condition ]
```

# [ *expression* **for** *value* **in** iterable **if** *condition* ]

- *expression* e *condition* possono dipendere da *value*
- La parte **if** è opzionale
- Si considerano tutti i *value* in iterable
  - Se *condition* è vera, il risultato di *expression* è aggiunto alla lista
- Equivalente a

```
result = [ ]  
for value in iterable:  
    if condition:  
        result.append(expression)
```

# Esempi

---

Lista dei quadrati dei numeri compresi tra 1 ed n

```
squares = [k*k for k in range(1, n+1)]
```

Lista dei divisori del numero n

```
factors = [k for k in range(1,n+1) if n % k == 0]
```

```
[str(round(pi, i)) for i in range(1, 6)]
```



```
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```



# *Doppia* comprehension

---

```
[(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
```




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

```
combs = []  
for x in [1,2,3]:  
    for y in [3,1,4]:  
        if x != y:  
            combs.append((x, y))
```

---

```
a = [(x, y) for x in [1,2,3] for y in ['a', 'b', 'c']]  
print(a)
```



```
[(1, 'a'), (1, 'b'), (1, 'c'), (2, 'a'), (2, 'b'), (2, 'c'),  
(3, 'a'), (3, 'b'), (3, 'c')]
```

# Altri tipi di comprehension

---

- list comprehension

$[ k*k \text{ for } k \text{ in range}(1, n+1) ]$

- set comprehension

$\{ k*k \text{ for } k \text{ in range}(1, n+1) \}$

- dictionary comprehension

$\{ k : k*k \text{ for } k \text{ in range}(1, n+1) \}$

---

Funzioni in



# Funzioni in Python

---

- Le funzioni sono definite usando la keyword **def**
- Viene introdotto un nuovo identificatore (il nome della funzione)
- Devono essere specificati
  - Il **nome** e la lista dei **parametri**
  - La funzione può avere un numero di parametri variabile
- L'istruzione **return** (opzionale) restituisce un valore ed interrompe l'esecuzione della funzione

# Esempi

---

```
def contains(data, target):  
    for item in data:  
        if item == target:  
            return True  
    return False
```

```
def count(data, target):  
    n = 0  
    for item in data:  
        if item == target:  
            n += 1  
    return n
```

```
def sum(values):  
    total = 0  
    for v in values:  
        total = total + v  
    return total
```

# Esempi

```
def bubble_sort(a):  
    n=len(a)  
    while(n>0):  
        for i in range(0,n-2):  
            if(a[i]>a[i+1]):  
                a[i], a[i+1] = a[i+1], a[i]  
        n -= 1  
    return a
```

Il parametro a è  
passato per  
riferimento

Assegnamento multiplo  
swap in un rigo

```
a = [5, 3, 1, 7, 8 ,2]  
print(a)  
bubble_sort(a)  
print(a)
```



```
[5, 3, 1, 7, 8, 2]  
[1, 2, 3, 5, 7, 8]
```

```
a = [5, 3, 1, 7, 8 ,2]  
print('a =', a)  
b = bubble_sort(a[:])  
print('b =', b)  
print('a =', a)
```



```
a = [5, 3, 1, 7, 8, 2]  
b = [1, 2, 3, 5, 7, 8]  
a = [5, 3, 1, 7, 8, 2]
```

# Stringa di documentazione

---

- La prima riga di codice nella definizione di una funzione dovrebbe essere una breve spiegazione di quello che fa la funzione
  - docstring

```
def my_function():  
    """Do nothing, but document it. ...  
       No, really, it doesn't do anything.  
    """  
  
    pass # Istruzione che non fa niente
```

```
print(my_function.__doc__)
```



```
Do nothing, but document it. ...  
No, really, it doesn't do anything.
```

# Variabili globali

---

- Nel corpo di una funzione si può far riferimento a variabili definite nell'ambiente (scope) esterno alla funzione, ma tali variabili non possono essere modificate
- Per poterle modificare bisogna dichiararle **global** nella funzione
- Se si prova ad accedere ad esse senza dichiararle **global** viene generato un errore



# Esempi

---

```
n = 111
def varGlobali():
    print('nella funzione n =', n)

varGlobali()
print('fuori la funzione n =', n)
```

nella funzione n = 111  
fuori la funzione n = 111

```
m = 999
def varGlobaliDue():
    m = 1
    print('nella funzione m =', m)

varGlobaliDue()
print('fuori la funzione m =', m)
```

nella funzione m = 1  
fuori la funzione m = 999

# Esempi

---

```
m=999
def varGlobaliTre():
    print('nella funzione m =', m)
    m = 1

varGlobaliTre()
print('fuori la funzione m =', m)
```

UnboundLocalError: local variable 'm' referenced before assignment

```
n = 777
def varGlobaliQuattro():
    global n
    print('nella funzione n =', n)
    n=3

print('fuori la funzione n =', n)
varGlobaliQuattro()
print('fuori la funzione n =', n)
```

fuori la funzione n = 777  
nella funzione n = 777  
fuori la funzione n = 3

# Esempio

```
z = 100
```

```
def a():  
    def b():  
        print('z =', z)
```

```
        z = 999
```

```
        b()
```

```
a()
```

Formattazione secondo lo stile Python  
Due righe vuote dopo fine  
assegnamenti/funzioni

Definizione funzione a()



```
z = 999
```

# Nomenclatura

---

- Parametri **formali** di una funzione
  - Identificatori usati per descrivere i parametri di una funzione nella sua definizione
- Parametri **attuali** di una funzione
  - Valori passati alla funzione all'atto della chiamata
  - Argomenti di una funzione
- Argomento **keyword**
  - Argomento preceduto da un identificatore in una chiamata a funzione
- Argomento **posizionale**
  - Argomento che non è un argomento keyword

# Passaggio dei parametri

---

- Il passaggio dei parametri avviene tramite un riferimento ad oggetti
  - Per valore, dove il valore è il riferimento (puntatore) dell'oggetto passato

```
lst = [1, 'due']  
  
def modifica(lista):  
    lista.append('nuovo')  
  
print('lista =', lst)  
modifica(lst)  
print('lista =', lst)
```



```
lista = [1, 'due']  
  
lista = [1, 'due', 'nuovo']
```

# Parametri di default

---

- Nella definizione della funzione, ad ogni parametro formale può essere assegnato un valore di default
  - a partire da quello più a destra
- La funzione può essere invocata con un numero di parametri inferiori rispetto a quello con cui è stata definita

```
def default(a, b=3):  
    print('a =', a, 'b =', b)
```

```
default(2)  
default(1,1)
```



```
a = 2 b = 3  
a = 1 b = 1
```

# Attenzione

---

- I parametri di default sono valutati nello scope in cui è definita la funzione

```
d = 666
def default_due(a, b=d):
    print('a =', a, 'b =', b)

d = 0
default_due(11)
default_due(22, 33)
```



```
a = 11 b = 666
a = 22 b = 33
```

# Attenzione

---

- I parametri di default sono valutati solo una volta (quando si definisce la funzione)
  - Attenzione a quando il parametro di default è un oggetto mutable

```
def f(a, L=[]):  
    L.append(a)  
    return L
```

```
print(f(1))  
print(f(2))  
print(f(3))
```

La lista L conserva il proprio valore tra chiamate successive, non è inizializzata ad ogni chiamata

```
[1]  
[1, 2]  
[1, 2, 3]
```



# Attenzione

---

- Se non si vuole che il parametro di default sia condiviso tra chiamate successive si può adottare la seguente tecnica (lo si inizializza nel corpo della funzione)

`L` è un segnaposto (variabile metasintattica)

```
def f(a, L=None):  
    if L is None:  
        L = []  
    L.append(a)  
    return L
```

```
print(f(1))  
print(f(2))  
print(f(3))
```



```
[1]  
[2]  
[3]
```

# Numero variabile di argomenti

---

- In Python si possono definire funzioni con un numero variabile di parametri
- L'ultimo parametro è preceduto da \*
- Dopo ci possono essere solo parametri keyword (dettagli in seguito)
- I parametri variabili sono passati in una tuple
  - Nel corpo della funzione possiamo accedere al valore dei parametri variabili tramite la posizione

# Esempio

---

```
def variabili(v1, v2=4, *arg):  
    print('primo parametro =', v1)  
    print('secondo parametro =', v2)  
    print('# argomenti passati', len(arg) + 2)  
    if arg:  
        print('# argomenti variabili', len(arg))  
        print('arg =', arg)  
        print('primo argomento variabile =', arg[0])  
    else:  
        print('nessun argomento in più')
```

`variabili(1, 'a', 4, 5, 7)`

```
primo parametro = 1  
secondo parametro = a  
# argomenti passati 5  
# argomenti variabili 3  
arg = (4, 5, 7)  
primo argomento variabile = 4
```

`variabili(3, 'b')`

```
primo parametro = 3  
secondo parametro = b  
# argomenti passati 2  
nessun argomento in più
```

# Parametri keyword


---

- Sono argomenti di una funzione preceduti da un identificatore oppure passati come dizionario (**dict**) preceduto da **\*\***
- Un argomento keyword può essere specificato anche assegnando esplicitamente, attraverso il nome, un parametro attuale ad un parametro formale
- Nella definizione di una funzione i parametri keyword possono essere rappresentati dall'ultimo parametro della funzione preceduto da **\*\***
  - Il parametro è considerato un dizionario (**dict**)

# Esempi

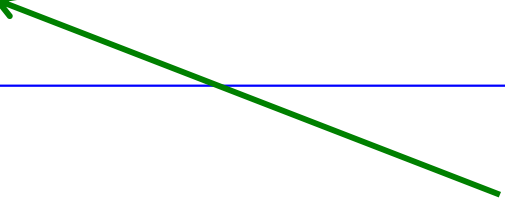
```
def f(v1, v2, v3=999):  
    print('v1 =', v1, 'v2 =', v2, 'v3 =', v3)  
  
f(1,2,3)  
f(1,2)  
f(v2=222,v1=111)    #parametri keyword
```

f(1) genera un errore

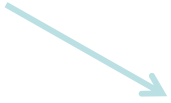


v1	=	1	v2	=	2	v3	=	3
v1	=	1	v2	=	2	v3	=	999
v1	=	111	v2	=	222	v3	=	999

```
def f(a='a', b='b', c='c'):  
    print('a =', a, 'b =', b, 'c =', c)  
  
f()  
f(b=2)  
f(c=3)
```



Evitiamo di specificare gli  
argomenti posizionali  
nell'ordine previsto



a	=	a	b	=	b	c	=	c
a	=	a	b	=	2	c	=	c
a	=	a	b	=	b	c	=	3

# Esempio

---

```
def esempio_kw(arg1, arg2, arg3, **cmd):
    if cmd.get('operando') == '+':
        print('La somma degli argomenti è: ', arg1 + arg3 + arg3)
    elif cmd.get('operando') == '*':
        print('Il prodotto degli argomenti è: ', arg1 * arg3 * arg3)
    else:
        print('Operando non supportato')

    if cmd.get('azione') == "stampa":
        print('arg1 =', arg1, 'arg2 =', arg2, 'arg3 =', arg3)
```

```
esempio_kw(2, 3, 4, operando='+')
```

La somma degli argomenti è: 10

```
esempio_kw(2, 3, 4, operando='*')
```

Il prodotto degli argomenti è: 32

```
esempio_kw(2, 3, 4, operando='/')
```

Operando non supportato

# Esempio

---

```
def esempio_kw(arg1, arg2, arg3, **cmd):  
    if cmd.get('operando') == '+':  
        print('La somma degli argomenti è: ', arg1 + arg3 + arg3)  
    elif cmd.get('operando') == '*':  
        print('Il prodotto degli argomenti è: ', arg1 * arg3 * arg3)  
    else:  
        print('Operando non supportato')  
  
    if cmd.get('azione') == "stampa":  
        print('arg1 =', arg1, 'arg2 =', arg2, 'arg3 =', arg3)
```

```
esempio_kw(2, 3, 4, operando='+', azione='stampa')
```

```
La somma degli argomenti è: 10  
arg1 = 2 arg2 = 3 arg3 = 4
```

# Esempio

---

```
def esempio_kw(arg1, arg2, arg3, **cmd):
    if cmd.get('operando') == '+':
        print('La somma degli argomenti è: ', arg1 + arg3 + arg3)
    elif cmd.get('operando') == '*':
        print('Il prodotto degli argomenti è: ', arg1 * arg3 * arg3)
    else:
        print('Operando non supportato')

    if cmd.get('azione') == "stampa":
        print('arg1 =', arg1, 'arg2 =', arg2, 'arg3 =', arg3)
```

```
esempio_kw(2, 3, 4, **{'operando': '+', 'azione': 'stampa'})
```

```
La somma degli argomenti è: 10
arg1 = 2 arg2 = 3 arg3 = 4
```



# Riassumendo

---

- Una funzione può essere definita con tutti e tre i tipi di parametri
  - Parametri posizionali
    - Non inizializzati e di default
  - Numero di parametri variabile
  - Parametri keyword


```
def tutti(arg1, arg2=222, *args, **kwargs):  
    #Corpo della funzione
```

# Esempio

---


```
def tutti(arg1, arg2=222, *args, **kwargs):  
    print('arg1          =', arg1)  
    print('arg2          =', arg2)  
    print('*args         =', args)  
    print('**kwargs      =', kwargs)
```

```
tutti('prova', 999, 'uno', 2, 'tre', a=1, b='sette')
```



```
arg1          = prova  
arg2          = 999  
*args         = ('uno', 2, 'tre')  
**kwargs      = {'a': 1, 'b': 'sette'}
```

```
tutti('seconda prova')
```



```
arg1          = seconda prova  
arg2          = 222  
*args         = ()  
**kwargs      = {}
```

# Funzioni come parametro di funzioni

---

- È possibile passare l'identificatore di una funzione **a** come parametro di un'altra funzione **b**
  - Si passa il riferimento alla funzione **a**
- Nel corpo della funzione **b**, si può invocare **a**
  - Come nome della funzione si usa il parametro formale specificato nella definizione della funzione **b**

# Esempio

riferimento a funzione

```
def insertion_sort(a):
    for i in range(1, len(a)):
        val = a[i]
        j = i - 1
        while (j >= 0 and a[j] > val):
            a[j + 1] = a[j]
            j = j - 1
            a[j + 1] = val
    return a
```

```
def ordina(lista, metodo, copia=True):
    if copia == True:
        #si ordina una copia della
        #lista
        return metodo(lista[:])
    else:
        return metodo(lista)
```

```
a = [5, 3, 1, 7, 8, 2]
print('a =', a)
b = ordina(a, insertion_sort)
print('a =', a)
print('b =', b)
print('-----')
a = [5, 3, 1, 7, 8, 2]
print('a =', a)
b = ordina(a, bubble_sort, copia=False)
print('a =', a)
print('b =', b)
```

```
a = [5, 3, 1, 7, 8, 2]

a = [5, 3, 1, 7, 8, 2]
b = [1, 2, 3, 5, 7, 8]
-----

a = [5, 3, 1, 7, 8, 2]

a = [1, 2, 3, 5, 7, 8]
b = [1, 2, 3, 5, 7, 8]
```

# Funzioni Python built-in

---

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

# Input: funzione input

---

- Riceve input da tastiera
- Può mostrare un cursore opzionale specificato come stringa
- Quello che viene letto è considerato stringa
  - Potrebbe dover essere convertito al tipo richiesto
- L'input termina con la pressione di invio (\n) che non viene inserito nella stringa letta

# Esempi

---

```
a = input('Inserisci un valore: ')\nprint(a, type(a))
```



```
Inserisci un valore: e\n                      e <class 'str'>
```

```
a = input('Inserisci un valore: ')\nprint(a, type(a))
```



```
Inserisci un valore: 12\n                      12 <class 'str'>
```

```
a = int(input('Inserisci un valore: '))\nprint(a, type(a))
```



```
Inserisci un valore: 14\n                      14 <class 'int'>
```

# Gestione dei file

---

- I file sono aperti con la funzione built-in **open**
  - Aperti in lettura di default
  - Open restituisce un oggetto con cui manipolare il file
- Maggiori dettagli nella sezione **11. File and Directory Access** del manuale **The Python Standard Library**
  - <https://docs.python.org/2.7/library/filesys.html>



# Accesso ai file (1)

---

- I file vengono gestiti in modo molto semplice e simile al C
- `open(nomefile[modo])` apre “nomefile” in modalità “modo” (“r” è il valore di default) e ritorna un oggetto di tipo “file”
- I metodi sono gli stessi del C
- I metodi principali degli oggetti file sono:
  - `read([n])` ritorna “n” byte dal file. Se “n” è omissso legge tutto il file
  - `readline()` ritorna una riga
  - `readlines()` ritorna una lista con le righe rimanenti nel file
  - `write(data)` scrive “data” sul file

## Accesso ai file (2)

---

- `writelines(list)` scrive tutti gli elementi di `list` su file
- `close()` chiude il file (richiamato automaticamente dall'interprete)
- `flush()` scrive su disco i dati presenti in eventuali buffer
- `seek(offset, posiz)` muove di `offset` byte da `posiz`. I valori di `posiz` sono:
  - 0: dall'inizio del file (valore di default)
  - 1: dalla posizione corrente
  - 2: dalla fine del file (`offset` è normalmente negativo)
- `tell()` ritorna la posizione corrente
- `truncate([n])` tronca il file a non più di `n` byte. Il valore di default è la posizione corrente

---

Moduli in



# I moduli in Python

---

- Un modulo è un particolare script Python
  - È uno script che può essere utilizzato in un altro script
  - Uno script incluso in un altro script è chiamato modulo
- Sono utili per decomporre un programma di grande dimensione in più file, oppure per riutilizzare codice scritto precedentemente
  - Le definizioni presenti in un modulo possono essere importate in uno script (o in altri moduli) attraverso il comando **import**
  - Il nome di un modulo è il nome del file script (esclusa l'estensione `'.py'`)
  - All'interno di un modulo si può accedere al suo nome tramite la variabile globale `__name__`

# Moduli esistenti

---

- Esistono vari moduli già disponibili in Python

Existing Modules	
Module Name	Description
array	Provides compact array storage for primitive types.
collections	Defines additional data structures and abstract base classes involving collections of objects.
copy	Defines general functions for making copies of objects.
heapq	Provides heap-based priority queue functions (see Section 9.3.7).
math	Defines common mathematical constants and functions.
os	Provides support for interactions with the operating system.
random	Provides random number generation.
re	Provides support for processing regular expressions.
sys	Provides additional level of interaction with the Python interpreter.
time	Provides support for measuring time, or delaying a program.

# Utilizzare i moduli

---

- All'interno di un modulo/script si può accedere al nome del modulo/script tramite l'identificatore `__name__`
- Per utilizzare un modulo deve essere incluso tramite l'istruzione **import**
  - **import** `math`
- Per far riferimento ad una funzione del modulo importato bisogna far riferimento tramite il nome qualificato completamente
  - `math.gdc(7,21)`

# Utilizzare i moduli

---

- Con l'istruzione **from** si possono importare singole funzioni a cui possiamo far riferimento direttamente con il loro nome
  - **from** math **import** sqrt
  - **from** math **import** sqrt, floor

```
import math
print(math.gcd(7,21))

from math import sqrt
print(sqrt(3))
```



7

1.7320508075688772

```
from math import *
```

tutte le funzioni di **math** sono importate

# Caricamento moduli

---

- Ogni volta che un modulo è caricato in uno script è eseguito
- Il modulo può contenere funzioni e codice *libero*
- Le funzioni sono *interpretate*, il codice libero è eseguito
- Lo script che importa (eventualmente) altri moduli ed è eseguito per primo è chiamato dall'interprete Python \_\_main\_\_
- Per evitare che del codice *libero* in un modulo sia eseguito quando il modulo è importato dobbiamo controllare che il nome del modulo sia \_\_main\_\_
  - In tal caso eseguire il codice libero



# Esempio

## testNoMain.py

```
def modifica(lista):  
    lista.append('nuovo')  
  
lst = [1, 'due']  
print('lista =', lst)  
modifica(lst)  
print('lista =', lst)
```

## esecuzione testNoMain.py

```
lista = [1, 'due']  
lista = [1, 'due', 'nuovo']
```

## test.py

```
def modifica(lista):  
    lista.append('nuovo')  
  
if __name__ == '__main__':  
    lst = [1, 'due']  
    print('lista =', lst)  
    modifica(lst)  
    print('lista =', lst)
```

## esecuzione test.py

```
lista = [1, 'due']  
lista = [1, 'due', 'nuovo']
```

# Esempio

importUNO.py

```
import test
lista = [3,9]
print(lista)
test.modifica(lista)
print(lista)
```

esecuzione importUNO.py

```
[3, 9]
[3, 9, 'nuovo']
```

importDUE.py

```
import testNoMain
lista = [3,9]
print(lista)
testNoMain.modifica(lista)
print(lista)
```

importDUE.py

```
def modifica(lista):
    lista.append('nuovo')
```

```
lst = [1, 'due']
print('lista =', lst)
modifica(lst)
print('lista =', lst)
```

esecuzione importDUE.py

```
lista = [1, 'due']
lista = [1, 'due', 'nuovo']
[3, 9]
[3, 9, 'nuovo']
```

```
lista = [3,9]
print(lista)
testNoMain.modifica(lista)
print(lista)
```

# Moduli popolari per python

---

- **matplotlib.pyplot**: è una libreria per la creazione dei grafici
  - **Link:**<https://matplotlib.org/>
- **numpy**: introduce oggetti per array multidimensionali e matrici ed operazioni statistiche
  - **Link:**<http://www.numpy.org/>
- **scipy**: collezione di funzioni potenti per algebra lineare, equazioni differenziali, ottimizzazione, etc
  - **Link:**<https://www.scipy.org/scipylib/>
- **pandas**: strutture dati e strumenti per lavorare con tabelle
  - **Link:**<http://pandas.pydata.org/>
- **snap**: gestione ed analisi di reti
  - **Link:**<http://snap.stanford.edu/snappy/index.html>

# Come importare ed usare moduli

---

- Per usare le funzioni definite in determinate librerie di Python è necessario importare tali librerie nel programma.

- La sintassi è la seguente:

`import nome_libreria`

- Talvolta, per abbreviare il nome della libreria si definisce un alias tramite la parola chiave `as` dopo il nome della libreria, seguita dall'alias: `import nome_libreria as alias`

- `import math as m`
- `import numpy as np`
- `import matplotlib.pyplot as plt`

- In questo modo, se volessimo utilizzare

- la funzione `pow()` della libreria `math`, potremmo scrivere direttamente

`m.pow()` anziché `math.pow()`.