

**Programmazione avanzata a.a 2019-20**  
A. De Bonis  
**Introduzione a Python**  
**(III parte)**

Programmazione Avanzata a.a. 2019-20  
A. De Bonis

1

**Funzioni in Python**

Programmazione Avanzata a.a. 2019-20  
A. De Bonis

2

## 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-1):
            if(a[i]>a[i+1]):
                a[i], a[i+1] = a[i+1], a[i]
            n -= 1
    return a
```

Assegnamento multiplo  
swap in un rigo

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

Il parametro a è passato  
per riferimento

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

↓

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

↓

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

Programmazione Avanzata a.a. 2019-20  
A. De Bonis

5

## 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.
```

Programmazione Avanzata a.a. 2019-20  
A. De Bonis

6

## 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

Programmazione Avanzata a.a. 2019-20  
A. De Bonis

7

## Esempi

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

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



```
nella funzione n = 111
fuori la funzione n = 111
```

```
m = 999
def f1():
    m = 1
    print('nella funzione m =', m)

f1()
print('fuori la funzione m =', m)
```



```
nella funzione m = 1
fuori la funzione m = 999
```

Programmazione Avanzata a.a. 2019-20  
A. De Bonis

8

## Esempi

```

m=999
def f3():
    print('nella funzione m =', m)
    m = 1

f3()
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

Programmazione Avanzata a.a. 2019-20  
A. De Bonis

9

## 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

Programmazione Avanzata a.a. 2019-20  
A. De Bonis

10

## 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

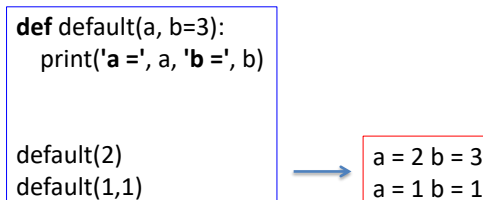


Programmazione Avanzata a.a. 2019-20  
A. De Bonis

11

## 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



Programmazione Avanzata a.a. 2019-20  
A. De Bonis

12

## 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

```
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]
```

Programmazione Avanzata a.a. 2019-20  
A. De Bonis

15

## 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)
- Il parametro formale preceduto da \* indica la sequenza in cui sono contenuti un numero variabile di parametri
  - Nel corpo della funzione possiamo accedere al valore di questi parametri tramite la posizione

Programmazione Avanzata a.a. 2019-20  
A. De Bonis

16



## 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
```

## L'operatore \*

- Ogni tipo iterabile può essere spaccettato usando l'operatore \* (unpacking operator).
- Se in un assegnamento con due o più variabili a sinistra dell'assegnamento, una di queste variabili è preceduta da \* allora i valori a destra sono assegnati uno ad uno alle variabili (senza \*) a sinistra e i restanti valori vengono assegnati alla variabile preceduta da \*.
- Possiamo passare come argomento ad una funzione che ha k parametri posizionali una collezione iterabile di k elementi preceduta da \*
  - Questo è diverso dal caso in cui utilizziamo \* davanti ad un parametro formale nella definizione della funzione

## Esempi di uso di \*

```
>>> primo, secondo, *rimanenti = [1,2,3,4,5,6]
>>> primo
1
>>> secondo
2
>>> rimanenti
[3, 4, 5, 6]
```

```
>>> primo, *rimanenti, sesto, = [1,2,3,4,5,6]
>>> primo
1
>>> sesto
6
>>> rimanenti
[2, 3, 4, 5]
```

## Esempi di uso di \*

```
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)

L=[4,5,7]  
variabili(1,'a',\*L)

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

## Esempi di uso di \*

```
def somma(addendo1, addendo2, addendo3):
    return addendo1+addendo2+addendo3

addendi=[56,2,4]

print("somma =",somma(*addendi))
```

**Attenzione:**  
**addendi** deve  
 contenere  
 esattamente 3  
 elementi

```
somma = 62
```

## Unpacking

- Quando a sinistra di un assegnamento ci sono due o piu` variabili e a sinistra c'è una sequenza, la collezione viene spaccettata e gli elementi assegnati alle variabili a sinistra
  - Lo abbiamo già visto per le tuple
- Esempio:

```
>>> l=[1,2,3,4]
>>> a,b,c,d = l
>>> a
1
>>> b
2
>>> c
3
>>> d
4
```

## 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**)

## L'operatore **\*\***

- L'operatore **\*\*** è il mapping unpacking operator e può essere applicato ai tipi mapping (collezione di coppie chiave-valore) quali i dizionari per produrre una lista di coppie chiavi-valore adatta ad essere passata come argomento ad una funzione.

## Esempio

Qui `cmd` è un dizionario

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

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

## Esempio

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

La somma degli argomenti è: 9

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

Il prodotto degli argomenti è: 24

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

operazione non supportata

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

La somma degli argomenti è: 9  
arg1 = 2 arg2 = 3 arg3 = 4

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

La somma degli argomenti è: 9  
arg1 = 2 arg2 = 3 arg3 = 4

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

La somma degli argomenti è: 9  
arg1 = 2 arg2 = 3 arg3 = 4

## Esempio

Parametri variabili      Parametro keyword

```
def concat(*args, sep="/"):
    return sep.join(args)

print(concat('ciao','a','tutti', sep='/'))
print(concat('ciao','a','tutti', sep='.'))
```

```
ciao/a/tutti
ciao.a.tutti
```

Programmazione Avanzata a.a. 2019-20  
A. De Bonis

27

## Riassumendo

- Ci sono due modi per assegnare valori ai parametri formali di una funzione
- Secondo la **posizione**
  - Parametri *tradizionali*
  - Parametri di default
  - Gli argomenti posizionali non hanno keyword e devono essere assegnati per primi
  - La posizione è importante
- Secondo la **keyword**
  - Gli argomenti keyword hanno keyword e sono assegnati in seguito, dopo i parametri posizionali
  - La posizione non è importante
    - f('casa', a=3, b=7) è la stessa cosa di f('casa', b=7, a=3)

```
def f(x, a, b): ...
```

Programmazione Avanzata a.a. 2019-20  
A. De Bonis

28

## Riassumendo

- Una funzione può anche 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 = {}
```

## Annotazioni

- Le annotazioni sono dei metadati associati alle funzioni definite dal programmatore
- Sono memorizzate come un dizionario nell'attributo `__annotation__` della funzione
- Non hanno nessun effetto sulla funzione
- Servono ad indicare il tipo dei parametri e del valore eventualmente restituito

## Annotazioni

- L'**annotazione di parametri** è definita da `:` dopo il nome del parametro seguito da un'espressione che, una volta valutata, indica il tipo del valore dell'annotazione
- Le **annotazioni di ritorno** sono definite da `->` seguita da un'espressione e sono poste tra la lista dei parametri e i due punti che indicano la fine dell'istruzione `def`



## Esempio

```
def saluta(nome: str, età: int = 23) -> str:
    print('Ciao ', nome, 'hai ', età, ' anni')
    return nome + ' ' + str(età)
```

```
s=saluta('mario')
print(s)
s=saluta('luisa', 21)
print(s)
```

```
Ciao mario hai 23 anni
mario 23
Ciao luisa hai 21 anni
luisa 21
```

```
print(saluta.__annotations__)
```

```
{'età': <class 'int'>, 'nome': <class 'str'>, 'return': <class 'str'>}
```

Programmazione Avanzata a.a. 2019-20  
A. De Bonis

33

## A cosa servono?

- Potrebbero essere utilizzate come help della funzione

```
def saluta(nome: 'rappresenta il nome dell\'utente ', età: int = 23) -> str:
    print('Ciao ', nome, 'hai ', età, ' anni')
    return nome + ' ' + str(età)
```

```
print(saluta.__annotations__)
```

```
{'età': <class 'int'>, 'nome': "rappresenta il nome dell'utente ", 'return': <class 'str'>}
```

Programmazione Avanzata a.a. 2019-20  
A. De Bonis

34

## 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**

```
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
```

riferimento a funzione

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

## Esempio

```
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]
```

## Espressioni **lambda**

- Funzioni anonime create usando la keyword **lambda**
- **lambda** *a,b,c* : *a + b + c*
  - Restituiscono la valutazione dell'espressione presente dopo i due punti
    - Può essere presente solo un'istruzione
  - Possono far riferimento a variabili presenti nello scope (ambiente) in cui sono definite
  - Possono essere restituite da funzioni
    - Una funzione che restituisce una funzione
  - Possono essere assegnate ad un identificatore
- Maggiori dettagli in seguito

Programmazione Avanzata a.a. 2019-20  
A. De Bonis

37

## Esempi

```
def f(x): return x**2
g = lambda x: x**2
```

```
print(g(3))
print(f(3))
```

f e g sono equivalenti, nel senso che producono lo stesso risultato

```
9
9
```

```
dati = [1, -4, 2, 7, -10, -3]
print(dati)
```

```
dati.sort(key=lambda x: abs(x))
print(dati)
```

Ordina la lista **dati** considerando il valore assoluto degli elementi

```
[1, -4, 2, 7, -10, -3]
```

```
[1, 2, -3, -4, 7, -10]
```

Programmazione Avanzata a.a. 2019-20  
A. De Bonis

38

## 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/Output

## Output: funzione `print`

- Riceve un numero variabile di parametri da stampare e due parametri keyword (`end` e `sep`)
- Aggiunge automaticamente `\n` alla fine dell'output
- Parametri keyword (opzionali)
  - `sep` - stringa di separazione dell'output (default spazio)
  - `end` - stringa finale dell'output (default `\n`)
- Gli argomenti ricevuti sono convertiti in stringhe, separati da `sep` e seguiti da `end`

## Esempi

```
dati = [1, -4, 2, 7, -10, -3]
a = 1
b = 'a'
c = 'casa'
```

```
print(a, b, c, dati)
print(a, b, c)
print(dati)
print(a, b, c, dati, sep=':')
```

```
1 a casa [1, -4, 2, 7, -10, -3]
1 a casa
[1, -4, 2, 7, -10, -3]
1:a:cas:[1, -4, 2, 7, -10, -3]
```

```
for v in dati:
    print(v)
```

```
-4
2
7
-10
-3
```

```
for v in dati:
    print(v, end=' ')
```

```
1 -4 2 7 -10 -3 1
```

## Output formattato

```
print('{} {}'.format('primo', 'secondo'))
print('{0} {1}'.format('primo', 'secondo'))
print('{1} {0}'.format('primo', 'secondo'))
print('{2} {0}'.format('primo', 'secondo', 'terzo'))
```

```
primo secondo
primo secondo
secondo primo
terzo primo
```

## Output formattato

- Esempio di uso di format con parametri keywords

```
>>> d={"parola1": "ciao", "parola2": "?"}
>>> s="{parola1} Laura, come va {parola2}".format(**d)
>>> s
'ciao Laura, come va ?'
```

```
>>> s="{parola1} Laura, come va {parola2}".format(parola1="ciao", parola2="?")
>>> s
'ciao Laura, come va ?'
```

```
>>> s="{parola1} Laura, come va {parola2}".format(parola2="?", parola1="ciao")
>>> s
'ciao Laura, come va ?'
```

## Output formattato

- Consultare
  - <https://docs.python.org/3/tutorial/inputoutput.html#reading-and-writing-files>
- Oppure consultate il tutorial più immediato presso
  - <https://pyformat.info/>

## 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: ')
print(a, type(a))
```

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

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

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

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

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

## Lettura e scrittura di file

- La funzione built-in `open()` restituisce un file object che ci permette di agire sui file
- Comunemente `open()` e` invocato con due argomenti:
  - `open(filename,mode)`
  - Esempio: `p=open("file.txt","w")`
- Il primo argomento `filename` e` la stringa contenente il nome del file
- Il secondo argomento `mode` e` una piccola stringa che descrive in che modalita` deve essere aperto il file
  - `'r'` : modalita` di sola lettura
  - `'w'` : modalita` di sola scrittura; se il file non esiste lo crea; se il file gia` esiste il suo contenuto viene cancellato
  - `'a'` : modalita` di append; se il file non esiste lo crea; se il file gia` esiste il suo contenuto viene non cancellato
  - `'r+'` : modalita` di lettura e scrittura; il contenuto del file non viene cancellato
  - Se il secondo argomento non e` specificato viene utilizzato il valore di default che e` `'r'`



## Letture e scrittura di file

Esempio: file.txt inizialmente vuoto

```
>>> fp=open("file.txt",'r+')
>>> fp.write("cominciamo a scrivere nel file")
30
>>> fp.write("\nvado al prossimo rigo")
22
```

## Letture e scrittura di file

- Possiamo usare `close()` per chiudere il file e liberare immediatamente qualsiasi risorsa di sistema usata per tenerlo aperto.
- Se il file non venisse chiuso esplicitamente, il garbage collector di Python ad un certo punto distruggerebbe il file object e chiuderebbe il file.
  - Cio` potrebbe avvenire pero` dopo molto tempo.
    - Dipende dall'implementazione di Python che stiamo utilizzando
- Dopo aver chiuso il file non e` possibile accedere in lettura o scrittura al file

## Letture e scrittura di file

Esempio (stesso file di prima)

```
>>> fp.close()
```

```
>>> fp.readline()
```

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

ValueError: I/O operation on closed file.

## Funzioni sui file

Calling Syntax	Description
<code>fp.read()</code>	Return the (remaining) contents of a readable file as a string.
<code>fp.read(k)</code>	Return the next $k$ bytes of a readable file as a string.
<code>fp.readline()</code>	Return (remainder of) the current line of a readable file as a string.
<code>fp.readlines()</code>	Return all (remaining) lines of a readable file as a list of strings.
<code>for line in fp:</code>	Iterate all (remaining) lines of a readable file.
<code>fp.seek(k)</code>	Change the current position to be at the $k^{\text{th}}$ byte of the file.
<code>fp.tell()</code>	Return the current position, measured as byte-offset from the start.
<code>fp.write(string)</code>	Write given string at current position of the writable file.
<code>fp.writelines(seq)</code>	Write each of the strings of the given sequence at the current position of the writable file. This command does <i>not</i> insert any newlines, beyond those that are embedded in the strings.
<code>print(..., file=fp)</code>	Redirect output of print function to the file.

## Letture e scrittura di file

```
Esempio:
>>> f=open("newfile",'w')
>>> f.write("prima linea\n")
12
>>> f.write("seconda linea\n")
14
>>> f.write("terza linea\n")
12
>>> f.write("quarta linea\n")
13
>>> f.close()
>>> f=open('newfile','r')
>>> for line in f:
...     print(line)
...
prima linea

seconda linea

terza linea

quarta linea
```

Contenuto di newfile

```
prima linea
seconda linea
terza linea
quarta linea
```

## Letture e scrittura di file

Esempio: continua dalla slide precedente

```
>>> f.seek(0)
0
>>> f.readline()
'prima linea\n'
>>> for linea in f:
...     print(linea)
...
seconda linea

terza linea

quarta linea
```

Contenuto di newfile

```
prima linea
seconda linea
terza linea
quarta linea
```

## Gestione dei file

- Maggiori dettagli in
  - <https://docs.python.org/3/library/filesys.html>

## Moduli e Pacchetti

## Namespace

- Quando si utilizza un identificativo si attiva un processo chiamato risoluzione del nome (*name resolution*) per determinare il valore associato all'identificativo
- Quando si associa un valore ad un identificativo tale associazione è fatta all'interno di uno scope
- Il **namespace** (spazio dei nomi) gestisce tutti i nomi definiti in uno scope (ambito)

Programmazione Avanzata a.a. 2019-20  
A. De Bonis

57

## Namespace

- Python implementa il namespace tramite un dizionario che mappa ogni identificativo al suo valore
- Uno scope può contenere al suo interno altri scope
- **Non c'è nessuna relazione tra due identificatori che hanno lo stesso nome in due namespace differenti**
- Tramite le funzioni **dir()** e **vars()** si può conoscere il contenuto del namespace dove sono invocate
  - **dir** elenca gli identificatori nel namespace
  - **vars** visualizza tutto il dizionario

Programmazione Avanzata a.a. 2019-20

A. De Bonis

58

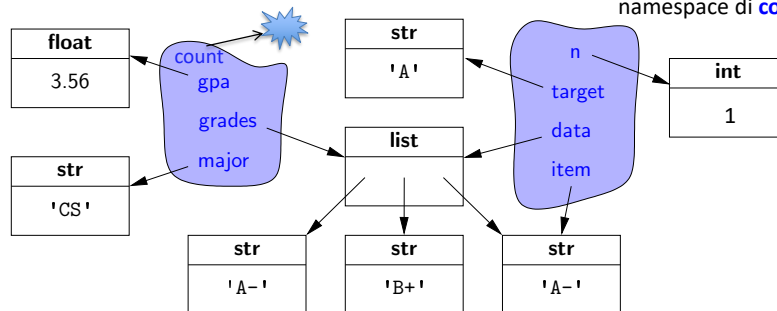
## Esempio

```
grades = ['A', 'B+', 'A-']
gpa = 3.56
major = 'CS'
count(grades, 'A')
```

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

namespace dove è chiamata **count**

namespace di **count**



Programmazione Avanzata a.a. 2019-20  
A. De Bonis

59

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

```
grades = ['A', 'B+', 'A-']
gpa = 3.56
major = 'CS'
```

```
count(grades, 'A')
```

```
print(dir())
```

## Esempio

```
[
    '__annotations__', '__builtins__',
    '__cached__', '__doc__', '__file__',
    '__loader__', '__name__', '__package__',
    '__spec__', 'count', 'gpa', 'grades', 'major'
]
```

Programmazione Avanzata a.a. 2019-20  
A. De Bonis

60

## 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__`

Programmazione Avanzata a.a. 2019-20  
A. De Bonis

61

## Moduli esistenti

- Esistono vari moduli già disponibili in Python
  - Alcuni utili moduli sono i seguenti

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.

Programmazione Avanzata a.a. 2019-20  
A. De Bonis

62

## 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.gcd(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 inserire un controllo nel modulo sul nome del modulo stesso. Se il nome del modulo è `__main__` allora il codice libero è eseguito; altrimenti il codice non viene eseguito.

## La variabile `__name__`

- Ogni volta che un modulo è importato Python crea una variabile per il modulo chiamata `__name__` e salva il nome del modulo in questa variabile.
- Il nome di un modulo è il nome del suo file `.py` senza l'estensione `.py`.
- Supponiamo di importare il modulo contenuto nel file `test.py`. Quando questo modulo è importato la variabile `__name__` ha valore `"test"`.
- Supponiamo che il modulo `test.py` contenga del codice libero. Se prima di questo codice inseriamo il controllo `if __name__ == '__main__':` allora il codice libero viene eseguito se e solo se `__name__` ha valore `__main__`. Di conseguenza, se importiamo il modulo `test` allora il suddetto codice libero non è eseguito.
- Ogni volta che un file `.py` è eseguito Python crea una variabile per il programma chiamata `__name__` e pone il suo valore uguale a `"__main__"`. Di conseguenza se eseguiamo `test.py` come se fosse un programma allora il valore della sua variabile `__name__` è `__main__` e il codice libero dopo `if` viene eseguito.

## Esempio

testNolfMain.py

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

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

test.py

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

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

esecuzione testNolfMain.py

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

Stesso comportamento se eseguiti entrambi come programmi

esecuzione test.py

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

Programmazione Avanzata a.a. 2019-20  
A. De Bonis

67

## Esempio

importUNO.py

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

importDUE.py

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

esecuzione importUNO.py

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

In questo caso l'if presente in test.py evita che vengano eseguite le linee di codice libero presenti in test.py

esecuzione importDUE.py

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

In questo caso vengono eseguite anche le linee di codice libero di testNoMain.py perché non sono precedute dall'if

Programmazione Avanzata a.a. 2019-20  
A. De Bonis

68

## package

- Modo per strutturare codice Python in moduli, cartelle e sotto-cartelle
- Il package è una collezione di moduli
  - Il package è una cartella in cui, oltre ai moduli o subpackage, è presente il file `__init__.py` che contiene istruzioni di inizializzazione del package (può essere anche vuoto)
  - `__init__.py` serve ad indicare a Python di trattare la cartella come un package

```

sound/
  __init__.py
  formats/
    __init__.py
    wavread.py
    wavwrite.py
    aiffread.py
    aiffwrite.py
    auread.py
    auwrite.py
    ...
  effects/
    __init__.py
    echo.py
    surround.py
    reverse.py
    ...
  filters/
    __init__.py
    equalizer.py
    vocoder.py
    karaoke.py
    ...

```

Top-level package  
Initialize the sound package  
Subpackage for file format conversions

In uno script presente nella cartella che contiene **sound**

```

import sound.effects.echo
sound.effects.echo.echofilter(input, output, delay=0.7)

```

Subpackage for sound effects

```

from sound.effects import echo
echo.echofilter(input, output, delay=0.7)

```

Subpackage for filters

```

from sound.effects.echo import echofilter
echofilter(input, output, delay=0.7)

```

```

sound/
  __init__.py
  formats/
    __init__.py
    wavread.py
    wavwrite.py
    aiffread.py
    aiffwrite.py
    auread.py
    auwrite.py
    ...
  effects/
    __init__.py
    echo.py
    surround.py
    reverse.py
    ...
  filters/
    __init__.py
    equalizer.py
    vocoder.py
    karaoke.py
    ...

```

Top-level package  
Initialize the sound package

Subpackage for file format conversions

Per importare moduli in surround.py  
si usa un import relativo

```

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

```

Programmazione Avanzata a.a. 2019-20  
A. De Bonis

71

## Importare moduli tra **package**

- Lo script che importa il modulo deve conoscere la posizione del modulo da importare
  - Non è necessario quando
    - il modulo è un modulo di Python
    - il modulo è stato installato
  - La variabile `sys.path` è una lista di stringhe che determina il percorso di ricerca dell'interprete Python per i moduli
  - Occorre aggiungere a `sys.path` il percorso assoluto che contiene il modulo da importare

## Importare moduli tra **package**

- Quando il modulo `miomodulo` è importato l'interprete prima cerca un modulo built-in con quel nome. Se non lo trova, cerca un file `miomodulo.py` nella lista di directory date dalla variabile `sys.path`
- `sys.path` è inizializzata dalle seguenti locazioni:
  - La directory contenente lo script input
  - [PYTHONPATH](#) (una lista di nomi di directory con la stessa sintassi della variabile shell `PATH`).
  - Default dipendente dall'installazione

