

PER ALTRI APPUNTI CONSULTARE IL SITO:

https://luigi-v.github.io/Appunti_Universita/

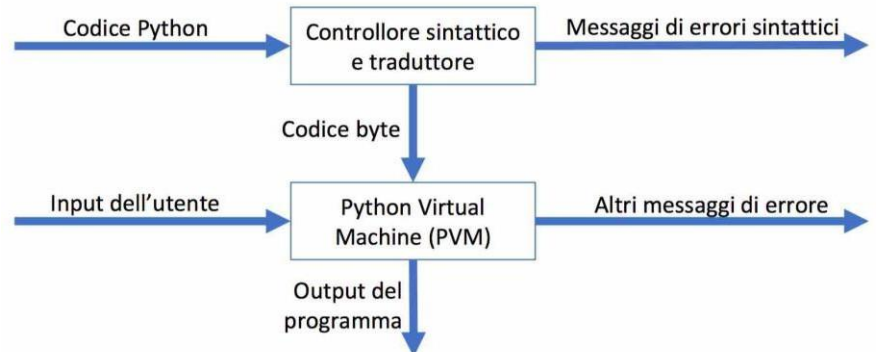
Il linguaggio Python - <https://docs.python.org/3/tutorial/> o <http://docs.python.it/>

Python è un linguaggio interpretato. I comandi sono eseguiti da un interprete, che riceve un comando, lo valuta e restituisce il risultato.

Un programmatore memorizza una serie di comandi in un file di testo a cui faremo riferimento con il termine codice sorgente o script (**modulo**). Convenzionalmente il codice sorgente è memorizzato in un file con estensione .py.

L'interprete svolge il ruolo di controllore sintattico e di traduttore, il **bytecode** è la traduzione del codice Python in un linguaggio di basso livello.

È la **Python Virtual Machine** ad eseguire il **bytecode**.



Come scrivere i codici:

- Commento introduttivo
- Import dei moduli richiesta dal programma: Subito dopo il commento introduttivo
- Inizializzazione di eventuali variabili del modulo
- Definizione delle funzioni: Tra cui la funzione main (*non è necessaria*)
- Docstring per ogni funzione definita nel modulo
- Uso di nomi significativi

esempio di modulo: file fact.py

```
def factorial(n):      # funzione che computa il fattoriale
    result=1.          # inizializza la variabile che contiene il risultato
    for k in range(1,n+1):
        result=result*k
    return result      # restituisce il risultato
```

```
print("fattoriale di 3:",factorial(3))
print("fattoriale di 1:",factorial(1))
print("fattoriale di 0:",factorial(0))
```

```
fattoriale di 3: 6.0
fattoriale di 1: 1.0
fattoriale di 0: 1.0
```

*La funzione **main** non è necessaria introdurla*

Convenzioni:

- Nomi di funzioni, metodi e di variabili iniziano sempre con la lettera minuscola
- Nomi di classi iniziano con la lettera maiuscola
- Nel caso di costanti scrivere il nome tutto in maiuscolo

*Si usa la notazione **CamelCase***

Commenti

Possono essere usati per spiegare il codice Python, per rendere il codice più leggibile e per impedire l'esecuzione durante il test del codice.

```
"""
This is a comment
written in
more than just one line
"""
```

-- oppure -- #This is a comment

Variabili

Identificatori:

- Sono case sensitive
- Possono essere composti da lettere, numeri e underscore (_)
- Un identificatore non può iniziare con un numero e non può essere una delle seguenti parole riservate

Gli identificatori possono contenere caratteri **unicode** che somigliano a lettere: résumé, π

Tipi di variabili:

Il tipo di una variabile (intero, carattere, virgola mobile, ..., anche numeri complessi: 3+5j) è basato sull'utilizzo della variabile e non deve essere specificato prima dell'utilizzo.

La variabile può essere riutilizzata nel programma e il suo tipo può cambiare in base alla necessità corrente.

<div style="border: 1px solid blue; padding: 5px; width: fit-content;">script</div> <pre>a = 3 print(a, type(a)) a = "casa" print(a, type(a)) a = 4.5 print(a, type(a))</pre>	<div style="color: red;">output</div>	<div style="border: 1px solid red; padding: 5px; width: fit-content;">output</div> <pre>3 <class 'int'> casa <class 'str'> 4.5 <class 'float'></pre>
---	---------------------------------------	--

Si possono assegnare valori a variabili su una sola riga: `x, y, z = "Orange", "Banana", "Cherry"`

Si può assegnare lo stesso valore a più variabili: `x = y = z = "Orange"`

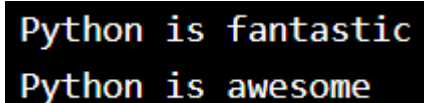
Variabili globali:

Le variabili create al di fuori di una funzione sono note come variabili globali, e possono essere utilizzate da tutti, sia all'interno che all'esterno delle funzioni.

Se si crea una variabile con lo stesso nome all'interno di una funzione, questa variabile sarà locale e può essere utilizzata solo all'interno della funzione. La variabile globale con lo stesso nome rimarrà com'era, globale e con il valore originale.

```
x = "awesome"
def myfunc():
    x = "fantastic"
    print("Python is " + x)

myfunc()
print("Python is " + x)
```



Parola chiave "global":

Per creare una variabile globale all'interno di una funzione, è possibile utilizzare la parola chiave **global**.

```
def myfunc():
    global x
    x = "fantastic"
```

Oggetti

- La classe per i numeri interi **int**
- La classe per i numeri in virgola mobile **float**
- La classe per le stringhe **str**

t = 3.8 crea una nuova istanza della classe float

In alternativa possiamo invocare il costruttore float(): t=float(3.8)

Casting:

Il cast in python viene quindi fatto usando le funzioni di costruzione:

- **int ()** (restituisce 0 di default)- costruisce un numero intero da un valore intero letterale, un valore letterale float (arrotondando per difetto al numero intero precedente) o letterale stringa (purché la stringa rappresenti un numero intero)
- **float ()** (restituisce **0.0** di default)- costruisce un numero float da un valore intero letterale, un valore letterale float o un valore letterale stringa (purché la stringa rappresenti un valore float o un numero intero)
- **str ()** - costruisce una stringa da una varietà di tipi di dati, inclusi stringhe, valori letterali interi e valori float

```
x = int(1)    # x will be 1
y = int(2.8)  # y will be 2
z = int("3")  # z will be 3
```

```
x = float(1)    # x will be 1.0
y = float(2.8)  # y will be 2.8
z = float("3")  # z will be 3.0
w = float("4.2") # w will be 4.2
```

```
x = str("s1")  # x will be 's1'
y = str(2)     # y will be '2'
z = str(3.0)   # z will be '3.0'
```

Stringhe:

Puoi assegnare una stringa multilinea a una variabile usando tre virgolette:

```
a = """Lorem ipsum dolor sit amet,
ut labore et dolore magna aliqua."""
```

```
a = '''Lorem ipsum dolor sit amet,
ut labore et dolore magna aliqua.'''
```

Le stringhe sono array, le parentesi quadre possono essere utilizzate per accedere agli elementi della stringa.

```
a = "Hello, World!"
print(a[1])
```

output→ e

È possibile restituire un intervallo di caratteri utilizzando la sintassi della sezione.

```
b = "Hello, World!"
print(b[2:5])
```

output→ llo

La funzione **len()** restituisce la lunghezza di una stringa:

```
a = "Hello, World!"
print(len(a))
```

output→ 13

Metodi stringa:

```
a = " Hello, World! "  
print(a)  
print(a.strip()) # returns "Hello, World!"  
print(a.lower())  
print(a.upper())  
print(a.replace("H", "J"))  
print(a.split(",")) # returns ['Hello', ' World!']
```

```
Hello, World!  
Hello, World!  
hello, world!  
HELLO, WORLD!  
Jello, World!  
[' Hello', ' World! ']
```

*Il metodo **split()** divide la stringa in sottostringhe se trova istanze del separatore

Per verificare se una frase o carattere è presente in una stringa, possiamo usare le parole chiave **in** o **not in**:

```
txt = "The rain in Spain stays mainly in the plain"  
x = "ain" in txt  
print(x)
```

output → True

Il metodo **format()** accetta gli argomenti passati, li formatta e li inserisce nella stringa in cui sono i segnaposto {}:

```
age = 36  
txt = "My name is John, and I am {}"  
print(txt.format(age))
```

Puoi utilizzare i numeri di indice {0} per assicurarti che gli argomenti siano inseriti nei segnaposto corretti:

```
quantity = 3  
itemno = 567  
price = 49.95  
myorder = "I want to pay {2} dollars for {0} pieces of item {1}."  
print(myorder.format(quantity, itemno, price))
```

Booleani:

La funzione **bool()** (restituisce **False** id default) ti consente di valutare valore e darti **True** o **False** in cambio:

```
bool("abc")  
bool(123)  
bool(["apple", "cherry", "banana"])
```

-
Tutti True

```
bool(False)  
bool(None)  
bool(0)  
bool("")  
bool()  
bool([])  
bool({})
```

-
Tutti False
-

Oggetti mutable/immutable:

Oggetti il cui valore può cambiare sono chiamati **mutable**.

Una classe è **immutable** se un oggetto della classe una volta inizializzato non può essere modificato in seguito

Un oggetto contenitore **immutable** che contiene un riferimento ad un oggetto **mutable**, può cambiare quando l'oggetto contenuto cambia, il contenitore è comunque considerato **immutable** perché la collezione di oggetti che contiene non può cambiare.

Class	Description	Immutable?
bool	Boolean value	✓
int	integer (arbitrary magnitude)	✓
float	floating-point number	✓
list	mutable sequence of objects	
tuple	immutable sequence of objects	✓
str	character string	✓
set	unordered set of distinct objects	
frozenset	immutable form of set class	✓
dict	associative mapping (aka dictionary)	

Operatori

Operatori aritmetici:

+	Addition	$x + y$
-	Subtraction	$x - y$
*	Multiplication	$x * y$
/	Division	x / y
%	Modulus	$x \% y$
**	Exponentiation	$x ** y$
//	Floor division	$x // y$

Operatori di confronto:

==	Equal	$x == y$
!=	Not equal	$x != y$
>	Greater than	$x > y$
<	Less than	$x < y$
>=	Greater than or equal to	$x >= y$
<=	Less than or equal to	$x <= y$

Operatori logici:

and	Returns True if both statements are true	$x < 5$ and $x < 10$
or	Returns True if one of the statements is true	$x < 5$ or $x < 4$
not	Reverse the result, returns False if the result is true	$\text{not}(x < 5 \text{ and } x < 10)$

Operatori di identità:

is	Returns true if both variables are the same object	$x \text{ is } y$
is not	Returns true if both variables are not the same object	$x \text{ is not } y$

Operatori di assegnamento:

=	$x = 5$	$x = 5$
+=	$x += 3$	$x = x + 3$
-=	$x -= 3$	$x = x - 3$
*=	$x *= 3$	$x = x * 3$
/=	$x /= 3$	$x = x / 3$
%=	$x \% = 3$	$x = x \% 3$
//=	$x //= 3$	$x = x // 3$
**=	$x ** = 3$	$x = x ** 3$
&=	$x \& = 3$	$x = x \& 3$
=	$x = 3$	$x = x 3$
^=	$x \wedge = 3$	$x = x \wedge 3$
>>=	$x >> = 3$	$x = x >> 3$
<<=	$x << = 3$	$x = x << 3$

Operatori di associatività:

in	Returns True if a sequence with the specified value is present in the object	$x \text{ in } y$
not in	Returns True if a sequence with the specified value is not present in the object	$x \text{ not in } y$

L'espressione **a is b** risulta vera solo se a e b sono alias dello stesso oggetto.

L'espressione **a == b** risulta vera anche quando gli identificatori a e b si riferiscono ad oggetti che possono essere considerati equivalenti. Due oggetti dello stesso tipo che **contengono** gli stessi valori.

```
lst1 = ['a','b',['ab','ba']]
lst2 = lst1
if lst1 is lst2:
    print('Oggetti identici')
else:
    print('Oggetti distinti')

if lst1 == lst2:
    print('Oggetti equivalenti')
else:
    print('Oggetti non equivalenti')
```

Oggetti identici
Oggetti equivalenti

```
lst1 = ['a','b',['ab','ba']]
lst2 = lst1.copy()
if lst1 is lst2:
    print('Oggetti identici')
else:
    print('Oggetti distinti')

if lst1 == lst2:
    print('Oggetti equivalenti')
else:
    print('Oggetti non equivalenti')
```

Oggetti distinti
Oggetti equivalenti

Sequenze

Classe *List*:

È una raccolta **ordinata** e **modificabile**. **Consente membri duplicati**.

thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"] o list= list()

- Accedere agli elementi:
`print(thislist[1])`
`print(thislist[-1])`
`print(thislist[2:5])` restituisce il 3,4,5 elemento
`x = fruits.index("cherry")` restituisce la posizione dell'elemento, si può specificare anche `list.index(x, start, end)`
`x = fruits.count("cherry")` restituisce il numero di volte dell'elemento
- Modificare valori:
`thislist[1] = "blackcurrant"`
`fruits.reverse()` restituisce la lista invertita
- Scorrere una lista:
`for x in thislist:`
`print(x)`
- Controllare se è presente un oggetto:
`if "apple" in thislist:`
`print("Yes, 'apple' is in the fruits list")`
- Lunghezza lista:
`print(len(thislist))`
- Aggiungere un elemento:
`thislist.append("orange")`
`fruits.insert(1, "orange")`
- Rimuovere un elemento:
`thislist.remove("banana")`
`thislist.pop()` estrae l'ultimo elemento
`del thislist[0]`
`del thislist` elimina tutta la lista
`thislist.clear()` svuota tutta la lista
- Copiare una lista:
List1=List2, si copierà solo il riferimento
`mylist = thislist.copy()`
`mylist = list(thislist)`

```
>>> x=["anna","michele","carla","antonio","fabio"]
>>> x
['anna', 'michele', 'carla', 'antonio', 'fabio']
>>> x.sort()
>>> x
['anna', 'antonio', 'carla', 'fabio', 'michele']
>>> x.sort(reverse=True)
>>> x
['michele', 'fabio', 'carla', 'antonio', 'anna']
>>> x.sort(key=len)
>>> x
['anna', 'fabio', 'carla', 'michele', 'antonio']
```
- Estendere una lista:
`list1.extend(list2)`
`list3 = list1 + list2`
- Ordinamento list:
`list.sort(reverse=True|False, key=myFunc)`
key= Una funzione per specificare i criteri di ordinamento
reverse= reverse = True ordinerà l'elenco in ordine decrescente.

Forma Comprehension:

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

La parte **if** è *opzionale*

– In sua assenza, 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)
```

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]
```

[(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)]
```

Classe Tuple:

È una collezione **ordinata** e **immutabile**(*non si può aggiungere o modificare*). **Consente membri duplicati**.

```
thistuple= ("apple", "banana", "cherry") o thistuple= tuple(("apple", "banana", "cherry"))
```

- Per creare una **tupla** con un solo elemento, devi aggiungere una virgola dopo l'elemento:

```
thistuple = ("apple",)  
print(type(thistuple))
```
- Accedere agli elementi:

```
print(thistuple[1])  
print(thistuple[-1])  
print(thistuple[2:5])
```
- Non può essere modificata ma è possibile convertire la tupla in lista, modificare l'elemento e riconvertirla:

```
x = ("apple", "banana", "cherry")  
y = list(x)  
y[1] = "kiwi"  
x = tuple(y)
```
- Scorrere una tupla:

```
for x in thislist:  
    print(x)
```
- Controllare se è presente un oggetto:

```
if "apple" in thislist:  
    print("Yes, 'apple' is in the fruits list")
```
- Lunghezza lista:

```
print(len(thislist))
```
- Eliminare la tupla:


```
thistuple = ("apple", "banana", "cherry")
del thistuple
print(thistuple) #this will raise an error because the tuple no longer exists
```

- Unire due tuple:

```
tuple1 = ("a", "b", "c")
tuple2 = (1, 2, 3)
tuple3 = tuple1 + tuple2
```

- Restituisce il numero di volte in cui il valore 5 appare nella tupla:

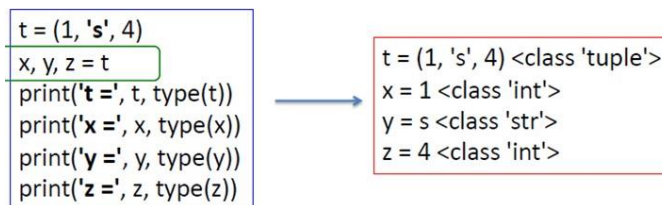
```
thistuple = (1, 3, 7, 8, 7, 5, 4, 6, 8, 5)
x = thistuple.count(5)
```

- Cerca la prima occorrenza del valore 8 e restituisce la sua posizione:

```
thistuple = (1, 3, 7, 8, 7, 5, 4, 6, 8, 5)
x = thistuple.index(8)
```

- **tuple packing/unpacking:**

Il **packing** è la creazione di una tupla. **L'unpacking** è la creazione di variabili a partire da una tupla



Classe Set:

È una raccolta **non ordinata** e **non indicizzata**. **Nessun membro duplicato**.

thisset = {"apple", "banana", "cherry"} o thisset = set(("apple", "banana", "cherry"))

- Aggiungere un elemento:

```
fruits.add("orange")
```

- Scorrere il set:

```
for x in thislist:
    print(x)
```

- Controllare se è presente un elemento:

```
print("banana" in thisset)
```

- Una volta creato un set, non è possibile modificarne gli elementi, ma è possibile aggiungere nuovi elementi.

- Aggiungere elementi:

```
thisset.add("orange")
thisset.update(["orange", "mango", "grapes"])
```

aggiunta di più elementi

- Lunghezza set:

```
print(len(thisset))
```

- Rimuovere un elemento:

```
thisset.remove("banana")
thisset.discard("banana")
thisset.pop()
```

se l'elemento da rimuovere non esiste, genererà un errore.
questo metodo non genera errore se non lo trova.
rimuove e lo restituisce.

- Eliminare set:
`thisset.clear()` svuota set
`del thisset` elimina set
- Unire più set:
`set3 = set1.union(set2)`
`set1.update(set2)`
- Copiare un set:
`x = fruits.copy()`

Classe *frozenset*:

È una classe *immutable* del tipo set, si può avere un set di *frozenset*.

Stessi metodi ed operatori di *set*, si possono eseguire facilmente test di (non) appartenenza, operazioni di unione, intersezione, differenza, ...

Classe *Dictionary*:

È una raccolta *non ordinata, modificabile e indicizzata. Nessun membro duplicato.*

```
thisdict = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}
```

- Accedere agli elementi:
`x = thisdict["model"]` ottieni il valore della chiave
`x = thisdict.get("model")` //
`for x in thisdict:` stampa tutti i nomi
 `print(x)`
`for x in thisdict:` stampa tutti i valori
 `print(thisdict[x])`
`for x in thisdict.values():` //
 `print(x)`
`if "model" in thisdict:`
 `print("Yes, 'model' is one of the keys in the thisdict dictionary")`
- Modifica valori:
`thisdict["year"] = 2018`
- Lunghezza del dizionario:
`print(len(thisdict))`
- Aggiungere un elemento:
`thisdict["color"] = "red"`
`car.update({"color": "White"})`
- Rimuovere un elemento:
`thisdict.pop("model")`
`thisdict.popitem()` rimuove l'ultimo elemento
`del thisdict["model"]`

- Eliminare dizionario:
`thisdict.clear()`
- Copiare un dizionario:
`mydict = thisdict.copy()`

```
tel = {'jack': 4098, 'sape': 4139}
tel['guido'] = 4127
print('tel =', tel)
tel['irv'] = 4127
print('tel =', tel)
del tel['sape']
print('tel =', tel)
```

```
tel = {'jack': 4098, 'guido': 4127, 'sape': 4139}
tel = {'jack': 4098, 'irv': 4127, 'guido': 4127, 'sape': 4139}
tel = {'jack': 4098, 'irv': 4127, 'guido': 4127}
```

```
chiavi = tel.keys()
print('chiavi =', chiavi)
valori = tel.values()
print('valori =', valori)
for i in chiavi:
    print(i)
```

```
chiavi = dict_keys(['guido', 'irv', 'jack'])
valori = dict_values([4127, 4127, 4098])
guido
irv
jack
```

```
for i in tel.keys():
    print(i)
```

```
elementi = tel.items()
for k,v in elementi:
    print(k,v)
```

```
irv 4127
guido 4127
jack 4098
```

```
print('tel =', tel)
tel2 = {'guido': 1111, 'john': 666}
print('tel2 =', tel2)
tel.update(tel2)
print('tel =', tel)
tel.update([('mary', 1256)])
print('tel =', tel)
```

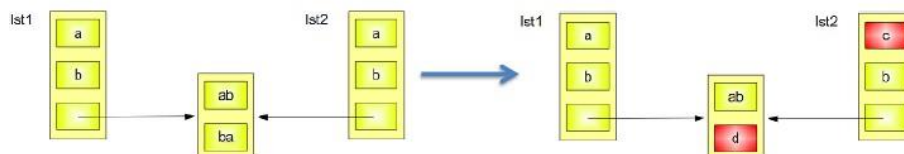
```
tel = {'irv': 4127, 'guido': 4127, 'jack': 4098}
tel2 = {'guido': 1111, 'john': 666}
tel = {'guido': 1111, 'john': 666, 'irv': 4127, 'jack': 4098}
tel = {'guido': 1111, 'mary': 1256, 'john': 666, 'irv': 4127, 'jack': 4098}
```

shallow vs deep copy

shallow: Costruisce un nuovo oggetto composto e inserisce in esso i riferimenti agli oggetti presenti nell'originale.

```
lst1 = ['a','b',['ab','ba']]
lst2 = lst1.copy() #metodo della classe list
print('lista1 =', lst1)
print('lista2 =', lst2)
lst2[0] = 'c'
lst2[2][1] = 'd'
print('lista1 =', lst1)
print('lista2 =', lst2)
```

```
lista1 = ['a', 'b', ['ab', 'ba']]
lista2 = ['a', 'b', ['ab', 'ba']]
lista1 = ['a', 'b', ['ab', 'd']]
lista2 = ['c', 'b', ['ab', 'd']]
```



deep: Costruisce un nuovo oggetto composto e ricorsivamente inserisce in esso le copie degli oggetti presenti nell'originale. **Se un oggetto a contenente un riferimento a se stesso allora una copia deep di a causa un loop.**

```

from copy import deepcopy
lst1 = ['a','b',['ab','ba']]
lst2 = deepcopy(lst1)
print('lista1 =', lst1)
print('lista2 =', lst2)
lst2[0] = 'c'
lst2[2][1] = 'd'
print('lista1 =', lst1)
print('lista2 =', lst2)

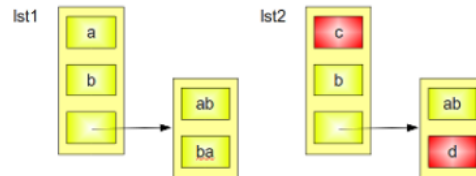
```



```

lista1 = ['a', 'b', ['ab', 'ba']]
lista2 = ['a', 'b', ['ab', 'ba']]
lista1 = ['a', 'b', ['ab', 'ba']]
lista2 = ['c', 'b', ['ab', 'd']]

```



List Comprehension:

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 e Dizionari.

[expression for value in iterable if condition]

expression e **condition** possono dipendere da **value**

La parte **if** è opzionale

- In sua assenza, si considerano tutti i value in iterable
- Se condition è vera, il risultato di expression è aggiunto alla lista

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]

[(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)]

list comprehension [k*k for k in range(1, n+1)]

set comprehension { k*k for k in range(1, n+1) }

dictionary comprehension { k : k*k for k in range(1, n+1) }

Operatori per sequenze **list**, **tuple** e **str**:

s[j]	element at index <i>j</i>
s[start:stop]	slice including indices [start,stop)
s[start:stop:step]	slice including indices start, start + step, start + 2*step, ..., up to but not equalling or stop
s + t	concatenation of sequences
k * s	shorthand for s + s + s + ... (k times)
val in s	containment check
val not in s	non-containment check

```

t = [2] * 7
print(t)

```



```
[2, 2, 2, 2, 2, 2, 2]
```



```

t = 7 * [2]
print(t)

```

Confronto di sequenze:

Le sequenze possono essere confrontate in base all'ordine lessicografico

– Il confronto è fatto elemento per elemento

– Ad esempio, `[5, 6, 9] < [5, 7]` (True)

<code>s == t</code>	equivalent (element by element)
<code>s != t</code>	not equivalent
<code>s < t</code>	lexicographically less than
<code>s <= t</code>	lexicographically less than or equal to
<code>s > t</code>	lexicographically greater than
<code>s >= t</code>	lexicographically greater than or equal to

Operatori per insiemi:

Le classi **set** e **frozenset** supportano i seguenti operatori:

<code>key in s</code>	containment check
<code>key not in s</code>	non-containment check
<code>s1 == s2</code>	s1 is equivalent to s2
<code>s1 != s2</code>	s1 is not equivalent to s2
<code>s1 <= s2</code>	s1 is subset of s2
<code>s1 < s2</code>	s1 is proper subset of s2
<code>s1 >= s2</code>	s1 is superset of s2
<code>s1 > s2</code>	s1 is proper superset of s2
<code>s1 s2</code>	the union of s1 and s2
<code>s1 & s2</code>	the intersection of s1 and s2
<code>s1 - s2</code>	the set of elements in s1 but not s2
<code>s1 ^ s2</code>	the set of elements in precisely one of s1 or s2

Operatori per dizionari:

La classe **dict** supporta i seguenti operatori:

<code>d[key]</code>	value associated with given key
<code>d[key] = value</code>	set (or reset) the value associated with given key
<code>del d[key]</code>	remove key and its associated value from dictionary
<code>key in d</code>	containment check
<code>key not in d</code>	non-containment check
<code>d1 == d2</code>	d1 is equivalent to d2
<code>d1 != d2</code>	d1 is not equivalent to d2

Assegnamento esteso:

In C e Java `i += 3` è equivalente a `i = i + 3`. In Python per i tipi **immutable** si crea un nuovo oggetto a cui si assegna un nuovo valore e l'identificatore è riassegnato al nuovo oggetto.

Alcuni tipi di dato (e.g., list) ridefiniscono la semantica dell'operatore +=

```
alpha = [1, 2, 3]
beta = alpha
print('alpha =', alpha)
print('beta =', beta)
beta += [4, 5]
print('beta =', beta)
beta = beta + [6, 7]
print('beta =', beta)
print('alpha =', alpha)
```



```
alpha = [1, 2, 3]
beta = [1, 2, 3]
beta = [1, 2, 3, 4, 5]
beta = [1, 2, 3, 4, 5, 6, 7]
alpha = [1, 2, 3, 4, 5]
```

Chaining

Assegnamento: `x = y = z = 0`, in Python è permesso l'assegnamento concatenato.

Operatori di confronto: `1 < x + y <= 9`, Equivalente a `(1 < x+y) and (x + y <= 9)`, ma l'espressione `x+y` è calcolata una sola volta.

Condizioni

```
if b > a:
    print("b is greater than a")
else:
    print("a is greater than b")
```

La parola chiave **elif** è il modo per dire "se le condizioni precedenti non erano vere, allora prova questa condizione".

```
a = 33
b = 33
if b > a:
    print("b is greater than a")
elif a == b:
    print("a and b are equal")
```

If abbreviato:

```
if a > b: print("a is greater than b")
print("A") if a > b else print("B")
print("A") if a > b else print("=") if a == b else print("B")
```

Si possono usare gli **operatori logici** e viene utilizzata per combinare le istruzioni condizionali:

```
if a > b and c > a:
    print("Both conditions are True")

if a > b or a > c:
    print("At least one of the conditions is True")
```

if nidificato:

```
if x > 10:
    print("Above ten,")
    if x > 20:
        print("and also above 20!")
    else:
        print("but not above 20.")
```

Cicli

Con il ciclo **while** possiamo eseguire una serie di istruzioni purché una condizione sia vera.

```
i = 1
while i < 6:
    print(i)
    i += 1
```

Con il ciclo for possiamo eseguire un set di istruzioni, una volta per ogni elemento in un elenco, tupla, set ecc.

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    print(x)
```

Anche le stringhe sono oggetti iterabili, contengono una sequenza di caratteri:

```
for x in "banana":
    print(x)
```

La funzione range () restituisce una sequenza di numeri, a partire da 0 per impostazione predefinita, e aumenta di 1 (per impostazione predefinita) e termina con un numero specificato.

```
for x in range(6):
    print(x)
```

Il "loop interno" verrà eseguito una volta per ogni iterazione del "loop esterno":

```
for x in adj:
    for y in fruits:
        print(x, y)
```

Con l'istruzione **break** possiamo interrompere il ciclo anche se la condizione while è vera:

```
i = 1
while i < 6:
    print(i)
    if i == 3:
        break
    i += 1
```

Con l'istruzione **continue** possiamo interrompere l'iterazione corrente e continuare con la successiva:

```
i = 0
while i < 6:
    i += 1
    if i == 3:
        continue
    print(i)
```

Con l'istruzione **else** possiamo eseguire un blocco di codice una volta quando la condizione non è più vera:

```
i = 1
while i < 6:
    print(i)
    i += 1
else:
    print("i is no longer less than 6")
```

```
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')
```

Con n=2

4 è un multiplo di 2

non ci sono multipli di 3 nella lista

Funzione range()

La funzione **range(start, stop, step)** restituisce una sequenza di numeri, a partire da 0 per impostazione predefinita, e aumenta di 1 (per impostazione predefinita) e termina con un numero specificato.

- **Start**, Un numero intero che specifica da quale posizione iniziare.
- **Stop**, Un numero intero che specifica in quale posizione terminare.
- **Step**, Un numero intero che specifica l'incremento.

Utile quando vogliamo iterare in una sequenza di dati utilizzando un indice: **for i in range(n)**

Crea una sequenza di numeri da 0 a 5 e stampa ogni elemento nella sequenza:

```
x = range(6)
for n in x:
    print(n)

x = range(3, 6)
for n in x:
    print(n)

x = range(3, 20, 2)
for n in x:
    print(n)
```

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

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

Crea una lista infinita

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

Il ciclo while richiede che le variabili pertinenti siano pronte, in questo esempio dobbiamo definire una variabile di indicizzazione, *i*, che impostiamo su 1.

```
i = 1
while i < 6:
    print(i)
    i += 1
```

Funzioni

In Python una funzione è definita usando la parola chiave **def**:

```
def my_function():
    print("Hello from a function")
```

Parametri:

I parametri sono specificati dopo il nome della funzione, tra parentesi, separati con una virgola.

```
def my_function(fname):
    print(fname + " Refsnes")
```

Parametri predefiniti:

Se chiamiamo la funzione senza parametro, utilizza il valore predefinito:

```
def my_function(country = "Norway"):
    print("I am from " + country)
```

Parametri elenco:

Se si invia un Elenco come parametro, sarà comunque un Elenco quando raggiunge la funzione:

```
fruits = ["apple", "banana", "cherry"]

def my_function(food):
    for x in food:
        print(x)
```

Stringa di documentazione

La prima riga di codice nella definizione di una funzione dovrebbe essere una breve spiegazione di quello che fa la funzione, chiamata **docstring**.

```
def my_function():
    """Do nothing, but document it. ...
    """
    pass # Istruzione che non fa niente
```

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

```
Do nothing, but document it. ...

No, really, it doesn't do anything.
```


Numero variabile di argomenti alle funzioni

Si possono definire funzioni con un numero variabile di parametri, l'ultimo parametro è preceduto da “*”.

Nel corpo della funzione possiamo accedere al valore di questi parametri tramite la posizione.

```
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
```

Ogni tipo iterabile può essere spaccettato usando l'operatore * (**unpacking operator**).

```
>>> 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]
```

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
```

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

addendi=[56,2,4]

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

somma = 62

Attenzione:
addendi deve
contenere
esattamente 3
elementi

Parametri keyword

Sono argomenti di una funzione preceduti da un identificatore oppure passati come dizionario (**dict**) preceduto da ******. Il parametro è considerato un dizionario (**dict**).

L'operatore ****** è il mapping **unpacking** operator e può essere applicato ai tipi mapping (collezione di coppie **chiave-valore**) quali i dizionari.

Qui cmd è un dizionario

```
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: 9

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

Il prodotto degli argomenti è: 24

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

Operando non supportato

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

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

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

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

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

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

Il metodo join()

Il metodo **join()** prende tutti gli elementi in un iterabile e li unisce in una stringa.

È necessario specificare una stringa come separatore.

```
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

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
```

Annotazioni

Le annotazioni sono dei metadati associati alle funzioni definite dal programmatore, memorizzate come un dizionario nell'attributo `__annotation__` della funzione.

L'**annotazione di parametri** è definita da “ : ” dopo il nome del parametro seguito da un'espressione

Le **annotazioni di ritorno** sono definite da “ -> ” seguita da un'espressione tra la lista dei parametri e i due punti che indicano la fine dell'istruzione **def**.

```
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'>}
```

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'>}
```

Funzioni come parametro

È possibile passare l'identificatore di una funzione **a** come parametro di un'altra 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)
```

```
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

Una **funzione lambda** è una piccola *funzione anonima*, può accettare qualsiasi numero di argomenti, ma può avere solo un'espressione.

lambda arguments : expression

Funzioni anonime create usando la **keyword lambda**

- 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.

Una funzione lambda che aggiunge 10 al numero passato come argomento e stampa il risultato:

```
x = lambda a : a + 10
print(x(5))
```

Una funzione lambda che somma l'argomento a, b e c e stampa il risultato:

```
x = lambda a, b, c : a + b + c
print(x(5, 6, 2))
```

Il potere di lambda viene mostrato meglio quando li usi come una funzione anonima all'interno di un'altra funzione.

Supponi di avere una definizione di funzione che accetta un argomento e che tale argomento verrà moltiplicato per un numero sconosciuto:

```
def myfunc(n):
    return lambda a : a * n
```

Utilizzare quella definizione di funzione per creare una funzione che raddoppia sempre il numero inviato:

```
def myfunc(n):
    return lambda a : a * n
```

```
mydoubler = myfunc(2)
```

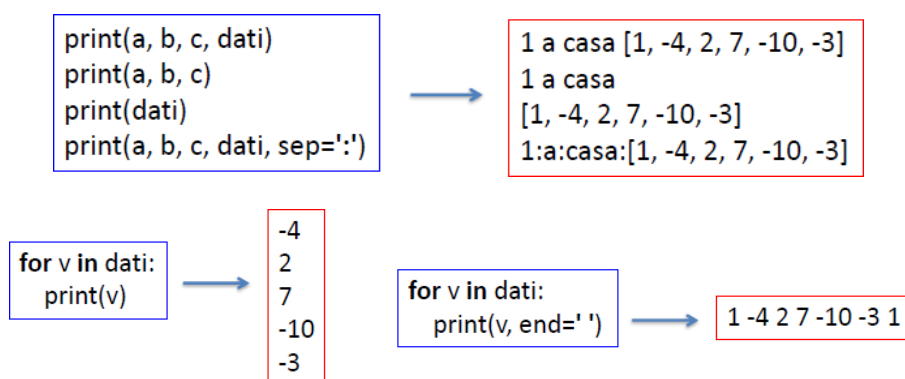
```
print(mydoubler(11))
```

Output: funzione print

Riceve un numero variabile di parametri da stampare e due parametri keyword:

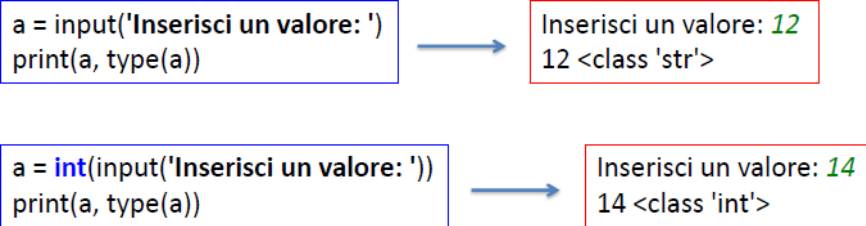
- **sep** - stringa di separazione dell'output (default spazio)
- **end** - stringa finale dell'output (default \n)

Gli argomenti ricevuti sono convertiti in stringhe, separati da **sep** e seguiti da **end**.



Input: funzione input

Riceve input da tastiera, e quello che viene letto è considerato stringa. L'input termina con la pressione di invio (\n) che non viene inserito nella stringa letta.



Gestione dei file

La funzione **open()** accetta due parametri: *nome file* e *modalità* .

Esistono quattro diversi metodi (modalità) per aprire un file:

- **"r"** - Leggi - Valore predefinito. Apre un file per la lettura, errore se il file non esiste
- **"a"** - Aggiungi: apre un file da aggiungere, crea il file se non esiste
- **"w"** - Scrivi - Apre un file per la scrittura, crea il file se non esiste
- **"x"** - Crea - Crea il file specificato, restituisce un errore se il file esiste

Inoltre, è possibile specificare se il file deve essere gestito in modalità binaria o di testo

- **"t"** - Testo: valore predefinito. Modalità testo
- **"b"** - Binario - Modalità binaria (ad es. Immagini)

Per aprire un file per la lettura è sufficiente specificare il nome del file:

`f = open("demofile.txt")` uguale a `f = open("demofile.txt", "rt")`

Perché "r" per read e "t" per text sono i valori predefiniti, non è necessario specificarli.

Lettura di file

La funzione **open()** restituisce un oggetto file, che ha un metodo **read()** per leggere il contenuto del file:

```
f = open("demofile.txt", "r")
print(f.read())
```

Per impostazione predefinita, il metodo **read()** restituisce l'intero testo, ma puoi anche specificare quanti caratteri vuoi restituire:

```
f = open("demofile.txt", "r")
print(f.read(5))
```

È possibile restituire una riga utilizzando il metodo **readline()**:

```
f = open("demofile.txt", "r")
print(f.readline())
```

Scrittura di file

Per scrivere su un file esistente, è necessario aggiungere un parametro alla **open()** funzione:

- **"a"** - Aggiungi: verrà aggiunto alla fine del file
- **"w"** - Scrivi: sovrascriverà qualsiasi contenuto esistente

```
f = open("demofile2.txt", "a")
f.write("Now the file has more content!")

f = open("demofile3.txt", "w")
f.write("Woops! I have deleted the content!")
```

Chiusura di un file

È buona norma chiudere sempre il file al termine.

```
f = open("demofile.txt", "r")
print(f.readline())
f.close()
```

è necessario chiudere sempre i file, in alcuni casi, a causa del buffering, le modifiche apportate a un file potrebbero non essere visualizzate fino alla chiusura del file.

Eliminare un file

Per eliminare un file, è necessario importare il modulo del sistema operativo ed eseguire **os.remove()**:

```
import os
os.remove("demofile.txt")
```

Per evitare di ricevere un errore, potresti voler verificare se il file esiste prima di provare a eliminarlo:

```
import os
if os.path.exists("demofile.txt"):
    os.remove("demofile.txt")
else:
    print("The file does not exist")
```

Moduli e pacchetti

Namespace:

Quando si utilizza un identificativo si attiva un processo chiamato **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). Python implementa il namespace tramite un dizionario che mappa ogni identificativo al suo valore.

Si può conoscere il contenuto del **namespace** dove sono invocate, tramite:

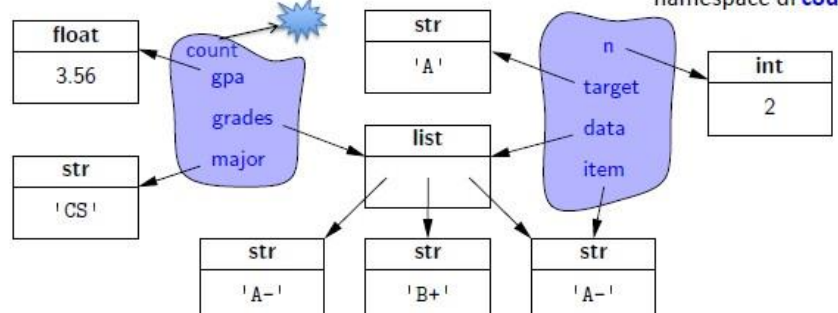
- **dir()**: elenca gli identificatori nel namespace
- **vars()**: visualizza tutto il dizionario

```
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**



Moduli:

Un modulo è come una *libreria* di codici, ovvero un file contenente una serie di funzioni che desideri includere nella tua applicazione, ma anche variabili di tutti i tipi (array, dizionari, oggetti ecc.).

Sintassi: *nome_modulo.nome_funzione*

Per creare un modulo basta salvare il codice desiderato in un file con l'estensione **.py**.

```
mymodule.py:      def greeting(name):
                    print("Hello, " + name)

                    person1 = {
                        "name": "John",
                        "age": 36,
                        "country": "Norway"
                    }
```

Importa il modulo denominato “*mymodule*” e chiama la funzione di saluto:

```
import mymodule

mymodule.greeting("Jonathan")
a = mymodule.person1["age"]
print(a)
```

Puoi nominare il file del modulo come preferisci, ma deve avere l'estensione del file .py.

Puoi creare un alias quando importi un modulo, usando la parola chiave **as**:

```
import mymodule as mx

a = mx.person1["age"]
print(a)
```

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.

C'è una funzione integrata per elencare tutti i nomi delle funzioni (o nomi delle variabili) in un modulo. La **dir()**:

```
import platform

x = dir(platform)
print(x)
```

Puoi scegliere di importare solo parti da un modulo, usando la parola chiave **from**:

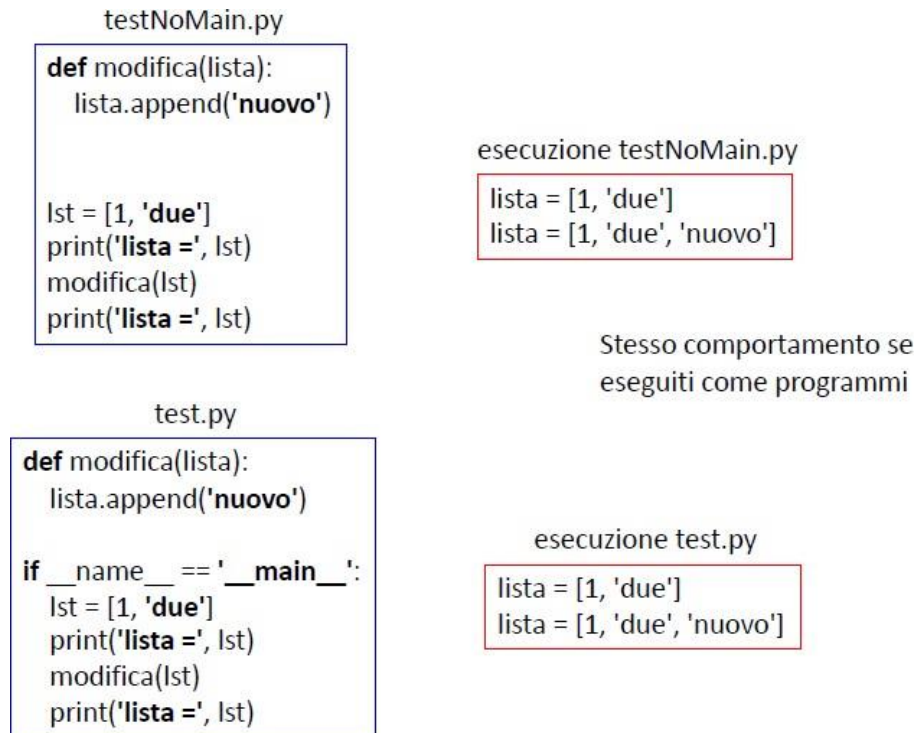
```
from mymodule import person1

print (person1["age"])
```

All'interno di un modulo/script si può accedere al nome del modulo/script tramite l'identificatore `_name_`.

Ogni volta che un modulo è caricato in uno script è eseguito, il quale può contenere **funzioni**, che vengono interpretate, e **codice libero**, che viene eseguito.

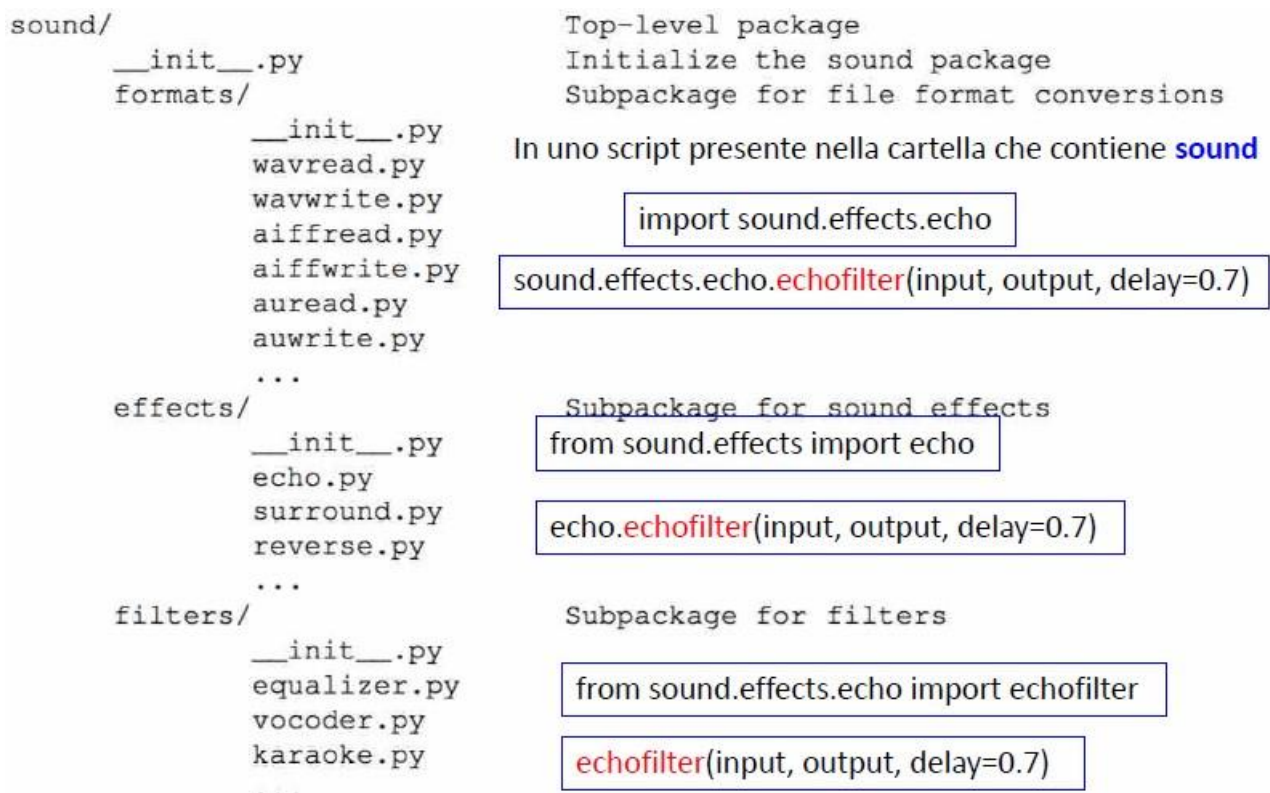
Lo script che importa altri moduli ed è eseguito per primo è chiamato dall'interprete Python **_main_**. Per evitare che del codice *libero* in un modulo sia eseguito quando il modulo è importato dobbiamo inserire un controllo nel modulo sul nome del modulo stesso. Se il nome del modulo è `__main__` allora il codice libero è eseguito.



Package:

Modo per strutturare codice Python in moduli, cartelle e sotto-cartelle, ovvero una **collezione di moduli**.

Il package è una cartella in cui, oltre ai moduli o sub-package, è presente il file `__init__.py` che contiene istruzioni di inizializzazione del package. `__init__.py` serve ad indicare a Python di trattare la cartella come un package.



Eccezioni

Python offre due funzionalità molto importanti per gestire eventuali errori imprevisti, aggiungendo funzionalità aggiuntive:

▪ Gestori di Eccezioni

Exception	Classe di base per tutte le eccezioni
StopIteration	Generato quando il metodo next() di un iteratore non punta a nessun oggetto
ArithmeticError	Classe di base per tutti gli errori che si verificano per il calcolo numerico
OverflowError	Generato quando un calcolo supera il limite massimo per un tipo numerico
FloatingPointError	Sollevato quando un calcolo in virgola mobile non riesce
AssertionError	Generato in caso di fallimento dell'affermazione Assert
EOFError	Generato quando non è presente alcun input dalla funzione input () e il file finisce
ImportError	Generato quando un'istruzione di importazione ha esito negativo
IndexError	Generato quando un indice non viene trovato in una sequenza
SystemError	Generato quando l'interprete rileva un problema interno
TypeError	Generato quando si tenta di eseguire un'operazione non valida per il tipo di dati
RuntimeError	Generato quando un errore generato non rientra in nessuna categoria

- **Asserzioni**, è un controllo di integrità che è possibile attivare o disattivare al termine dei test del programma. Viene verificata un'espressione e se il risultato risulta falso, viene sollevata un'eccezione.

Le asserzioni vengono eseguite dall'istruzione **assert**. La sintassi è: `assert Expression[, Arguments]`

Le eccezioni **AssertionError** possono essere rilevate e gestite come qualsiasi altra eccezione utilizzando l'istruzione **try-except**, ma se non gestite, termineranno il programma e produrranno un **traceback**.

Gestire un'eccezione:

Il blocco **try** consente di verificare la presenza di errori in un blocco di codice.

Il blocco **except** consente di gestire l'errore.

Il blocco **finally** consente di eseguire il codice, indipendentemente dal risultato dei blocchi **try** e **except**.

```
try:
    print(x)
except:
    print("An exception occurred")
```

È possibile definire tutti i blocchi di eccezioni desiderati, ad esempio se si desidera eseguire un blocco di codice speciale per un tipo speciale di errore:

```
try:
    print(x)
except NameError:
    print("Variable x is not defined")
except:
    print("Something else went wrong")
```

È possibile utilizzare la parola chiave **else** per definire un blocco di codice da eseguire se non sono stati generati errori:

```
try:
    print("Hello")
except:
    print("Something went wrong")
else:
    print("Nothing went wrong")
```

Il blocco **finally** verrà eseguito indipendentemente dal fatto che il blocco try generi un errore o meno.

```
try:
    f = open("demofile.txt")
    f.write("Lorum Ipsum")
except:
    print("Something went wrong when writing to the file")
finally:
    f.close()
```

Classi e oggetti

Quasi tutto in Python è un oggetto, con le sue proprietà e metodi. Una **classe** è come un *costruttore di oggetti* o un "*progetto*" per la creazione di oggetti. Tutti i membri di una classe (dati e metodi) sono **pubblici**.

Crea una **classe** denominata MyClass, con una **proprietà** denominata x:

```
class MyClass:
    x = 5
```

Crea un **oggetto** chiamato p1 e stampa il valore di x:

```
p1 = MyClass()
print(p1.x)
```

Attributi:

- Le **variabili di classe** sono di solito aggiunte alla classe mediante assegnamenti **all'esterno** delle funzioni
- Le **variabili di istanza** sono aggiunte all'istanza mediante assegnamenti effettuati **all'interno** di funzioni che hanno **self** tra gli argomenti.

```
class myClass:
    a=3
    def method(self):
        self.a=4

x=myClass()
print(x.a)
x.method()
print(x.a)
y=myClass()
print(y.a)
print(myClass.a)
```

3
4
3
3

La funzione __init__():

Tutte le classi hanno una funzione chiamata **__init__()**, che viene sempre eseguita all'avvio della classe.

Crea una **classe** di nome Persona, usa la funzione **__init__()** per assegnare **valori** per nome ed età:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

```
p1 = Person("John", 36)
```

Metodi oggetto:

I metodi negli oggetti sono funzioni che appartengono all'oggetto.

Inserisci una **funzione** che stampa un saluto ed esegui sull'oggetto p1:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def myfunc(self):
        print("Hello my name is " + self.name)

p1 = Person("John", 36)
p1.myfunc()
```

I metodi di una classe possono essere definiti fuori la classe stessa

```
def f1(self, x, y):
    return min(x, x+y)

class C:
    f = f1

    def g(self):
        return 'Ciao Mondo!'

c = C()
print(c.f(2,3))
```

Il parametro **self** è un riferimento all'istanza corrente della classe e viene utilizzato per accedere alle variabili che appartengono alla classe.

L'auto-parametro:

Il parametro **self** è un riferimento all'istanza corrente della classe e viene utilizzato per accedere alle variabili che appartengono alla classe, puoi chiamarlo come preferisci, ma deve essere il **primo parametro di qualsiasi funzione della classe**:

```
class Person:
    def __init__(mysillyobject, name, age):
        mysillyobject.name = name
        mysillyobject.age = age
    def myfunc(abc):
        print("Hello my name is " + abc.name)

p1 = Person("John", 36)
p1.myfunc()
```

Modifica proprietà oggetto:

È possibile modificare le proprietà di oggetti come questo:

```
p1.age = 40
```

Elimina proprietà oggetto e l'oggetto stesso:

È possibile eliminare le proprietà sugli oggetti utilizzando la parola chiave **del**:

```
del p1.age
```

Puoi eliminare gli oggetti usando la parola chiave **del**:

```
del p1
```

Ereditarietà

L'**ereditarietà** ci consente di definire una classe che eredita tutti i metodi e le proprietà da un'altra classe.

La **classe genitore** è la classe da cui viene ereditata, chiamata anche classe base.

La **classe figlio** è la classe che eredita da un'altra classe, chiamata anche classe derivata.

Crea una classe genitore:

```
class Person:
    def __init__(self, fname, lname):
        self.firstname = fname
        self.lastname = lname

    def printname(self):
        print(self.firstname, self.lastname)
```

Crea una classe figlio:

Per creare una classe che eredita la funzionalità da un'altra classe, inviare la classe genitore come parametro durante la creazione della classe figlio.

Creare una **classe** denominata Student, che **erediterà** le **proprietà** e i **metodi** dalla **classe** Person:

```
class Student(Person):
    pass                                #Implementazione vuota
```

Ora la **classe** Student ha le stesse **proprietà** e **metodi** della classe Person.

```
x = Student("Mike", "Olsen")
x.printname()
```

funzione `__init__()` nel figlio:

Quando si aggiunge la funzione `_init_()`, la classe figlio non erediterà più la `_init_()` del genitore ma la sostituisce.

```
class Student(Person):
    def __init__(self, fname, lname):
        #add properties etc.
```

Per mantenere l'eredità della funzione `_init_()` genitore , aggiungi una chiamata alla `_init_()` genitore :

```
class Student(Person):
    def __init__(self, fname, lname):
        Person.__init__(self, fname, lname)
```

Utilizzare la funzione `super ()`:

Python ha anche una funzione **`super()`** che farà ereditare alla classe figlio tutti i metodi e le proprietà dal suo genitore:

```
class Student(Person):
    def __init__(self, fname, lname):
        super().__init__(fname, lname)
```

Aggiungi proprietà:

```
class Student(Person):
    def __init__(self, fname, lname, year):
        super().__init__(fname, lname)
        self.graduationyear = year
```

Aggiungi metodi:

```
class Student(Person):
    def __init__(self, fname, lname, year):
        super().__init__(fname, lname)
        self.graduationyear = year
```

```
def welcome(self):
    print("Welcome", self.firstname, self.lastname, "to the class of", self.graduationyear)
```

Se si aggiunge un metodo nella classe figlio con lo stesso nome di una funzione nella classe genitore, l'eredità del **metodo** genitore **verrà sovrascritta**.

Ereditarietà multipla

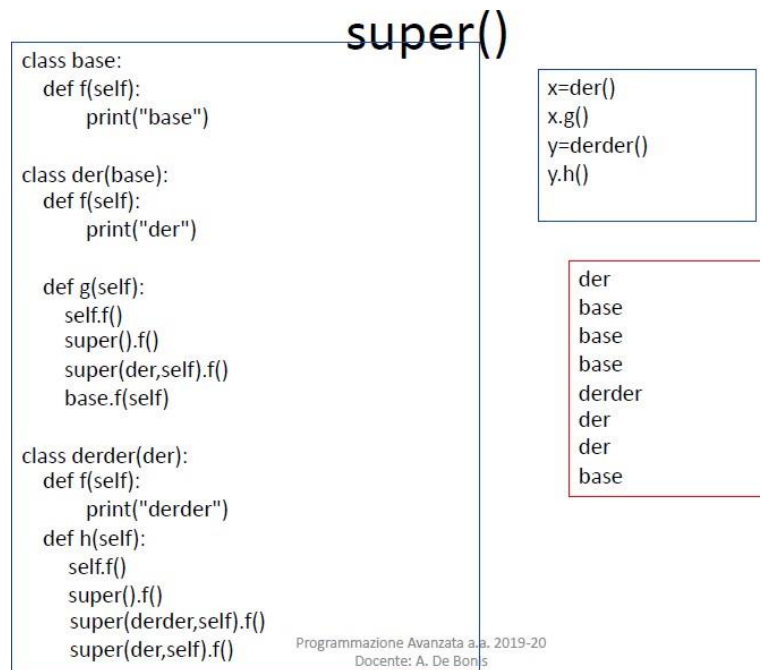
Python supporta ***l'ereditarietà multipla***.

```
class DerivedClassName(Base1, Base2, Base3):
```

Se un attributo non è trovato in `DerivedClassName` lo si cerca in `Base1`, dopo (ricorsivamente) nelle classi base di `Base1` e, se non è trovato si procede con `Base2` e così via...

Attributo `_mro_`:

L'attributo `_mro_` contiene l'elenco delle classi in cui si cerca il metodo che è stato invocato su un'istanza della classe

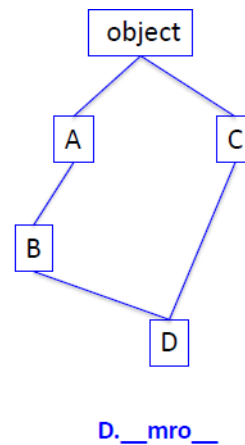


```
class A():
    pass
```

```
class B(A):
    pass
```

```
class C():
    pass
```

```
class D(B,C):
    pass
```



```
(<class '__main__.D'>, <class '__main__.B'>, <class '__main__.A'>, <class '__main__.C'>, <class 'object'>)
```

Attributo `__bases__`:

Contiene la tupla delle classi base di una classe

- Accessibile in lettura/scrittura
- Modificando `__bases__` l'attributo `__mro__` è *ricomputato*

Per modificare `__bases__` si usa la funzione **`setattr`**

- **`setattr`**(Derivata, 'bases', (Base2, Base1))

Funzione `isinstance(ist, classe)`:

Se il parametro type è una tupla, questa funzione tornerà True se l'oggetto è uno dei tipi nella tupla.

```
x = isinstance(5, int)
```

True

Funzione `issubclass(x,y)`:

Restituisce True se l'oggetto specificato è una sottoclasse dell'oggetto specificato, altrimenti False.

```
x = issubclass(myObj, myAge)
```

True

```
class A():
    def __init__(self, a, val):
        self._a = a
        self._val = val

    def stampa(self):
        print('a =', self._a, 'val =', self._val)
```

```
class B():
    def __init__(self, b, val):
        self._b = b
        self._val = val

    def stampa(self):
        print('b =', self._b, 'val =', self._val)
```

```
class C():
    def __init__(self, c, val):
        self._c = c
        self._val = val

    def stampa(self):
        print('c =', self._c, 'val =', self._val)
```

```
class D(A,B,C):
    def __init__(self, a, b, c, val):
        A.__init__(self, a, val)
        B.__init__(self, b, 2*val)
        C.__init__(self, c, 3*val)
```

```
def stampa(self):
    C.stampa(self)
    B.stampa(self)
    A.stampa(self)
```

```
d = D(1,2,3, 123)
d.stampa()
```

```
c = 3 val = 369
b = 2 val = 369
a = 1 val = 369
```

Overloading di operatori

In una classe Python implementiamo l'**overloading** degli operatori fornendo i metodi con nomi speciali (`_X_`) corrispondenti all'operatore.

Non ci sono default per questi metodi. Se una classe non definisce questi metodi allora l'operazione corrispondente non è supportata, nel caso venga usata un'operazione non supportata viene lanciata un'eccezione.

Se A non implementa `__add__` Python controlla se B implementa `__radd__` e lo esegue.

```
a = int(3)
b = int(2)
print(a.__pow__(b))
print(a.__rpow__(b))
```

→ 9
8

Common Syntax	Special Method Form
<code>a + b</code>	<code>a.__add__(b);</code> alternatively <code>b.__radd__(a)</code>
<code>a - b</code>	<code>a.__sub__(b);</code> alternatively <code>b.__rsub__(a)</code>
<code>a * b</code>	<code>a.__mul__(b);</code> alternatively <code>b.__rmul__(a)</code>
<code>a / b</code>	<code>a.__truediv__(b);</code> alternatively <code>b.__rtruediv__(a)</code>
<code>a // b</code>	<code>a.__floordiv__(b);</code> alternatively <code>b.__rfloordiv__(a)</code>
<code>a % b</code>	<code>a.__mod__(b);</code> alternatively <code>b.__rmod__(a)</code>
<code>a ** b</code>	<code>a.__pow__(b);</code> alternatively <code>b.__rpow__(a)</code>
<code>a << b</code>	<code>a.__lshift__(b);</code> alternatively <code>b.__rlshift__(a)</code>
<code>a >> b</code>	<code>a.__rshift__(b);</code> alternatively <code>b.__rrshift__(a)</code>
<code>a & b</code>	<code>a.__and__(b);</code> alternatively <code>b.__rand__(a)</code>
<code>a ^ b</code>	<code>a.__xor__(b);</code> alternatively <code>b.__rxor__(a)</code>
<code>a b</code>	<code>a.__or__(b);</code> alternatively <code>b.__ror__(a)</code>
<code>a += b</code>	<code>a.__iadd__(b)</code>
<code>a -= b</code>	<code>a.__isub__(b)</code>
<code>a *= b</code>	<code>a.__imul__(b)</code>
...	...
<code>+a</code>	<code>a.__pos__()</code>
<code>-a</code>	<code>a.__neg__()</code>
<code>~a</code>	<code>a.__invert__()</code>

<code>abs(a)</code>	<code>a.__abs__()</code>
<code>a < b</code>	<code>a.__lt__(b)</code>
<code>a <= b</code>	<code>a.__le__(b)</code>
<code>a > b</code>	<code>a.__gt__(b)</code>
<code>a >= b</code>	<code>a.__ge__(b)</code>
<code>a == b</code>	<code>a.__eq__(b)</code>
<code>a != b</code>	<code>a.__ne__(b)</code>
<code>v in a</code>	<code>a.__contains__(v)</code>
<code>a[k]</code>	<code>a.__getitem__(k)</code>
<code>a[k] = v</code>	<code>a.__setitem__(k,v)</code>
<code>del a[k]</code>	<code>a.__delitem__(k)</code>
<code>a(arg1, arg2, ...)</code>	<code>a.__call__(arg1, arg2, ...)</code>

Non-Operatori

<code>len(a)</code>	<code>a.__len__()</code>
<code>hash(a)</code>	<code>a.__hash__()</code>
<code>iter(a)</code>	<code>a.__iter__()</code>
<code>next(a)</code>	<code>a.__next__()</code>
<code>bool(a)</code>	<code>a.__bool__()</code>
<code>float(a)</code>	<code>a.__float__()</code>
<code>int(a)</code>	<code>a.__int__()</code>
<code>repr(a)</code>	<code>a.__repr__()</code>
<code>reversed(a)</code>	<code>a.__reversed__()</code>
<code>str(a)</code>	<code>a.__str__()</code>

Metodo__call__

Se all'interno di una classe è definito il metodo `__call__` allora le istanze della classe diventano **callable**, che viene invocato ogni volta che usiamo il nome di un'istanza della classe come se fosse il nome di una funzione.

```
class C:
    def __call__(self, *pargs, **kargs):
        print('Chiamata:', pargs, kargs)

x=C()
x(1, 2, 3)
x(1, 2, 3, x=4, y=5)
```

```
Chiamata: (1, 2, 3) {}
Chiamata: (1, 2, 3) {'y': 5, 'x': 4}
```

Iteratori

Un iteratore è un oggetto che implementa il protocollo iteratore, che consiste nei metodi `iter()` e `__next__()`.

Elenchi, tuple, dizionari e set sono tutti oggetti iterabili. Sono contenitori iterabili da cui è possibile ottenere un iteratore.

Tutti questi oggetti hanno un metodo `iter()` che viene utilizzato per ottenere un iteratore:

```
mytuple = ("apple", "banana", "cherry")
myit = iter(mytuple)
print(next(myit))
print(next(myit))
print(next(myit))
```

```
apple
banana
cherry
```

Anche le stringhe sono oggetti iterabili e possono restituire un iteratore:

```
mystr = "banana"
myit = iter(mystr)

print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
```

```
b
a
n
a
n
a
```

Possiamo anche usare un ciclo `for` per iterare attraverso un oggetto iterabile:

```
mytuple = ("apple", "banana", "cherry")

for x in mytuple:
    print(x)
```

Il ciclo `for` in realtà crea un oggetto iteratore ed esegue il metodo `next()` per ciascun ciclo.

Crea un Iteratore:

Per creare un **oggetto/classe** come iteratore devi implementare i metodi:

- Il metodo `iter()`, è possibile eseguire operazioni (inizializzazione ecc.), Ma deve sempre restituire l'oggetto iteratore stesso.
- Il metodo `next()` consente di eseguire operazioni e deve restituire l'elemento successivo nella sequenza.


```
class MyNumbers:
    def __iter__(self):
        self.a = 1
        return self

    def __next__(self):
        x = self.a
        self.a += 1
        return x

myclass = MyNumbers()
myiter = iter(myclass)

print(next(myiter))
```

StopIteration:

Per impedire che l'iterazione continui per sempre, possiamo usare l'affermazione **StopIteration**.

Nel metodo **__next__()**, possiamo aggiungere una condizione di terminazione per generare un errore se l'iterazione viene eseguita un numero specificato di volte:

```
class MyNumbers:
    def __iter__(self):
        self.a = 1
        return self

    def __next__(self):
        if self.a <= 20:
            x = self.a
            self.a += 1
            return x
        else:
            raise StopIteration

myclass = MyNumbers()
myiter = iter(myclass)

for x in myiter:
    print(x)
```

Valore yield:

Quando si incontra un **yield** l'esecuzione del generatore è sospesa, viene restituito il valore indicato da **yield**, ogni volta che si chiama **next()**, il generatore riparte da dove l'esecuzione era stata sospesa.

Si tratta di una sorta di funzione che genera una sequenza di valori restituiti uno per volta tramite **yield**. Nel generatore non possono coesistere **yield** e **return**.

```
def new_range(n):
    k=0
    while k<n:
        yield k
        k += 1
```

```
for i in new_range(10):
    print(i, end=' ')
```



0 1 2 3 4 5 6 7 8 9

Scope di variabili di classi

Una variabile è disponibile solo all'interno dell'area in cui è stata creata.

Scope locale:

Una variabile creata all'interno di una funzione appartiene all'ambito locale di quella funzione e può essere utilizzata solo all'interno di tale funzione.

```
def myfunc():  
    x = 300  
    print(x)
```

È possibile accedere alla variabile locale da una funzione all'interno della funzione:

```
def myfunc():  
    x = 300  
    def myinnerfunc():  
        print(x)  
    myinnerfunc()
```

Scope globale:

Le variabili globali sono disponibili in qualsiasi ambito, globale e locale.

Una variabile creata al di fuori di una funzione è globale e può essere utilizzata da chiunque:

```
x = 300  
def myfunc():  
    print(x)  
  
myfunc()  
  
print(x)
```

Variabili di denominazione:

Se si opera con lo stesso nome di variabile all'interno e all'esterno di una funzione, Python le tratterà come due variabili separate, una disponibile nell'ambito globale (all'esterno della funzione) e una disponibile nell'ambito locale (all'interno della funzione):

```
x = 300  
def myfunc():  
    x = 200  
    print(x)  
  
myfunc()  
print(x)
```



200
300

Parola chiave globale:

Se è necessario creare una variabile globale, ma sono bloccati nell'ambito locale, è possibile utilizzare la parola chiave **global**:

```
def myfunc():  
    global x  
    x = 300
```

Inoltre, utilizzare la parola chiave **global** se si desidera apportare una modifica a una variabile globale all'interno di una funzione.

```
x = 300
def myfunc():
    global x
    x = 200

myfunc()
print(x)
```

200

Superclassi astratte - Abstract Base Class (ABC)

Una superclasse astratta è una classe il cui comportamento è in parte specificato dalle sottoclassi.

Python, tramite il modulo **abc**, fornisce il supporto per definire formalmente una classe di base astratta.

```
from abc import ABCMeta, abstractmethod # need these definitions
```

```
class Sequence(metaclass=ABCMeta):
```

```
    """Our own version of collections.Sequence abstract base class."""
```

```
    @abstractmethod
```

```
    def len_(self):
```

```
        """Return the length of the sequence."""
```

```
    @abstractmethod
```

```
    def getitem_(self, j):
```

```
        """Return the element at index j of the sequence."""
```

Una **metaclass** fornisce un modello per la definizione della classe stessa, **ABCMeta** assicura che il costruttore della classe lanci un'eccezione quando si tenta di istanziare la classe astratta.

@abstractmethod è un decoratore, indica che non si fornisce un'implementazione del metodo (il metodo è astratto) e le classi derivate devono implementarlo.

I metodi statici e i metodi di classe

Un metodo di una classe normalmente riceve un'istanza della classe come primo argomento. A volte però i programmi necessitano di elaborare dati associati alle classi e non alle loro istanze. Ad esempio tenere traccia del numero di istanze della classe create.

Abbiamo però bisogno di metodi che non si aspettano di ricevere **self** come argomento e quindi funzionano indipendentemente dal fatto che esistano istanze della classe.

- **Metodi statici:** non ricevono **self** come argomento sia nel caso in cui vengano invocati su una classe, sia nel caso in cui vengano invocati su un'istanza della classe. Di solito tengono traccia di informazioni che riguardano tutte le istanze piuttosto che fornire funzionalità per le singole istanze
- **Metodi di classe:** ricevono un oggetto classe come primo argomento invece che un'istanza, sia che vengano invocati su una classe, sia nel caso in cui vengano invocati su un'istanza della classe. Questi metodi possono accedere ai dati della classe attraverso il loro argomento **cls** (corrisponde all'argomento **self** dei metodi "normali")

La funzione `printNumInstances` (non è né un metodo di classe né statico) non utilizza informazioni delle istanze ma solo informazioni della classe. Vogliamo invocarla senza far riferimento ad una particolare istanza:

```
class Spam:
    numInstances = 0
    def __init__(self):
        Spam.numInstances = Spam.numInstances + 1
    def printNumInstances():
        print("Number of instances created: %s" % Spam.numInstances)

>>> from spam import Spam
>>> a = Spam()
>>> b = Spam()
>>> c = Spam()

>>> Spam.printNumInstances()
Number of instances created: 3
>>> a.printNumInstances()
TypeError: printNumInstances() takes 0 positional arguments but 1 was given
```

- I metodi statici si definiscono invocando la funzione built-in **staticmethod**
- I metodi di classe si definiscono invocando la funzione built-in **classmethod**

```
# File bothmethods.py

class Methods:
    def imeth(self, x):
        print([self, x])

    def smeth(x):
        print([x])

    def cmeth(cls, x):
        print([cls, x])

    smeth = staticmethod(smeth)
    cmeth = classmethod(cmeth)
```

```
>>> Methods.smeth(3)
[3]
>>> obj.smeth(4)
[4]
```

```
>>> Methods.cmeth(5)
[<class 'bothmethods.Methods'>, 5]
>>> obj.cmeth(6)
[<class 'bothmethods.Methods'>, 6]
```

Alternativa per definire metodi statici e metodi di classe:

I metodi statici e i metodi di classe possono essere definiti usando i seguenti decoratori:

- ***@staticmethod***
- ***@classmethod***

```
@staticmethod
def smeth(x):
    print([x])

@classmethod
def cmeth(cls, x):
    print([cls, x])
```

Specificano comportamenti speciali per le funzioni e i metodi delle classi.

Creano intorno alla funzione un livello extra di logica implementato da un'altra funzione chiamata ***metafunzione*** (funzione che gestisce un'altra funzione).

Da un punto di vista sintattico, un decoratore di funzione è una sorta di dichiarazione riguardante la funzione che viene avviata durante l'esecuzione del programma. Un decoratore è specificato su una linea che precede lo ***statement def*** e consiste del simbolo ***@*** seguito da una ***metafunzione***.

Il decoratore di funzione può restituire la funzione originale così come è oppure restituire un nuovo oggetto che fa in modo che la funzione originale venga invocata indirettamente dopo aver eseguito il codice della ***metafunzione***.