

PYTHON

LA STORIA

Python è un popolare **linguaggio di programmazione** creato da Guido van Rossum e rilasciato nel **1991**.

UTILIZZI DI PYTHON:

- Sviluppo **web** (server-side);
- Sviluppo software;
- Calcoli matematici;
- Sistemi di scripting

COSA PUO' FARE:

- Può essere usato per **creare applicazioni web** su un server;
- Può connettersi a sistemi di **database** e può leggere e modificare **files**;
- Può essere usato per **gestire big data** e per eseguire complessi **calcoli matematici**;
- Può essere usato per la **prototipazione** e lo sviluppo di software pronti per la produzione.

PERCHE' PYTHON

- Python lavora su **diverse piattaforme** (Windows, Mac, Linux, Raspberry Pi, etc)
- Ha una **sintassi semplice** che permette di scrivere programmi usando **meno righe di codice** rispetto ad altri linguaggi di programmazione.
- Python è un **linguaggio interpretato**. Il codice può essere eseguito man mano che viene scritto e la prototipazione è molto veloce.
- Può essere usato per la **programmazione procedurale**, per la **programmazione object-oriented** o in modo **funzionale** (stile di programmazione software basato sulla valutazione di funzioni matematiche).

LA SINTASSI DI PYTHON RISPETTO AGLI ALTRI LINGUAGGI

- Python è stato progettato per la **leggibilità** ed ha delle somiglianze con la lingua inglese, con influenze dalla matematica
- Per completare un'istruzione usa **una nuova riga**; non usa parentesi o il punto e virgola;
- Python si affida all'**indentazione**, usando gli spazi bianchi **per definire lo scope di un'istruzione**, come ad esempio lo scope di un ciclo, di una funzione o di una classe. Altri linguaggi usano spesso le parentesi graffe.

INSTALLAZIONE DI PYTHON

VERIFICARE L'INSTALLAZIONE DI PYTHON SU WINDWS

Due metodi:

1. Cercare Python nella barra di ricerca iniziale



2. Lanciare la seguente istruzione sulla Command Line (cmd.exe)

```
C:\Users\Your Name>python --version
```

VERIFICARE L'INSTALLAZIONE DI PYTHON SU LINUX O MAC

In Linux aprire la command line, su Mac aprire il terminale e digitare:

```
python --version
```

Se non è presente alcuna versione di Python, scaricarlo da <https://www.python.org/>

PYTHON QUICKSTART

Python è un linguaggio di programmazione **interpretato**: lo sviluppatore scrive un file Python (.py) in un editor di testo (o un IDE) e lo passa all'interprete per essere eseguito.

Per lanciare un file Python da riga di comando digitare: **C\Users\Your Name>python helloworld.py**

Dove «helloworld» è il **nome del file Python**.

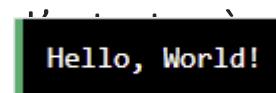
Creiamo un file Python chiamato helloworld.py utilizzando un qualsiasi editor di testo. All'interno del file scriviamo il codice

```
print ("Hello, World!")
```

e salviamo il file.

Apriamo la linea di comando, andiamo sulla directory nella quale è stato salvato il file e lanciamo il comando:

```
C\Users\Your Name>python helloworld.py
```



python vs py

python è l'eseguibile Python dell'installazione Python che hai selezionato come predefinito durante l'installazione. Questo in pratica inserisce il percorso di quella versione all'interno del PERCORSO, in modo che l'eseguibile sia direttamente disponibile.

py è il programma di avvio di Python che è un'utilità fornita con le installazioni di Python su Windows. Viene installato in C:\Windows\ quindi è disponibile senza richiedere modifiche al PERCORSO. Il programma di avvio di Python rileva quali versioni di Python sono installate sulla tua macchina ed è in grado di delegare automaticamente alla versione corretta. Per impostazione predefinita, utilizzerà l'ultima versione di Python presente sul tuo computer. Quindi, se hai installato 2.7, 3.5 e 3.6, l'esecuzione di py avvierà 3.6. Puoi anche specificare una versione diversa facendo ad es. py -3.5 per lanciare 3.5 o py -2 per avviare l'ultima versione di Python 2 sulla tua macchina.

```
PS C:\Corsi\Python\work> py -V
Python 3.10.4
PS C:\Corsi\Python\work> python -V
Python 3.8.5
PS C:\Corsi\Python\work>
```

```
py --list
```

è possibile installare più versioni di Python in Windows, per verificare le versioni installate:

```
py --list
Installed Pythons found by C:\WINDOWS\py.exe Launcher for Windows
-3.10-64 *
-3.9-64
-3.8-64
```

se si vuole utilizzare una determinata versione:

```
py -3.9
```

py vs python

se si vuole chiamare una determinata versione è sufficiente l'istruzione `py -*.*` dove `*.*` è la versione da eseguire

```
PS C:\Corsi\Python\work> py -3.8 --version  
Python 3.8.5  
PS C:\Corsi\Python\work> py -3.10 --version  
Python 3.10.4  
PS C:\Corsi\Python\work>
```

ad esempio per installare mysql-connector-python nell'ultima versione installata nel sistema:

```
|py -m pip install mysql-connector-python
```

per installare mysql-connector-python in una determinata versione:

```
|py -3.10 -m pip install mysql-connector-python
```

py vs python

altro esempio per lanciare da terminale uno script python con una determinata versione

```
|py -3.8 ./test.py  
|py -3.10 ./test.py
```

LINEA DI COMANDO DI PYTHON

Per testare una piccola porzione di codice, a volte è più rapido e semplice non scrivere il codice in un file. Questo è possibile perché Python può essere eseguito come **riga di comando**.

Digitare sulla riga di comando di Windows, Linux o Mac: **C\Users\Your Name>python**

Se il comando «python» non funziona, provare «py»: **C\Users\Your Name>python**

Ora si può digitare qualsiasi comando Python:

```
C:\Users\Your Name>python
Python 3.6.4 (v3.6.4:d48ebeb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello, World!")
```

Questo comando andrà a scrivere «Hello, World!» nella command line:

```
C:\Users\Your Name>python
Python 3.6.4 (v3.6.4:d48ebeb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello, World!")
Hello, World!
```

Per uscire dalla command line di Python, sarà sufficiente digitare il comando: **exit() o quit()**

LA SINTASSI – L'INDENTAZIONE

L'INDENTAZIONE IN PYTHON

L'indentazione è lo spazio che viene lasciato all'inizio di una riga di codice.

Negli altri linguaggi di programmazione, l'indentazione ha l'unico obiettivo di rendere il codice maggiormente leggibile, mentre in Python assume un ruolo molto importante.

Python **usa** infatti **l'indentazione per indicare un blocco di codice**.

```
if 5 > 2:  
    print("Five is greater than two!")
```

Se viene tralasciata l'indentazione, Python darà un messaggio di errore di sintassi.

```
if 5 > 2:  
print("Five is greater than two!")
```

La spaziatura dell'indentazione dipende dal programmatore, la più comune è di **4 spazi**, ma l'importante è che sia di almeno uno. L'importante è usare lo stesso numero di spazi in tutto il blocco di codice, altrimenti Python darà un messaggio di errore.

```
if 5 > 2:  
    print("Five is greater than two!")  
if 5 > 2:  
    print("Five is greater than two!")
```

I COMMENTI

COMMENTI

I commenti iniziano con **#** e servono per spiegare il codice, per renderlo più leggibile o per prevenire l'esecuzione del codice quando lo si testa.

```
#This is a comment.  
print("Hello, World!")
```

```
print("Hello, World!") #This is a comment
```

```
#print("Hello, World!")  
print("Cheers, Mate!")
```

MULTI LINE COMMENTS

Python **non ha una sintassi vera e propria per il commento multilinea**. Bisogna **aggiungere # ad ogni linea**.

```
#This is a comment  
#written in  
#more than just one line  
print("Hello, World!")
```

```
"""  
This is a comment  
written in  
more than just one line  
"""  
print("Hello, World!")
```

Oppure, anche se non previsto ufficialmente dalla sintassi di Python, si può usare una **stringa multi-linea**. Python infatti ignora le stringhe che non sono assegnate ad una variabile (le stringhe vengono lette ma ignorate) quindi basterà aggiungere una stringa multi-linea (compresa fra **tre apici**) al codice ed inserirvi il commento.

PEP

le regole per la scrittura del codice python sono indicate nel PEP 8

<https://www.python.it/doc/articoli/pep-8.html>

LE VARIABILI

LE VARIABILI

In Python le variabili vengono create quando viene assegnato loro un valore per la prima volta, non esiste un'istruzione apposita per dichiarare una variabile:

```
x = 5  
y = "John"  
print(x)  
print(y)
```

Non serve dichiarare il tipo di dato contenuto nella variabile e questo può anche cambiare dopo che la variabile è stata dichiarata.

```
x = 4      # x is of type int  
x = "Sally" # x is now of type str  
print(x)
```

CASTING

Il casting serve nel caso si voglia specificare o cambiare il tipo di dato di una variabile:

```
x = str(3)    # x will be '3'  
y = int(3)    # y will be 3  
z = float(3)  # z will be 3.0
```

LE VARIABILI

RECUPERARE IL TIPO DI DATO

Possiamo sapere che tipo di dato è stato assegnato ad una variabile con la funzione **type()**:

```
x = 5  
y = "John"  
print(type(x))  
print(type(y))
```

APICI DOPPI O SINGOLI PER LE STRINGHE

Le stringhe possono essere dichiarate sia usando i doppi apici che l'apice singolo:

```
x = "John"  
# is the same as  
x = 'John'
```

NOME VARIABILI E' CASE-SENSITIVE !!!

Il nome delle variabili è **case-sensitive** quindi a = 4 e A = 5 sono due variabili distinte.

```
a = 4  
A = "Sally"  
#A will not overwrite a
```

La variabile A non sovrascrive la variabile a.

LE VARIABILI – I nomi delle variabili

Una variabile può avere un nome breve (x, y) oppure un nome più descrittivo (age, total_volume).

REGOLE PER I NOMI DELLE VARIABILI IN PYTHON

- deve **iniziare con una lettera** o con **l'underscore (_)**
- **NON PUO'** iniziare con un **numero**
- può contenere **solo caratteri alfanumerici e underscores** (A-z, 0-9, _)
- i nomi delle variabili sono **case-sensitive** (age, Age, AGE sono tre variabili differenti)

NOMI
ACCETTATI:

```
myvar = "John"  
my_var = "John"  
_my_var = "John"  
myVar = "John"  
MYVAR = "John"  
myvar2 = "John"
```

NOMI NON
ACCETTATI:

```
2myvar = "John"  
my-var = "John"  
my var = "John"
```

MULTI WORDS NAME

Esistono diversi sistemi per rendere il nome di una variabile composto da più parole, maggiormente leggibile:

- **CAMEL CASE**: ogni parola, tranne la prima, inizia con la lettera maiuscola
`myVariableName = 'John'`
- **PASCAL CASE**: ogni parola inizia con una lettera maiuscola
`MyVariableName = 'John'`
- **SNAKE CASE**: ogni parola è separata da un underscore
`my_variable_name = 'John'`

LE VARIABILI – Assegnare valori multipli

1- ASSEGNAME PIU' VALORI A VARIABILI MULTIPLE

Python consente di assegnare valori a variabili multiple in un'unica riga di codice.

Assicurarsi che **il numero di variabili corrisponda al numero dei valori** altrimenti si riceverà un messaggio di errore.

```
x, y, z = "Orange", "Banana", "Cherry"  
print(x)  
print(y)  
print(z)
```

Orange
Banana
Cherry

2- ASSEGNAME UN VALORE A PIU' VARIABILI

Si può assegnare lo stesso valore a più variabili

```
x = y = z = "Orange"  
print(x)  
print(y)  
print(z)
```

Orange
Orange
Orange

3- ESTRARRE VALORI DA UNA COLLECTION

Se si ha un insieme di valori in una lista, tupla, ecc. Python permette di estrarne i valori e assegnarli a variabili. Questa operazione è chiamata **UNPACKING**.

```
fruits = ["apple", "banana", "cherry"]  
x, y, z = fruits  
  
print(x)  
print(y)  
print(z)
```

apple
banana
cherry

LE VARIABILI – L'output delle variabili

Per l'output delle variabili viene spesso usata la funzione `print()`.

```
x = "Python is awesome"  
print(x)
```

Python is awesome

Per l'output di **più variabili**, basta separarle con la **virgola**

```
x = "Python"  
y = "is"  
z = "awesome"  
print(x, y, z)
```

Python is awesome

Si può anche utilizzare **l'operatore +** per l'output di più variabili (**notare lo spazio** dopo "Python" e "is", senza il quale il risultato sarebbe *Pythonisawesome*)

```
x = "Python "  
y = "is "  
z = "awesome"  
print(x + y + z)
```

Python is awesome

Per i **numeri**, l'operatore + funziona come **normale operatore matematico**

```
x = 5  
y = 10  
print(x + y)
```

15

Se si tenta di combinare una stringa con un numero con l'operatore +, Python darà **errore**.

```
x = 5  
y = "John"  
print(x + y)
```

TypeError: unsupported operand type(s) for +: 'int' and 'str'

Il modo migliore per l'output di più variabili con la funzione `print()`, è **separare le variabili con una virgola**, che supporta anche tipi di dati differenti.

```
x = 5  
y = "John"  
print(x, y)
```

5 John

TIPI DI DATI

Text Type: `str`

Numeric Types: `int` , `float` , `complex`

Sequence Types: `list` , `tuple` , `range`

Mapping Type: `dict`

Set Types: `set` , `frozenset`

Boolean Type: `bool`

Binary Types: `bytes`

None Type: `NoneType`

BUILT-IN DATA TYPES

In programmazione, il concetto di tipo di dato è molto importante.

Le variabili possono memorizzare tipi di dato differenti. A loro volta, tipi di dati differenti possono fare cose differenti.

Qui a lato i tipi di dato predefiniti di Python.

TIPI DI DATI – Getting e setting data type

IMPOSTARE IL TIPO DI DATO

In Python il tipo di dato è deciso **quando si assegna
una valore ad una variabile.**

RECUPERARE IL TIPO DI DATO

Usando la funzione **type()**:

```
x = 5  
print(type(x))
```

```
<class 'int'>
```

Example	Data Type
x = "Hello World"	str
x = 20	int
x = 20.5	float
x = 1j	complex
x = ["apple", "banana", "cherry"]	list
x = ("apple", "banana", "cherry")	tuple
x = range(6)	range
x = {"name" : "John", "age" : 36}	dict
x = {"apple", "banana", "cherry"}	set
x = frozenset({"apple", "banana", "cherry"})	frozenset
x = True	bool
x = b"Hello"	bytes
x = None	NoneType

TIPI DI DATI – Impostare uno specifico tipo di dato

IMPOSTARE UN TIPO DI DATO SPECIFICO PER UNA VARIABILE

Per specificare il tipo di dato di una variabile, si può usare il costruttore:

`x = datatype(value)`

ESEMPIO:

`x = str('Hello World')`

Example	Data Type
<code>x = str("Hello World")</code>	str
<code>x = int(20)</code>	int
<code>x = float(20.5)</code>	float
<code>x = complex(1j)</code>	complex
<code>x = list(("apple", "banana", "cherry"))</code>	list
<code>x = tuple(("apple", "banana", "cherry"))</code>	tuple
<code>x = range(6)</code>	range
<code>x = dict(name="John", age=36)</code>	dict
<code>x = set(("apple", "banana", "cherry"))</code>	set
<code>x = frozenset(("apple", "banana", "cherry"))</code>	frozenset
<code>x = bool(5)</code>	bool
<code>x = bytes(5)</code>	bytes

I NUMERI IN PYTHON

In Python ci sono 3 differenti tipi numerici di dato:

- `int` `x = 1 # int`
- `float` `y = 2.8 # float`
- `complex` `z = 1j # complex`

1-INT (or INTEGER)

E' un numero intero, positivo o negativo, senza decimali, di lunghezza illimitata.

```
x = 1  
y = 35656222554887711  
z = -3255522
```

2-FLOAT (or FLOATING POINT NUMBER)

E' un numero intero, positivo o negativo, che contiene uno o più decimali.

Può anche essere un numero scientifico con una «e» ad indicare la potenza di 10.

```
x = 1.10  
y = 1.0  
z = -35.59  
  
x = 35e3  
y = 12E4  
z = -87.7e100
```

3-COMPLEX

Sono scritti con una «j» come parte immaginaria.

```
x = 3+5j  
y = 5j  
z = -5j
```

NUMERI IN PYTHON – Conversione tra tipi e numeri random

In Python è possibile convertire un tipo di dato numerico ad un altro con i metodi `int()`, `float()` and `complex()`

```
x = 1    # int
y = 2.8  # float
z = 1j   # complex

#convert from int to float:
a = float(x)

#convert from float to int:
b = int(y)

#convert from int to complex:
c = complex(x)  = (1+0j)
```

```
print(x)
print(y)
print(z)

print(type(x))
print(type(y))
print(type(z))
```

```
1.0
2
(1+0j)
<class 'float'>
<class 'int'>
<class 'complex'>
```

NON E' POSSIBILE CONVERTIRE NUMERI COMPLEX IN ALTRI TIPI NUMERICI DI DATO

```
>>> c = 1+2j
>>> print(int(c))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: int() argument must be a string, a bytes-like object or a real number, not 'complex'
>>> print(int(c))
```

NUMERI RANDOM

Python non ha una funzione `random()` che crea un numero casuale. Possiede però un modulo built-in chiamato `random` che può essere usato per creare numeri casuali

Per importare il modulo `random` e visualizzare un numero casuale tra 1 e 9:
`import random`
`print (random.randrange(1, 10))`

CASTING IN PYTHON – Specificare un tipo di variabile

A volte si ha la necessità di **cambiare il tipo di variabile**.

Questo può essere fatto con il **casting**.

Python è un linguaggio orientato agli oggetti e, in quanto tale, usa classi per definire i tipi di dati, inclusi i suoi tipi primitivi.

Il casting in Python viene quindi eseguito utilizzando le **funzioni di costruzione**:

- **int()**: crea un numero intero **da un integer**, da un **float** (rimuovendo tutti i decimali) o **da una stringa** (fornendo una stringa che rappresenta un numero intero)
- **float()**: crea un float number **da un intero**, **da un float** o **da una stringa** (dove la stringa rappresenta un intero o un float)
- **str()**: crea una stringa **da una grande varietà di tipi di dati** inclusi string, integer e float

INTEGERS

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

FLOAT

```
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
```

STRING

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

LE STRINGHE IN PYTHON

In Python le stringhe sono racchiuse sia tra singoli apici che tra doppi apici.

'hello' è lo stesso di "hello"

Si può visualizzare il contenuto di una stringa con la funzione **print()**

ASSEGNAME UNA STRINGA AD UNA VARIABILE

```
a = "Hello"  
print(a)
```

ASSEGNAME UNA STRINGA MULTI LINEA AD UNA VARIABILE

Si può assegnare una stringa multi linea ad una variabile usando **tre doppi apici** o **tre singoli apici**.

Le interruzioni di linea sono inserite come nel codice.

```
print("Hello")  
print('Hello')
```

```
Hello  
Hello
```

Differenza tra """ e \

```
a = "ciao \  
sono \  
una \  
stringa"  
  
print(a) # ciao sono una stringa
```

```
a = """ciao  
sono  
una  
stringa"""  
  
print(a)  
# ciao  
# sono  
# una  
# stringa
```

```
a = """Lorem ipsum dolor sit amet,  
consectetur adipiscing elit,  
sed do eiusmod tempor incididunt  
ut labore et dolore magna aliqua."""  
print(a)
```

```
a = '''Lorem ipsum dolor sit amet,  
consectetur adipiscing elit,  
sed do eiusmod tempor incididunt  
ut labore et dolore magna aliqua.'''  
print(a)
```

```
 Lorem ipsum dolor sit amet,  
consectetur adipiscing elit,  
sed do eiusmod tempor incididunt  
ut labore et dolore magna aliqua.
```

LE STRINGHE IN PYTHON – Stringhe sono Arrays

Come in molti altri linguaggi di programmazione, le stringhe in Python sono arrays di bytes che rappresentano caratteri unicode.

Python però non ha il data type character, un singolo carattere è semplicemente una stringa con lunghezza pari a 1.

Le parentesi quadre possono essere usate per accedere gli elementi della stringa.

Per recuperare il carattere in posizione 1 (ricordarsi bene che il primo elemento ha posizione [0]):

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

e

LE STRINGHE IN PYTHON – len(), in, not in

LUNGHEZZA DI UNA STRINGA

Per conoscere la lunghezza di una stringa si usa la funzione len() che ritorna la lunghezza della stringa.

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

13

ESAMINARE UNA STRINGA – if IN

Per controllare **se una certa frase o carattere è presente in una stringa**, si usa la keyword **in**.

Per controllare se «free» è presente nel testo:

```
txt = "The best things in life are free!"  
print("free" in txt)
```

True

Può essere usato anche **in** in un **if statement**

```
txt = "The best things in life are free!"  
if "free" in txt:  
    print("Yes, 'free' is present.")
```

Yes, 'free' is present.

```
>>> print("anna" in "arianna")  
>>> True
```

ESAMINARE UNA STRINGA – NOT IN

Per controllare **se una certa frase o carattere NON è presente in una stringa**, si usa la keyword **not in**.

Per controllare se «expensive» NON è presente nel testo:

```
txt = "The best things in life are free!"  
print("expensive" not in txt)
```

True

Può essere usato anche **in** in un **if statement**

```
# STAMPA SOLO SE "expensive" NON E' PRESENTE  
txt = "The best things in life are free!"  
if "expensive" not in txt:  
    print("No, 'expensive' is NOT present.")
```

No, 'expensive' is NOT present.

LE STRINGHE IN PYTHON – tagliare una stringa (slice)

1. TAGLIARE UNA STRINGA (SLICE)

Per ritornare una gamma di caratteri si usa la sintassi per tagliare una stringa.

Vanno specificati **l'inizio e la fine della stringa** che si desidera ritornare, separati da due punti.

Il primo carattere ha posizione 0

2. TAGLIARE DALL'INIZIO

Omettendo l'indice del carattere iniziale, la gamma di caratteri recuperati partirà dal primo carattere della stringa.

3. TAGLIARE FINO ALLA FINE

Omettendo l'indice finale, la gamma di caratteri recuperati arriverà fino alla fine della stringa.

4. INDICE NEGATIVO

Si usa un indice negativo per iniziare a tagliare la stringa dalla sua fine.

1. Per ritornare i caratteri **dalla posizione 2 alla posizione 5 (non compreso)**

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

1lo

2= posizione del **primo carattere** che si desidera recuperare
5= posizione dell'ultimo carattere (non compreso: l'ultimo carattere ritornato è quello in posizione 4)

2. Per ritornare i caratteri **dall'inizio della stringa alla posizione 5 (non compresa)**

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

Hello

3. Recuperare i caratteri **dalla posizione 2 fino alla fine** della stringa

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

1lo, World!

4. Recuperare i caratteri **dalla «o» di «World!» (posizione -5) a «d» (non incluso) in «World!»**

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

orl

LE STRINGHE IN PYTHON – step negativo

UTILIZZANDO UNO STEP NEGATIVO SI HA UN RISULTATO SOLAMENTE SE L'INIZIO è MAGGIORE DELLA FINE

ad esempio

```
print('ciao'[3:0:-1]) # oai
print('ciao'[0:3:-1]) # (vuoto non da risultato)
print('ciao'[0:3:])

print('ciao'[-1:-4:-1]) # oai
print('ciao'[-4:-1:-1]) # (vuoto non da risultato)
print('ciao'[-4:-1]) # cia
```

LE STRINGHE IN PYTHON – Modificare le stringhe

- UPPER CASE: per ritornare la stringa in **caratteri maiuscoli** si usa il metodo **upper()**

```
a = "Hello, World!"  
print(a.upper())
```

HELLO, WORLD!

- LOWER CASE: per ritornare la stringa in **caratteri minuscoli** si usa il metodo **lower()**

```
a = "Hello, World!"  
print(a.lower())
```

hello, world!

- RIMUOVERE GLI SPAZI BIANCHI: per rimuovere lo spazio prima e/o dopo un testo si usa il metodo **strip()** che rimuove ogni spazio bianco all'inizio o alla fine della stringa.

```
a = " Hello, World! "  
print(a.strip()) # returns "Hello, World!"
```

- SOSTITUIRE UNA STRINGA: il metodo **replace()** sostituisce una stringa con un'altra stringa

```
a = "Hello, World!"  
print(a.replace("H", "J"))
```

Jello, World!

- DIVIDERE UNA STRINGA IN SOTTOSTRINGHE: il metodo **split()** ritorna una lista dove il testo che si trova tra i separatori specificati, diventa un elemento della lista. La stringa viene quindi divisa in sottostringhe se vengono trovate istanze del separatore (la virgola nell'esempio sotto).

```
a = "Hello, World!"  
b = a.split(",")  
print(b)
```

['Hello', ' World!']

LE STRINGHE IN PYTHON – Concatenazione

CONCATENARE STRINGHE

Per concatenare o combinare due stringhe si usa l'operatore **+**.

ESEMPI

- Per unire la variabile **a** con la variabile **b** nella variabile c:

```
a = "Hello"  
b = "World"  
c = a + b  
print(c)
```

HelloWorld

- Per aggiungere uno spazio tra a e b, aggiungere " " :

```
a = "Hello"  
b = "World"  
c = a + " " + b  
print(c)
```

Hello World



LE STRINGHE IN PYTHON – Formattazione stringhe

FORMATTARE STRINGHE

Non si possono **combinare stringhe e numeri in questo modo:**

```
age = 36
txt = "My name is John, I am " + age
print(txt)

Traceback (most recent call last):
  File "demo_string_format_error.py", line 2, in <module>
    txt = "My name is John, I am " + age
TypeError: must be str, not int
```

Per combinare stringhe e numeri si usa il **metodo format()** che prende gli elementi passati, li **formatta** e li inserisce all'interno della stringa in corrispondenza del **segnaposto {}**:

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

Il metodo format accetta un numero illimitato di argomenti e li posiziona nei rispettivi segnaposto

```
quantity = 3
itemno = 567
price = 49.95
myorder = "I want {} pieces of item {} for {} dollars."
print(myorder.format(quantity, itemno, price))
#return I want 3 pieces of item 567 for 49.95 dollars.
```

Si possono usare **gli indici {0}, {1}....**per assicurarsi che gli argomenti siano posizionati nel corretto segnaposto.

```
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))
#return I want to pay 49.95 dollars for 3 pieces of item 567
```

```
>>> print("ciao {1} {0}".format("tutti", "a"))
ciao a tutti
```

LE STRINGHE IN PYTHON f-strings

FORMATTARE STRINGHE

Un'altra modalità per combinare stringhe è l'f-strings che consente di inserire gli elementi all'interno della stringa in corrispondenza del segnaposto { }:

```
>>> name = "Eric"  
>>> age = 25  
>>> f"Hello, {name}. You are {age}."  
'Hello, Eric. You are 25.'
```

è possibile anche effettuare dei calcoli

```
>>> print(f"risultato = {2*7}")
```



LE STRINGHE IN PYTHON – Caratteri di Escape

Per inserire in una stringa caratteri che non sono consentiti (ad esempio un doppio apice all'interno di una stringa racchiusa tra doppi apici), si usa il carattere di escape Backslash \ seguito dal carattere che si vuole inserire.

NON CONSENTITO

```
txt = "We are the so-called "Vikings" from the north."
```

```
File "demo_string_escape_error.py", line 1
    txt = "We are the so-called "Vikings" from the north."
                                         ^
SyntaxError: invalid syntax
```

Per risolvere il problema, si usa il carattere di escape \

```
txt = "We are the so-called \"Vikings\" from the north."
```

```
We are the so-called "Vikings" from the north.
```

CARATTERI DI ESCAPE

Altri caratteri di escape usati in Python sono:

Code	Result
\'	Single Quote
\\"	Backslash
\n	New Line
\r	Carriage Return
\t	Tab
\b	Backspace
\f	Form Feed
\ooo	Octal value
\xhh	Hex value

LE STRINGHE IN PYTHON - metodi

- **count()**

ritorna il numero di volte in cui un valore appare in una stringa

`string.count(value, start, end)`

```
txt = "I love apples, apple are my favorite fruit"
x = txt.count("apple")
print(x)
#return 2
```

- **endswith() – startswith()**

verifica se la stringa finisce (o inizia, nel caso di startswith) con un valore specificato. Ritorna True in caso positivo.

`string.endswith(value, start, end)`

```
txt = "Hello, welcome to my world."
x = txt.endswith(".")
print(x)
#return True
```

```
txt = "Hello, welcome to my world."
x = txt.startswith("Hello")
print(x)
#return True
```

- **join()**

Unisce in una stringa tutti gli elementi all'interno di una variabile iterabile (ad esempio una tuple). Deve essere specificata una stringa da usare come separatore.

```
print("-".join('prova')) # p-r-o-v-a
```

```
>>> print(" ".join("ciao a tutti"))
ciao a tutti
```

```
myTuple = ("John", "Peter", "Vicky")
x = "#".join(myTuple)
print(x)
#return John#Peter#Vicky
```

- **replace()**

Sostituisce una specifica frase con un'altra specifica frase.

!!!! Saranno sostituite tutte le occorrenze della frase specificata se non è indicato diversamente.

`string.replace(oldvalue, newvalue, count)`

```
#Sostituisce tutte le occorrenze della parola "one":
txt = "one one was a race horse, two two was one too."
x = txt.replace("one", "three")
print(x)
#return three three was a race horse, two two was three too.
```

```
#Sostituisce le prime due occorrenze della parola "one":
txt = "one one was a race horse, two two was one too."
x = txt.replace("one", "three", 2)
print(x)
#return three three was a race horse, two two was one too.
```



LE STRINGHE IN PYTHON – Metodi: strip(), lstrip(), rstrip()

• **strip()**

Rimuove o tronca i caratteri che vengono passati come parametro (di default gli spazi bianchi), all'inizio (leading characters) E alla fine (trailing characters) della stringa originale.

Ritorna la stringa modificata.

string.strip(characters)

```
txt = "      banana      "
x = txt.strip()
print("of all fruits", x, "is my favorite")
```

of all fruits banana is my favorite

```
txt = ",,,,,rrrtgg.....banana.....rrr"
x = txt.strip(",.grt")
print(x)
```

banana

• **lstrip()**

ritorna una copia della stringa dalla quale sono stati rimossi i caratteri iniziali (a seconda degli argomenti passati)

```
txt = "      banana      "
x = txt.lstrip()
print("of all fruits", x, "is my favorite")
```

of all fruits banana is my favorite

• **rstrip()**

ritorna una copia della stringa dalla quale sono stati rimossi i caratteri finali (a seconda degli argomenti passati)

```
txt = "      banana      "
x = txt.rstrip()
print("of all fruits", x, "is my favorite")
```

of all fruits banana is my favorite

LE STRINGHE IN PYTHON – Metodi: find(), rfind()

• **find()**

Trova la prima occorrenza di un valore specificato e ne ritorna la posizione.

Se non viene trovato alcun valore, ritorna -1.

Simile al metodo index() con la sola differenza che quest'ultimo segnala un exception se il valore non viene trovato.

string.find(value, start, end)

```
txt = "Hello, welcome to my world."
x = txt.find("welcome")           #If the value is not found, the find() method returns -1,
print(x)                          # but the index() method will raise an exception:
#return 7                         txt = "Hello, welcome to my world."
                                    print(txt.find("q"))
                                    print(txt.index("q"))
```

```
# Where in the text is the first occurrence of the letter "e"
# when you only search between position 5 and 10?:
```

```
txt = "Hello, welcome to my world."
x = txt.find("e", 5, 10)
print(x)

#return 8
```

```
-1
Traceback (most recent call last):
  File "demo_ref_string_find_vs_index.py", line 4 in <module>
    print(txt.index("q"))
ValueError: substring not found
```

• **rfind()**

Trova l'ultima occorrenza di un valore specificato e ne ritorna la posizione.

Se non viene trovato alcun valore, ritorna -1.

Simile al metodo rindex()

```
txt = "Hello, welcome to my world."
x = txt.rfind("e")
print(x)
#return 13
```

LE STRINGHE IN PYTHON – Metodi: index(), rindex()

• index()

Trova la prima occorrenza di un valore specificato e ne ritorna la posizione.

Segnala un exception se il valore non viene trovato.

```
string.index(value, start, end)
```

```
txt = "Hello, welcome to my world."
x = txt.index("welcome")
print(x)
#return 7

# Where in the text is the first occurrence of the letter "e"
#when you only search between position 5 and 10?

txt = "Hello, welcome to my world."
x = txt.index("e", 5, 10)

print(x)
#return 8
```

```
txt = "Hello, welcome to my world.

print(txt.find("q"))
print(txt.index("q"))
```

```
-1
Traceback (most recent call last):
  File "demo_ref_string_find_vs_index.py", line 4 in <module>
    print(txt.index("q"))
ValueError: substring not found
```

• rindex()

Trova l'ultima occorrenza di un valore specificato e ne ritorna la posizione.

Segnala un exception se il valore non viene trovato.

Simile al metodo rfind()

```
txt = "Hello, welcome to my world."
x = txt.rindex("e")
print(x)
#return 13
```

```
#Ultima occorrenza di "e" nelle posizioni tra 5 e 10
txt = "Hello, welcome to my world."
x = txt.rindex("e", 5, 10)
print(x)
#return 8
```

```
txt = "Hello, welcome to my world.

print(txt.rfind("q"))
print(txt.rindex("q"))
```

```
-1
Traceback (most recent call last):
  File "demo_ref_string_find_vs_index.py", line 4 in <module>
    print(txt.index("q"))
ValueError: substring not found
```

LE STRINGHE IN PYTHON – Metodi: split(), rsplit()

• **split()**

Divide una stringa in una lista. E' possibile specificare il separatore in base al quale dividere la stringa, quello di default è lo spazio. Se viene indicato il secondo parametro (maxsplit) la lista conterrà il numero specificato di elementi più uno.

string.split(separator, maxsplit)

```
#Divide la stringa, usando la virgola, seguita da uno spazio, come separatore:  
txt = "hello, my name is Peter, I am 26 years old"  
x = txt.split(", ")  
print(x)
```

```
['hello', 'my name is Peter', 'I am 26 years old']
```

```
#Usa un hash come separatore  
txt = "apple#banana#cherry#orange"  
x = txt.split("#")  
print(x)
```

```
['apple', 'banana', 'cherry', 'orange']
```

```
txt = "apple#banana#cherry#orange"  
  
# setting the maxsplit parameter to 1, will return a list with 2 elements!  
x = txt.split("#", 1)  
  
print(x)
```

```
['apple', 'banana#cherry#orange']
```

• **rsplit()**

Divide una stringa in una lista partendo da destra. Se maxsplit non è specificato, ritornerà la stessa lista del metodo split().

string.rsplit(separator, maxsplit)

```
txt = "apple, banana, cherry"  
  
# setting the maxsplit parameter to 1,  
# will return a list with 2 elements!  
  
x = txt.rsplit(", ", 1)  
  
print(x)
```

```
# note that the result has only 2 elements  
#"apple, banana" is the first element, and "cherry" is the last.
```

```
['apple, banana', 'cherry']
```

LE STRINGHE IN PYTHON – Metodi: partition(), rpartition()

•partition()

Cerca una specifica stringa e divide la stringa in una tupla contenente tre elementi:

- 1- primo elemento: contiene la parte che precede la stringa specificata
- 2- secondo elemento: contiene la stringa specificata
- 3- terzo elemento: contiene la parte che succede la stringa specificata.

Cerca la PRIMA OCCORRENZA della stringa specificata.
`string.partition(value)`

```
txt = "I could eat bananas all day"
x = txt.partition("bananas")
print(x)
```

```
('I could eat ', 'bananas', ' all day')
```

```
#Se la stringa specificata non viene trovata, viene ritornata una tupla che contiene
# 1 - l'intera stringa
# 2 - una stringa vuota
# 3 - una stringa vuota:

txt = "I could eat bananas all day"
x = txt.partition("apples")
print(x)
```

```
('I could eat bananas all day', '', '')
```

•rpartition()

Cerca L'ULTIMA OCCORRENZA di una specifica stringa e divide la stringa in una tupla di 3 elementi.

```
txt = "I could eat bananas all day, bananas are my favorite fruit"
x = txt.rpartition("bananas")
print(x)
```

```
('I could eat bananas all day, ', 'bananas', ' are my favorite fruit')
```

STRINGHE IN PYTHON – Metodi

- **isalnum()**

Controlla se tutti i caratteri di una stringa sono alfanumerici. Ritorna True se sono tutti alfanumerici (lettere e/o numeri).

string.isalnum()

```
txt = "Company12"
x = txt.isalnum()
print(x)
#return True
```

- **isalpha()**

Ritorna True se tutti i caratteri di una stringa sono alfabetici. (lettere)

string.isalpha()

```
txt = "CompanyX"
x = txt.isalpha()
print(x)
#return True
```

- **isdigit()**

Ritorna True se tutti i caratteri di una stringa sono numeri. (numeri)

string.isdigit()

es 2^2 è digit

```
txt = "50800"
x = txt.isdigit()
print(x)
#return True
```

```
a = "\u0030" #unicode for 0
b = "\u00B2" #unicode for  $^2$ 
print(a.isdigit()) #return True
print(b.isdigit()) #return True
```

- **isnumeric()**

ritorna True se tutti i caratteri sono numeri (0-9). Esponenti come 2 e $\frac{1}{4}$ sono considerati numeri. -1 e 1.5 **NON sono considerati numeri** perché TUTTI i caratteri della stringa **devono essere numeri** e il trattino - ed il punto . non lo sono

```
a = "\u0030" #unicode for 0
b = "\u00B2" #unicode for  $^2$ 
c = "10km2"
d = "-1"
e = "1.5"
```

```
print(a.isnumeric())      #True
print(b.isnumeric())      #True
print(c.isnumeric())      #False
print(d.isnumeric())      #False
print(e.isnumeric())      #False
```

- per 2 ad apice ALT+0178

STRINGHE – islower(), isupper(), istitle()

• **islower() – isupper()**

Ritornano True se tutti i caratteri sono rispettivamente lower case o upper case.

```
a = "Hello world!"  
b = "hello 123"  
c = "mynameisPeter"  
d = "my text"  
print(a.islower()) #False  
print(b.islower()) #True  
print(c.islower()) #False  
print(d.islower()) #True
```

```
a = "Hello World!"  
b = "hello 123"  
c = "MY NAME IS PETER"  
  
print(a.isupper()) #False  
print(b.isupper()) #False  
print(c.isupper()) #True
```

• **istitle()**

Ritorna True se ogni parola all'interno di un testo inizia con la lettera maiuscola E le altre lettere sono minuscole.

```
a = "HELLO, AND WELCOME TO MY WORLD"  
b = "Hello"  
c = "22 Names"  
d = "This Is %'!?"  
txt = "Hello, And Welcome To My World!"  
  
print(a.istitle()) #False  
print(b.istitle()) #True  
print(c.istitle()) #True  
print(d.istitle()) #True  
print(txt.istitle()) #True
```

```
print(a.istitle()) #False  
print(b.istitle()) #True  
print(c.istitle()) #True  
print(d.istitle()) #True  
print(txt.istitle()) #True
```

• **lower() – upper()**

Ritornano una stringa rispettivamente in caratteri minuscoli e maiuscoli

```
txt = "Hello my FRIENDS"  
x = txt.lower()  
print(x)  
# return hello my friends
```

```
txt = "Hello my FRIENDS"  
x = txt.upper()  
print(x)  
# return HELLO MY FRIENDS
```

• **title()**

Ritorna una stringa in cui la prima lettera delle parole è maiuscola.

```
txt = "Hello my FRIENDS"  
x = txt.title()  
print(x)  
# return Hello My Friends
```

isnumeric(), isdecimal

isnumeric() verifica se una stringa ha un valore numerico

isdecimal() verifica se tutti i caratteri di una stringa sono numeri [0-9]

```
print('ciao'.isnumeric()) # False
print('10'.isnumeric())   # True
print('10.5'.isnumeric()) # False
print('10j'.isnumeric()) # False
```

```
print('*****')
print('ciao'.isdecimal()) # False
print('10'.isdecimal())  # True
print('10.5'.isdecimal()) # False
print('10j'.isdecimal()) # False
```

```
str = "2¾" # ALT + 243
print(str.isdigit()) #False
print(str.isnumeric()) #True
```

per verificare se un numero è un float (10.5) utilizzare una function con try

```
def isfloat(num):
    try:
        float(num)
        return True
    except ValueError:
        return False

print(isfloat('s12'))    # False
print(isfloat('1.123')) # True
```

oppure utilizzare isinstance

```
print(isinstance(2.5, float))      # True
```

I BOOLEANI IN PYTHON

I booleani rappresentano uno dei due valori: **True** o **False**.

In programmazione si ha spesso bisogno di sapere se un'espressione è vera o falsa.

Si può valutare qualsiasi espressione in Python per ottenere una delle due risposte, True o False.

Ad esempio, quando si comparano due valori, l'espressione è valutata e Python ritorna una risposta booleana:

```
print(10 > 9)    #return True
print(10 == 9)   #return False
print(10 < 9)    #return False
```

Quando si lancia una condizione if, Python ritorna true o false.

```
#Print a message based on whether the condition is True or False:
a = 200
b = 33

if b > a:
    print("b is greater than a")
else:
    print("b is not greater than a")

#return b is not greater than a
```

VALUTARE VALORI E VARIABILI

La funzione `bool()` permette di valutare qualsiasi valore e ritornare True o False

- Valutare una stringa ed un numero:

```
print(bool("Hello"))
print(bool(15))
```

True
True

OPERATORI IN PYTHON

GRUPPI DI OPERATORI IN PYTHON:

- operatori **aritmetici**
- Operatori di **assegnamento**
- Operatori di **comparazione**
- Operatori **logici**
- Operatori di **identità**
- Operatori di **membership**
- Operatori **bitwise** (usati per comparare numeri binari)

Gli operatori sono usati per **effettuare operazioni** su valori e variabili.

Ad esempio, si usa l'operatore **+** per sommare due valori.

```
print(10 + 5)  
# return 15
```

OPERATORI IN PYTHON – Operatori numerici

OPERATORI MATEMATICI

Usati con valori numerici per effettuare operazioni matematiche comuni

OPERATORE	NOME	ESEMPIO
+	Addizione	$x+y$
-	Sottrazione	$x-y$
*	Moltiplicazione	$x*y$
/	Divisione	x/y
%	Modulo	$x \% y$
**	Elevamento a potenza	$x^{**}y$
//	Divisione arrotondata	$x//y$

```
x = 5
y = 3

# ADDIZIONE
print(x + y)

# SOTTRAZIONE
print (x - y)

# MOLTIPLICAZIONE
print (x * y)

# DIVISIONE
print (x / y)

# MODULO
print (x % y)

# ELEVAMENTO A POTENZA
print (x**y)

# DIVISIONE CON ARROTONDAMENTO
print (x//y)
```

```
ADDIZIONE
8

SOTTRAZIONE
2

MOLTIPLICAZIONE
15

DIVISIONE
1.6666666666666667

MODULO
2

ELEVAMENTO A POTENZA
125

DIVISIONE CON ARROTONDAMENTO
1
```

OPERATORI IN PYTHON – Operatori di assegnamento

OPERATORI DI ASSEGNAZIONE

Usati per assegnare valori alle variabili

OPERATORE	ESEMPIO	EQUIVALE A
=	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 = 5
x += 3
print(x) #return 8

x = 5
x -= 3
print(x) #return 2

x = 5
x *= 3
print(x) #return 15

x = 5
x /= 3
print(x) #return 1.6666666666666667

x = 5
x %= 3
print(x) #return 2

x = 5
x //= 3
print(x) #return 1

x = 5
x **= 3
print(x) #return 125
```

y = x += 5 # assegnazione a nuova variabile contemporaneamente NON si Può FARE

OPERATORI IN PYTHON – Operatori di assegnamento

OPERATORI DI ASSEGNAZIONE

OPERATORE	ESEMPIO	EQUIVALE A
&=	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

Usati per assegnare valori alle variabili

^ (xor bitwise)

```
>>> print(bin(14))
>>> print(bin(5))
1110 il 14 in bit
101 il 5 in bit
>>> print(5 ^ 14)
risultato: 11
```

1011 (perché in bit il 11 è
0b1011)
0 0 vero, 0 1 vero, 1 1 falso

<< (left shift bitwise)

```
>>> print(bin(5))
0b101
>>> print(bin(5 << 1))
0b1010 #aggiunge uno 0 a destra
>>> print(bin(5 << 2))
0b10100 # aggiunge due 0 a destra
>>> print(bin(5 << 3))
0b101000 # aggiunge tre 0 a destra
>>> print(bin(5 << 2)) aggiunge 2
zeri a destra
```

>> (right shift bitwise)

```
>>> print(bin(5))
0b101
>>> print(bin(5 >> 1))
0b10 #rimuove 1 a destra
>>> print(bin(5 >> 2))
0b1 # rimuove 2 a destra
>>> print(bin(5 >> 3))
0b0 # rimuove 3 a destra
```

```
x = 5
x &= 3
print(x) #return 1

x = 5
x |= 3
print(x) #return 7

x = 5
x ^= 3
print(x) #return 6

x = 5
x >>= 3
print(x) #return 0

x = 5
x <<= 3
print(x) #return 40
```

& (and bitwise)

```
>>> print(bin(14))
>>> print(bin(5))
1110 il 14 in bit
101 il 5 in bit
>>> print(5 & 14)
risultato: 4
100 (perché in bit il 4 è 0b100)
```

| (or bitwise)

```
>>> print(bin(14))
>>> print(bin(5))
1110 il 14 in bit
101 il 5 in bit
>>> print(5 | 14)
risultato: 15
1111 (perché in bit il 15 è
0b1111)
```

come vederlo:

```
n = 0b1111
>>> print(n)
15
```



OPERATORI IN PYTHON – Operatori di comparazione

OPERATORI DI COMPARAZIONE

Usati per confrontare e comparare due valori

OPERATORE	NOME	ESEMPIO
<code>==</code>	UGUALE	<code>x == y</code>
<code>!=</code>	DIVERSO	<code>x != y</code>
<code>></code>	MAGGIORI	<code>x > y</code>
<code><</code>	MINORE	<code>x < y</code>
<code>>=</code>	MAGGIORI O UGUALE	<code>x>=y</code>
<code><=</code>	MINORE O UGUALE	<code>x<=y</code>

```
x = 5
y = 3
print(x == y)
# returns False perchè 5 NON E' UGUALE A 3
```

```
x = 5
y = 3
print(x != y)
# returns True perchè 5 NON E' UGUALE A 3
```

```
x = 5
y = 3
print(x > y)
# returns True perchè 5 è MAGGIORI di 3
```

```
x = 5
y = 3
print(x < y)
# returns False perchè 5 è NON E' MINORE di 3
```

```
x = 5
y = 3
print(x >= y)
# returns True perchè 5 è MAGGIORI di 3
```

```
x = 5
y = 3
print(x <= y)
# returns False perchè 5 NON E' MINORE O UGUALE a 3
```

OPERATORI IN PYTHON – Operatori logici

OPERATORI DI COMPARAZIONE

Usati per creare combinazioni di condizioni

OPERATORE	DESCRIZIONE	ESEMPIO
and	Ritorna true se entrambe le condizioni sono vere	$x < 5 \text{ and } x < 10$
or	Ritorna true se una delle due condizioni è vera	$x < 5 \text{ or } x < 4$
not	Inverte il risultato, ritorna False se il risultato è True	not ($x < 5 \text{ and } x < 10$)

```
x = 5
print(x > 3 or x < 4)
# returns True perchè una delle condizioni è vera (5 is greater than 3, but 5 is not less than 4)
```

```
x = 5
print(x > 3 and x < 10)
# returns True perchè 5 is maggiorre di 3 AND 5 is minore than 10
```

```
x = 5
print(not(x > 3 and x < 10))
# returns False perchè "not" è usato per invertire il risultato
```

OPERATORI IN PYTHON – Operatori di membership (in)

OPERATORI DI MEMBERSHIP

Usati per testare se una sequenza è presente in un oggetto.

OPERATORE	DESCRIZIONE	ESEMPIO
in	Ritorna True se una sequenza con il valore specificato è presente in un oggetto	x in y
not in	Ritorna True se una sequenza con il valore specificato è NON PRESENTE nell'oggetto	x not in y

```
x = ["apple", "banana"]
print("banana" in x)
# returns True because a sequence with the value "banana" is in the list

x = ["apple", "banana"]
print("pineapple" not in x)
# returns True because a sequence with the value "pineapple" is not in the list
```

PYTHON COLLECTIONS

PYTHON COLLECTIONS (ARRAYS)

Esistono 4 tipi di collezioni di dati in Python:

- **LIST**: collezione modificabile e ordinata. Consente membri duplicati.
- **TUPLE**: ordinata e NON modificabile. Consente membri duplicati.
- **SET**: collection NON ORDINATA, NON modificabile e NON indicizzata. Non ammette membri duplicati. I set non sono modificabili ma si possono aggiungere e/o rimuovere elementi a piacere.
- **DICTIONARY**: collection ordinata (a partire dalla versione di Python 3.7) e modificabile. Non ammette membri duplicati.

Quando si sceglie che tipo di collection utilizzare è utile comprendere le proprietà di quel tipo. Scegliere il tipo più adatto può portare un miglioramento dell'efficienza e della sicurezza.

Modificabile= non si può modificare un elemento tipo `tupla[1]='ciao'` oppure `mio_set[1] = ciao`

COLLECTION	ORDINATA	MODIFICABILE	MEMBRI DUPLICATI	SINTASSI
LIST	✓	✓	✓	[]
TUPLE	✓	✗	✓	()
SET	✗	✗	✗	{ }
DICTIONARY	✓	✓	✗	{ }

PYTHON LIST

Le liste sono usate per **memorizzare oggetti multipli** in una singola variabile.

Sono uno dei 4 **tipi di dati built-in** di Python usati per **memorizzare collezioni di dati**. Gli altri 3 sono le Tuple, i Set e i Dictionary, tutti con proprietà e usi differenti.

Le liste sono create elencando **tra parentesi quadre** una serie di oggetti separati da virgole (oppure con il metodo list()), come si vedrà in seguito).

```
thislist = ["apple", "banana", "cherry"]  
print(thislist)
```

```
['apple', 'banana', 'cherry']
```

IL COSTRUTTORE list()

Per creare una lista è possibile anche utilizzare il metodo **list()**.

```
thislist = list(("apple", "banana", "cherry")) #notare le doppie parentesi tonde  
print(thislist)  
#returns ['apple', 'banana', 'cherry']
```

ELEMENTI DELLA LISTA

Gli elementi della lista sono:

- **indicizzati**: Gli elementi della lista sono identificati da un indice, il primo elemento ha indice [0], il secondo [1] e così via.
- **ordinati**: gli elementi hanno un ordine definito e quell'**ordine non cambierà**. Se si aggiungono nuovi elementi alla lista, il nuovo elemento sarà posizionato alla fine.
Esistono dei metodi che possono cambiare l'ordine della lista ma in generale, esso non cambia.
- **modificabili**: dopo la creazione della lista, gli elementi possono essere modificati, aggiunti e rimossi.
- **consentono valori duplicati**: essendo la lista indicizzata, può contenere anche elementi con lo stesso valore.

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

PYTHON LIST – lenght,data types, type

• LIST LENGTH – len()

Per determinare quanti elementi contiene una lista, si usa la funzione **len()**

```
thislist = ["apple", "banana", "cherry"]
print(len(thislist))
#return 3
```

• type()

Dal punto di vista di Python, le liste sono definite come oggetti di tipo 'list'(<class 'list'>)

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

print(type(mylist))
#returns <class 'list'>
```

ELEMENTI DELLA LISTA – TIPI DI DATI

Gli elementi della lista possono essere di **qualsiasi tipo**.

```
#STRING
list1 = ["apple", "banana", "cherry"]
#INTEGER
list2 = [1, 5, 7, 9, 3]
#BOOLEAN
list3 = [True, False, False]

print(list1) #return ['apple', 'banana', 'cherry']
print(list2) #return [1, 5, 7, 9, 3]
print(list3) #return [True, False, False]
```

La lista PUÒ anche contenere **tipi di dati diversi tra loro**.

```
list1 = ["abc", 34, True, 40, "male"]

print(list1)
#returns ['abc', 34, True, 40, 'male']
```

PYTHON LIST – Accedere gli elementi della list

Gli elementi della lista sono indicizzati e sono **accedibili attraverso il loro indice**

```
# Accedere al secondo elemento della list
thislist = ["apple", "banana", "cherry"]
print(thislist[1])
# returns banana
```

RANGE DI INDICI

Si può specificare un intervallo di valori indicando da **gli indici di inizio e fine dell'intervallo**.

Quando si specifica un intervallo di valori, verrà ritornata una nuova lista contenente gli elementi specificati.

L'indice iniziale è compreso nell'intervallo, mentre quello finale è escluso.

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[2:5])

#This will return the items from position 2 to 5(NOT INCLUDED).

#Remember that the first item is position 0,
#and note that the item in position 5 is NOT included

#returns ['cherry', 'orange', 'kiwi']
```

Se si omette il valore di partenza, l'intervallo inizierà dal primo elemento.

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[:4])

#Ritorna gli elemtni dall'indice 0 all'indice 4 (NON COMPRESO).
#Remember that index 0 is the first item, and index 4 is the fifth item

#return ['apple', 'banana', 'cherry', 'orange']
```

Se si omette il valore finale, l'intervallo arriverà fino all'ultimo elemento.

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[2:])

#Ritorna gli elementi dall'indice 2 fino alla fine

#Returns ['cherry', 'orange', 'kiwi', 'melon', 'mango']
```

PYTHON LIST – Accedere gli elementi della list

INDICE NEGATIVO

Inserire un indice negativo significa iniziare a contare gli elementi della lista dalla fine. -1 è l'ultimo elemento, -2 è il penultimo, ecc.

```
thislist = ["apple", "banana", "cherry"]
print(thislist[-1])
#returns cherry
```

RANGE DI INDICI NEGATIVI

Per iniziare la ricerca dalla fine della lista, specificare indici negativi.

```
#Ritorna gli elementi da "orange" (-4) a "mango" (-1) che però non è incluso:
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[-4:-1])

#Indici negativi vogliono dire partire *dalla fine della lista.
#Ricordarsi che l'indice dell'ultimo elemento è -1

#Returns ['orange', 'kiwi', 'melon']
```

CONTROLLARE SE UN ELEMENTO ESISTE - in

Per verificare la presenza di uno specifico elemento all'interno della lista si usa `in`.

```
#Controlla se l'elemento "apple" è nella lista

thislist = ["apple", "banana", "cherry"]
if "apple" in thislist:
    print("Yes, 'apple' is in the fruits list")

#returns Yes, 'apple' is in the fruits list
```

PYTHON LIST – Modificare gli elementi della list

• CAMBIARE VALORE DI UN ELEMENTO

Per modificare il valore di uno specifico elemento gli si assegna un nuovo valore usando questa sintassi:

```
list[index] = new_value

#Change the second item:

thislist = ["apple", "banana", "cherry"]
thislist[1] = "blackcurrant"

print(thislist)

#returns ['apple', 'blackcurrant', 'cherry']
```

• CAMBIARE I VALORI DI UN INTERVALLO DI ELEMENTI

Per modificare il valore di elementi che si trovano all'interno di uno specifico intervallo, si definisce una list con i nuovi valori e si indicano gli indici dell'intervallo all'interno del quale si vogliono inserire i nuovi valori.

```
#Cambia i valori "banana" e "cherry" con i valori "blackcurrant" e "watermelon"

thislist = ["apple", "banana", "cherry", "orange", "kiwi", "mango"]
thislist[1:3] = ["blackcurrant", "watermelon"]

print(thislist)

#returns ['apple', 'blackcurrant', 'watermelon', 'orange', 'kiwi', 'mango']
```

Se si inseriscono **più valori** rispetto agli elementi indicati nell'intervallo, i nuovi elementi saranno inseriti dopo quelli modificati e gli altri si sposteranno di conseguenza.

```
#Cambia il secondo valore sostituendolo con due nuovi valori
thislist = ["apple", "banana", "cherry"]
thislist[1:2] = ["blackcurrant", "watermelon"]

print(thislist)

#returns ['apple', 'blackcurrant', 'watermelon', 'cherry']
```

Se si inseriscono **meno valori**, i nuovi valori saranno inseriti e gli elementi dell'intervallo per cui non è specificato un nuovo valore, saranno eliminati.

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

thislist[1:3] = ["watermelon"]
print(thislist)

#returns ['apple', 'watermelon']
```

La **lunghezza della lista** cambia quando il **numero degli elementi inseriti non corrisponde al numero degli elementi da sostituire**

PYTHON LIST – Aggiungere elementi

• **insert()**

Per inserire un nuovo elemento nella lista senza sostituire un valore già esistente, si usa il metodo `insert()` che inserisce un elemento nell'indice specificato.
NON si può passare una lista

Sintassi: `listname.insert(index,element)`

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

thislist.insert(2, "watermelon")

print(thislist)

#returns ['apple', 'banana', 'watermelon', 'cherry']
```

• **append()**

Inserisce un nuovo elemento alla fine della lista.
NON si può passare una lista

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

thislist.append("orange")

print(thislist)

#returns ['apple', 'banana', 'cherry', 'orange']
```

• **extend()**

Aggiunge gli elementi di una lista alla fine della lista corrente.

```
thislist = ["apple", "banana", "cherry"]
tropical = ["mango", "pineapple", "papaya"]

thislist.extend(tropical)

print(thislist)
#returns ['apple', 'banana', 'cherry', 'mango', 'pineapple', 'papaya']
```

Il metodo `extend()` può aggiungere alla lista corrente gli elementi di qualsiasi oggetto iterabile (tuple, sets, dictionaries...)

```
thislist = ["apple", "banana", "cherry"]
thistuple = ("kiwi", "orange")

thislist.extend(thistuple)

print(thislist)
#returns ['apple', 'banana', 'cherry', 'kiwi', 'orange']
```

PYTHON LIST – Rimuovere elementi dalla list

- **remove()**

Rimuove l'elemento specificato (il **VALORE**, NON L'INDICE)

```
thislist = ["apple", "banana", "cherry"]
thislist.remove("banana")
print(thislist)
#returns ['apple', 'cherry']
```

- **pop()**

Rimuove l'elemento all'indice specificato.

```
thislist = ["apple", "banana", "cherry"]
thislist.pop(1)
print(thislist)
#returns ['apple', 'cherry']
```

Se l'indice non è specificato, il metodo pop() **rimuove l'ultimo elemento** della lista.

```
thislist = ["apple", "banana", "cherry"]
thislist.pop()
print(thislist) .
#returns ['apple', 'banana']
```

- **del**

Come il metodo pop(), anche del **rimuove l'indice specificato**.

```
thislist = ["apple", "banana", "cherry"]
del thislist[0]
print(thislist)
#returns ['banana', 'cherry']
```

Del può anche **eliminare l'intera lista**

```
#elimina l'intera list
thislist = ["apple", "banana", "cherry"]
del thislist
print(thislist)
#this will cause an error because you have successfully deleted "thislist".
```

```
Traceback (most recent call last):
  File "demo_list_del2.py", line 3, in <module>
    print(thislist) #this will cause an error because you have successfully deleted "thislist".
NameError: name 'thislist' is not defined
```

- **clear()**

Svuota la lista: la lista rimane, non viene eliminata, ma è vuota

```
thislist = ["apple", "banana", "cherry"]
thislist.clear()
print(thislist)
#returns []
```

PYTHON LIST – ciclare le liste – CICLO FOR

CICLO FOR

Si possono ciclare gli elementi della lista usando un ciclo for:

```
for item in list:  
    # process the item
```

In ogni iterazione, alla variabile item viene assegnato un singolo elemento della lista list.

All'interno del for si può agire su ogni elemento individualmente.

```
cities = ['New York', 'Beijing', 'Cairo', 'Mumbai', 'Mexico']  
  
for city in cities:  
    print(city)
```

```
New York  
Beijing  
Cairo  
Mumbai  
Mexico
```

In questo caso, il ciclo for assegna un elemento della lista cities alla variabile city e ne stampa il valore in ogni iterazione.

CICLARE ATTRAVERSO GLI INDICI

Si può ciclare la lista facendo riferimento agli indici.
Per farlo si usa i metodi **range()** e **len()**.

Il metodo **range()** ritorna una **sequenza di numeri** che, di default, parte da 0, **incrementa di 1** e si ferma all'indice specificato.
range(start?, stop, step?)

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

```
3  
4  
5
```

```
thislist = ["apple", "banana", "cherry"]  
for i in range(len(thislist)):  
    print(thislist[i])
```

```
apple  
banana  
cherry
```

L'oggetto iterabile che viene creato con il metodo range() nell'esempio sopra è [0, 1, 2]

PYTHON LIST – ciclare le liste – CICLO WHILE

CICLO WHILE

Per ciclare gli elementi della lista si può usare un ciclo while.

Per **determinare la lunghezza della lista** si usa la funzione **len()** e si ciclano gli elementi usando i loro indici.

Ricordarsi di **incrementare l'indice di 1 dopo ogni iterazione**.

Vedi ciclo while

```
# Stampa tutti gli elementi usando un ciclo while attraverso tutti gli indici
thislist = ["apple", "banana", "cherry"]
i = 0
while i < len(thislist):
    print(thislist[i])
    i = i + 1
```

```
apple
banana
cherry
```

CICLARE USANDO LA LIST COMPREHENSION

La list comprehension offre la **sintassi più semplice** per ciclare le liste

```
#stampa tutti gli elementi nella lista
thislist = ["apple", "banana", "cherry"]
[print(x) for x in thislist]
```

```
apple
banana
cherry
.
```

PYTHON LIST – LIST COMPREHENSION

Le List comprehensions offrono una **sintassi abbreviata e semplificata** per **creare liste e sottoliste**.

Partendo da una lista di frutta, supponiamo di voler creare una nuova lista contenente solo i frutti che contengono la lettera «a» nel loro nome.

Senza list comprehension, dovremmo scrivere un for con una condizione al suo interno:

Grazie alla list comprehension è possibile scrivere tutto con **una sola riga di codice**.

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = []

for x in fruits:
    if "a" in x:
        newlist.append(x)

print(newlist)
```

```
['apple', 'banana', 'mango']
```

```
newlist = [x for x in fruits if "a" in x]
```

PYTHON LIST – LIST COMPREHENSION

LA SINTASSI

```
newlist = [expression for item in iterable if condition == True]
```

Viene ritornata una nuova lista lasciando quella precedente inalterata.

- CONDITION

la condizione è come un filtro che accetta solo gli elementi che vengono valutati come True.

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

#la condizione accetta solo gli elementi che non sono "apple"
newlist = [x for x in fruits if x != "apple"]

print(newlist)
#returns ["banana", "cherry", "kiwi", "mango"]
```

La condizione `if x != "apple"` ritorna True per tutti gli elementi a parte "apple", facendo in modo che la nuova lista contenga tutti gli elementi tranne "apple".

La condizione è opzionale e può essere omessa:

```
newlist = [x for x in fruits]
```

- ITERABLE (oggetto iterabile)

Può essere qualsiasi oggetto iterabile come list, tuple, set ecc.

Per creare un oggetto iterabile si può usare la funzione `range()`:

```
newlist = [x for x in range(10)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Stesso esempio ma con una condizione:

```
#accetta solo numeri minori di 5
newlist = [x for x in range(10) if x < 5]
[0, 1, 2, 3, 4]
```

PYTHON LIST – LIST COMPREHENSION

• EXPRESSION

L'expression è l'elemento corrente nell'iterazione, ma è anche un risultato che si può manipolare prima che divenga un elemento della nuova lista che si sta creando.

```
#trasforma il valore della nuova lista in uppercase  
  
newlist = [x.upper() for x in fruits]  
  
['APPLE', 'BANANA', 'CHERRY', 'KIWI', 'MANGO']
```

Si può agire sul risultato come si preferisce

```
#imposta tutti i valori della lista a "hello"  
  
newlist = ['hello' for x in fruits]  
  
'hello', 'hello', 'hello', 'hello', 'hello']
```

L'expression può anche contenere delle condizioni che non agiscono come filtro ma come modo per manipolare il risultato

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]  
  
newlist = [x if x != "banana" else "orange" for x in fruits]  
  
['apple', 'orange', 'cherry', 'kiwi', 'mango']
```

Questa expression dice: "ritorna l'element se non è "banana", se è "banana" ritorna "orange"

PYTHON LIST – ORDINARE LA LIST – sort()

ORDINARE LE LIST ALFANUMERICAMENTE

Gli oggetti di tipo list possiedono il metodo sort() che consente di ordinare la list alfanumericamente in maniera ascendente (default):

ORDINARE LA LISTA ALFABETICAMENTE

```
thislist = ["orange", "mango", "kiwi", "pineapple", "banana"]
thislist.sort()
print(thislist)
#['banana', 'kiwi', 'mango', 'orange', 'pineapple']
```

ORDINARE LA LISTA NUMERICAMENTE

```
thislist = [100, 50, 65, 82, 23]
thislist.sort()
print(thislist)
#returns [23, 50, 65, 82, 100]
```

ORDINE DISCENDENTE (reverse = True)

Per ordinare la list in maniera discendente si usa l'argomento reverse = True passato alla funzione sort()

ORDINE ALFABETICO DISCENDENTE

```
thislist = ["orange", "mango", "kiwi", "pineapple", "banana"]
thislist.sort(reverse = True)
print(thislist)
#returns ['pineapple', 'orange', 'mango', 'kiwi', 'banana']
```

ORDINE NUMERICO DISCENDENTE

```
thislist = [100, 50, 65, 82, 23]
thislist.sort(reverse=True)
print(thislist)
#returns [100, 82, 65, 50, 23]
```

PYTHON LIST – ORDINARE LA LIST – custom sort

Si può creare la propria funzione di ordinamento usando l'argomento

key = function

Il metodo o la funzione assegnati a key saranno applicati a tutti gli elementi della lista prima che questa venga ordinata e specificheranno le logiche del criterio di ordinamento.

Ipotizziamo di voler ordinare una lista di stringhe in base alla loro lunghezza.

Per farlo si assegna il metodo built-in len() al parametro key.

Il metodo len() conterà la lunghezza di ogni elemento della list.

```
programming_languages = ["Python", "Swift", "Java", "C++", "Go", "Rust"]

programming_languages.sort(key=len)

print(programming_languages)

#output

#[ 'Go', 'C++', 'Java', 'Rust', 'Swift', 'Python']
```

Le stringhe saranno ordinate **sempre in ordine ascendente** ma in base alla loro lunghezza.

Il **parametro key e il parametro reverse possono essere combinati**.

Ad esempio, per ordinare una list in base alla lunghezza delle stringhe ma in ordine discendente

```
programming_languages.sort(key=len, reverse=True)

#output
#
#[ 'Python', 'Swift', 'Java', 'Rust', 'C++', 'Go']
```

SECONDO ESEMPIO

La funzione myfunc() ritorna un numero assoluto che sarà utilizzato per ordinare la list in base alla vicinanza degli elementi al numero 50. L'ordine sarà sempre ascendente perché non diversamente specificato.

```
#Sort the list based on how close the number is to 50:

def myfunc(n):
    return abs(n - 50)

thislist = [100, 50, 65, 82, 23]
thislist.sort(key = myfunc)
print(thislist)

#output
#[50, 65, 23, 82, 100]
```

PYTHON LIST – ORDINARE LA LIST

ORDINAMENTO CASE SENSITIVE

Di default il metodo sort() è case sensitive con il risultato che le lettere maiuscole saranno messe sempre prima delle lettere minuscole, dando dei risultati non ottimali per un ordinamento alfabetico

```
thislist = ["banana", "Orange", "Kiwi", "cherry"]
thislist.sort()
print(thislist)

#output
#['Kiwi', 'Orange', 'banana', 'cherry']
```

Per ovviare al problema si possono usare funzioni built-in come key functions. In questo caso, per ottenere una funzione di ordinamento case-insensitive, si può usare str.lower come key function

```
thislist = ["banana", "Orange", "Kiwi", "cherry"]
thislist.sort(key=str.lower)
print(thislist)

#output
#[ 'banana', 'cherry', 'Kiwi', 'Orange']
```

INVERTIRE L'ORDINE – reverse()

Per invertire l'ordine degli elementi della lista senza ordinarli si usa il metodo reverse()

```
thislist = ["banana", "Orange", "Kiwi", "cherry"]
thislist.reverse()
print(thislist)

#output
#[ 'cherry', 'Kiwi', 'Orange', 'banana']
```

PYTHON LIST – COPIARE UNA LIST

NON è possibile semplicemente copiare una list digitando

```
list2 = list1
```

perché list2 sarà solo un riferimento a list1 (avranno la stessa
allocazione di memoria) e tutte le modifiche fatte a list1 saranno
automaticamente apportate anche in list2.

METODO copy()

Uno dei modi per creare una copia di una list è usare il metodo
built-in delle list **copy()**

```
thislist = ["apple", "banana", "cherry"]
mylist = thislist.copy()
print(mylist)

#output
#[ 'apple', 'banana', 'cherry']
```

METODO list()

Un altro modo per creare una copia di una list è usare il metodo
built-in **list()**

```
thislist = ["apple", "banana", "cherry"]
mylist = list(thislist)
print(mylist)
#output
#[ 'apple', 'banana', 'cherry']
```

PYTHON LIST – unire lists

Esistono diversi sistemi per unire o concatenare due o più liste.

CONCATENAZIONE (+)

```
list1 = ["a", "b", "c"]
list2 = [1, 2, 3]

list3 = list1 + list2
print(list3)

#output
#[ 'a', 'b', 'c', 1, 2, 3]
```

EXTEND()

```
list1 = ["a", "b" , "c"]
list2 = [1, 2, 3]

list1.extend(list2)
print(list1)

#output
#[ 'a', 'b', 'c', 1, 2, 3]
```

APPEND()

```
list1 = ["a", "b" , "c"]
list2 = [1, 2, 3]

for x in list2:
| list1.append(x)

print(list1)

#output
#[ 'a', 'b', 'c', 1, 2, 3]
```

PYTHON LIST – unire lists

Non si può copiare una list digitando

12 = 11

Perché **12** sarebbe solo un riferimento a **11** e le modifiche fatte a **12** verrebbero automaticamente fatte a **11**.

Ci sono diversi modi per fare una copia:

1. METODO BUILT-IN `copy()`

11.`copy()` *non accetta argomenti.*

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

```
l1 = l.copy()
```

```
l1[1] = 4
```

```
print(l)
```

Quando si usa il metodo `copy()` viene creata una nuova list popolata con i riferimenti dalla lista originale.

2. METODO BUILT-IN `list()`

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

```
l2=list(l)
```

```
l2[1] = 5
```

```
print(l)
```

PYTHON LIST – METODI BUILT-IN

METODO	DESCRIZIONE
<code>append()</code>	Aggiunge un elemento alla fine della lista
<code>clear()</code>	Rimuove tutti gli elementi della lista
<code>copy()</code>	Ritorna una copia della lista
<code>count()</code>	Ritorna il numero di elementi con il valore specificato
<code>extend()</code>	Aggiunge gli elementi di una lista (o di qualsiasi oggetto iterabile) alla fine della lista corrente
<code>index()</code>	Ritorna l'indice del primo elemento con uno specifico valore
<code>insert()</code>	Aggiunge un elemento nella posizione specificata
<code>pop()</code>	Rimuove l'elemento che ha l'indice specificato
<code>remove()</code>	Rimuove l'elemento che ha il valore specificato
<code>reverse()</code>	Inverte l'ordine della lista
<code>sort()</code>	Ordina la lista

LE TUPLES

Così come le lists, anche le tuple sono un **insieme di elementi ordinati**.

A differenza delle lists, però, le tuple sono **IMMUTABILI**.

Questo significa che **non possono essere cambiate**.

Altra differenza rispetto alle lists è che quest'ultime sono scritte con le parentesi quadre, mentre **le tuple sono scritte con parentesi tonde**.

DEFINIRE UNA TUPLA

```
rgb = ('red', 'green', 'blue')
```

per definire una tuple si può anche utilizzare il **metodo tuple()**

```
thistuple = tuple(("apple", "banana", "cherry")) # note the double round-brackets
```

ELEMENTI DI UNA TUPLA

Gli elementi di una tuple sono ordinati, immutabili ed ammettono valori duplicati.

Sono **indicizzati** e, come nelle lists, **il primo elemento ha indice [0]**.

Una volta definita la tuple, **i singoli elementi sono accedibili attraverso i rispettivi indici**.

```
rgb = ('red', 'green', 'blue')
```

```
print(rgb[0])
```

red

```
print(rgb[1])
```

green

```
print(rgb[2])
```

blue

Le tuple sono **ordinate**, questo significa che **gli elementi hanno un ordine definito** e quest'ordine non cambierà.

IMMUTABILI

Le tuple **NON possono essere modificate**, questo vuol dire che **non possono essere aggiunti o rimossi elementi** dopo la loro creazione.

ACCETTA VALORI DUPLICATI

Essendo indicizzate, le tuple **possono contenere elementi con lo stesso valore**.

LE TUPLES

LUNGHEZZA DI UNA TUPLE

Per determinare quanti elementi possiede una tuple, si usa la funzione `len()`

```
thistuple = tuple(("apple", "banana", "cherry"))
print(len(thistuple))
#returns 3
```

CREARE UNA TUPLE CON SOLO UN ELEMENTO

Per creare una tuple con un solo elemento, bisogna aggiungere una virgola dopo quell'elemento altrimenti Python non la riconoscerà come tuple.

```
thistuple = ("apple")
print(type(thistuple))

#NOT a tuple
thistuple = ("apple")
print(type(thistuple))
```



```
<class 'tuple'>
<class 'str'>
```

ELEMENTI DELLA TUPLE – TIPI DI DATI

Gli elementi della tuple possono essere di **qualsiasi tipo**. Può anche contenere tipi di dati differenti contemporaneamente.

```
tuple1 = ("apple", "banana", "cherry") #('apple', 'banana', 'cherry')
tuple2 = (1, 5, 7, 9, 3) #(1, 5, 7, 9, 3)
tuple3 = (True, False, False) #(True, False, False)
```

ESEMPIO CON TIPI DIFFERENTI DI DATI

```
tuple1 = ("abc", 34, True, 40, "male") #('abc', 34, True, 40, 'male')
```

`type()`

Per Pythons le tuples sono oggetti di tipo 'tuple'

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

#output
# <class 'tuple'>
```

TUPLE – ACCEDERE GLI ELEMENTI DELLE TUPLE

E' possibile accedere gli elementi delle tuples **riferendosi al loro indice**, inserito tra parentesi quadre. L'indice del primo elemento è 0

```
thistuple = ("apple", "banana", "cherry")
print(thistuple[1]) #output banana
```

INDICIZZAZIONE NEGATIVA

Se si usa un indice negativo, si considerano gli elementi **partendo dalla fine della tuple**.
-1 è l'ultimo elemento, -2 è il penultimo e così via.

```
thistuple = ("apple", "banana", "cherry")
print(thistuple[-1]) #output: cherry
```

INTERVALLO DI INDICI

Si può indicare un intervallo di indici degli elementi che si vogliono recuperare specificando gli indici di inizio e fine intervallo. Quando si specifica un intervallo, il valore di ritorno sarà una **nuova tuple** contenente gli elementi specificati. L'elemento avente per indice il valore di fine intervallo non è compreso.

Come per le lists, se si **omette l'indice iniziale** l'intervallo partire dall'indice 0; se si **omette l'indice finale**, l'intervallo arriverà fino all'ultimo elemento della tuple.

```
#Ritorna il terzo, quarto e quinto elemento.
thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
print(thistuple[2:5])
```

```
#This will return the items from position 2 to 5.
#and note that the item in position 5 is NOT included
#output: ('cherry', 'orange', 'kiwi')
```

INTERVALLO DI INDICI NEGATIVI

Specificare indici negativi se si vuole **iniziare l'intervallo dalla fine della tuple**.

```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
print(thistuple[-4:-1])
```

```
#Negative indexing means starting from the end of the tuple.
#This example returns the items from index -4 (included) to index -1 (excluded)
#Remember that the last item has the index -1,
```

```
#output: ('orange', 'kiwi', 'melon')
```

CONTROLLARE L'ESISTENZA DI UN ELEMENTO

Per verificare la presenza di un elemento all'interno di una tuple, **si usa la keyword in**.

```
thistuple = ("apple", "banana", "cherry")
if "apple" in thistuple:
    print("Yes, 'apple' is in the fruits tuple")
    #output: Yes, 'apple' is in the fruits tuple
```

TUPLE – AGGIORNARE LE TUPLES

```
x = ("apple", "banana", "cherry")
y = list(x)
y[1] = "kiwi"
x = tuple(y)

print(x)
#returns: ("apple", "kiwi", "cherry")
```

```
#Crea una nuova tuple con l'elemento "orange", e aggiunge questa tuple:
thistuple = ("apple", "banana", "cherry")
y = ("orange",)
thistuple += y

print(thistuple)

#returns: ('apple', 'banana', 'cherry', 'orange')
```

Le tuples sono immutabili ed immodificabili, questo significa che non si possono cambiare una volta che la tuple è stata creata.

Esistono però delle **soluzioni alternative**.

Cambiare valori in una tuple

Come detto le tuple sono immutabili, ma è possibile modificare un valore **convertendo la tuple in list, modificare la list e convertire di nuovo la list in tuple**.

AGGIUNGERE ELEMENTI

Essendo immutabili, le tuples non hanno un metodo built-in append() ma, per aggiungere loro nuovi elementi, si può:

1. **CONVERTIRE LA TUPLE IN LIST**, come per la modifica degli elementi
2. **AGGIUNGERE TUPLE AD UNA TUPLE**: è consentito aggiungere tuples ad altre tuples.
Quindi, se si vogliono aggiungere uno o più elementi ad una tuple, si potrà creare una nuova tuple ed aggiungerla all'esistente

TUPLE – AGGIORNARE LE TUPLES

DOPPIA CONVERSIONE

```
thistuple = ("apple", "banana", "cherry")
y = list(thistuple)
y.remove("apple")
thistuple = tuple(y)
#returns:
('banana', 'cherry')
```

ELIMINAZIONE TUPLE

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

Traceback (most recent call last):
  File "demo_tuple_del.py", line 3, in <module>
    print(thistuple) #this will raise an error because the tuple no longer exists
NameError: name 'thistuple' is not defined
```

RIMUOVERE ELEMENTI

NON SI POSSONO RIMUOVERE ELEMENTI DA UNA TUPLE.

Si possono però adottare gli stessi sistemi già visti per la modifica e l'aggiunta di elementi.

- Si può quindi convertire la tuple in list, eliminare l'elemento desiderato e riconvertire la list in tuple.
- Si può eliminare la tuple completamente

TUPLE – packing and unpacking

Quando si crea una tuple, normalmente le si **assegnano dei valori**. Questa operazione è chiamata **PACKING**:

```
fruits = ("apple", "banana", "cherry")  
  
print(fruits)  
#returns ('apple', 'banana', 'cherry')
```

Quando invece **estraiamo i valori di una tuple in variabili** si parla di **UNPACKING**. Si dividono quindi gli elementi della tuple in variabili multiple.

```
(green, yellow, red) = fruits  
  
print(green)  
print(yellow)  
print(red)
```

```
apple  
banana  
cherry
```

possibile anche green, yellow, red = fruits

Questa espressione assegna i valori della tupla a destra ad ogni variabile sulla sinistra basandosi sulla rispettiva posizione di ogni elemento.

Il numero delle variabili deve corrispondere al numero degli elementi della tuple. Se non corrispondono si può usare un **asterisco** per raccogliere gli elementi rimanenti della tuple in una list.

UTILIZZO DELL'ASTERISCO *

Se il **numero di variabili è inferiore al numero degli elementi** della tuple, si aggiunge un asterisco al nome della variabile e i valori saranno assegnati alla variabile come list.

```
fruits = ("apple", "banana", "cherry", "strawberry", "raspberry")  
  
(green, yellow, *red) = fruits  
  
print(green)  
print(yellow)  
print(red)
```

```
apple  
banana  
['cherry', 'strawberry', 'raspberry']
```

Se l'asterisco è **assegnato ad una variabile diversa dall'ultima**, Python assegnerà alla variabile valori finche il numero di elementi rimasti nella tupla non corrisponderà al numero di variabili rimaste

```
fruits = ("apple", "mango", "papaya", "pineapple", "cherry")  
  
(green, *tropic, red) = fruits  
  
print(green)  
print(tropic)  
print(red)
```

```
apple  
['mango', 'papaya', 'pineapple']  
cherry
```

Green = 'apple'
Tropic = ['mango', 'papaya', 'pineapple']
Red = 'cherry'

TUPLE – ciclo for e ciclo while

CICLO WHILE

Si possono ciclare gli elementi della tuple usando un ciclo for, similmente a quanto si fa con le lists.

```
thistuple = ("apple", "banana", "cherry")
for x in thistuple:
    print(x)
```

```
apple
banana
cherry
```

Cicla gli elementi della tuple e ne stampa il risultato

CICLO FOR USANDO GLI INDICI DEGLI ELEMENTI

Si possono ciclare gli elementi della tuple facendo riferimento ai loro indici.

Si usano le funzioni range() e len() per creare un oggetto iterabile adatto.

```
thistuple = ("apple", "banana", "cherry")
for i in range(len(thistuple)):
    .
    print(thistuple[i])
```

```
apple
banana
cherry
```

Cicla gli elementi della tuple facendo riferimento al loro indice e ne stampa il risultato

CICLO WHILE

Per **determinare la lunghezza della tuple**, si usa la **funzione len()**.

Si fa partire poi il ciclo da indice 0 . Ricordarsi di aumentare l'indice di 1 dopo ogni iterazione.

Indice iniziale = 0 →
thistuple = ("apple", "banana", "cherry")
i = 0
while i < len(thistuple):
 print(thistuple[i])
 i = i + 1

Incremento Indice →

```
apple
banana
cherry
```

TUPLE – unire tuples

OPERATORE +

Per unire due o più tuples, si può usare l'operatore +

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

tuple3 = tuple1 + tuple2
print(tuple3)
```

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

MOLTIPLICARE UNA TUPLE

Per moltiplicare il contenuto di una tuple di un dato numero di volte, si può usare l'operatore *

```
fruits = ("apple", "banana", "cherry")
mytuple = fruits * 2

print(mytuple)
```

```
('apple', 'banana', 'cherry', 'apple', 'banana', 'cherry')
```

TUPLE METODI BUILT-IN

Python ha due metodi built-in che possono essere usati con le tuple:

- **count()** : ritorna il **numero di volte** in cui uno specifico elemento è presente all'interno di una tuple

```
# tuple of vowels
vowels = ('a', 'e', 'i', 'o', 'i', 'u')

# counts the number of i's in the tuple
count = vowels.count('i')

print(count)

# Output: 2
```

- **index()** : ricerca uno specifico valore all'interno della tuple e ne ritorna **la posizione** quando lo trova.

```
# tuple containing vowels
vowels = ('a', 'e', 'i', 'o', 'u')

# index of 'e' in vowels
index = vowels.index('e')

print(index)

# Output: 1
```

PYTHON SETS

```
Myset = {"apple", "banana", "cherry"}
```

I set sono usati per «immagazzinare» diversi elementi all'interno di una variabile.

Insieme a List, Tuple e Dictionary, fanno parte dei tipi di dati built-in di Python per le collezioni di dati.

Un set è una collezione:

- **IMMUTABILE**: una volta creato il set, gli elementi non possono essere modificati (ma possono essere rimossi o aggiunti nuovi elementi),
- **NON ORDINATA**: gli elementi di un set non hanno un ordine definito (possono apparire in un ordine differente ogni volta che vengono utilizzati) e non possono essere riferiti tramite indice,
- **NON INDICIZZATA**,
- gli elementi all'interno di un set sono **UNICI** (non sono ammessi elementi duplicati). Eventuali elementi con valori uguali sono ignorati.

Gli elementi di un set sono racchiusi tra parentesi graffe.

PYTHON SETS

CREARE UN SET

```
thisset = {"apple", "banana", "cherry"}  
print(thisset)  
  
# Note: il set NON E' ORDINATO, questo significa: gli elementi appariranno in un ordine casuale.
```

I sets di Python **non sono ordinati**, questo significa che l'ordine degli elementi è casuale. Se si lancia un nuovo print(), il risultato ritornato sarà differente:

```
{'cherry', 'apple', 'banana'}
```

```
{'apple', 'banana', 'cherry'}
```

```
{'banana', 'cherry', 'apple'}
```

IL COSTRUTTORE SET()

Per creare un set si può usare anche il costruttore set()

```
#Utilizzo del costruttore set() per creare un set:  
thisset = set(("apple", "banana", "cherry")) # note the double round-brackets  
print(thisset)
```

!!!!ATTENZIONE!!!

Anche i **dictionaries** di Python utilizzano le parentesi graffe, ma i suoi elementi sono coppie chiave-valore.

Per definire un set vuoto

NON si può utilizzare la sintassi `empty_set = {}` perchè definisce un dictionary vuoto.

Bisogna invece usare **la funzione built-in `set()`**:

```
empty_set = set()
```

Un set vuoto viene considerato False in termini Booleani. Ad esempio:

```
skills = set()  
  
if not skills:  
    print('Empty sets are falsy') #returns Empty sets are falsy
```

PYTHON SET

LUNGHEZZA DI UN SET

Si usa la funzione len() per determinare quanti elementi contiene il set.

```
thisset = {"apple", "banana", "cherry"}  
  
print(len(thisset))  
#returns 3
```

type()

Per Python i sets sono definiti come oggetti di tipo «set»:

```
myset = {"apple", "banana", "cherry"}  
  
print(type(myset))  
#returns <class 'set'>
```

ELEMENTI DEL SET – TIPI DI DATI

Gli elementi di un set possono essere di qualsiasi tipo:

```
set1 = {"apple", "banana", "cherry"}  
set2 = {1, 5, 7, 9, 3}  
set3 = {True, False, False}  
  
print(set1) #{'cherry', 'apple', 'banana'}  
print(set2) #{1, 3, 5, 7, 9}  
print(set3) #{False, True}
```

Un set può contenere tipi di dati diversi:

```
set1 = {"abc", 34, True, 40, "male"}  
  
print(set1)  
#{True, 34, 40, 'male', 'abc'}
```

Possiamo anche passare un oggetto iterabile alla funzione set per creare un set, ad esempio si può passare una list

```
skills = set(['Problem solving','Critical Thinking'])  
print(skills)  
#returns {'Critical Thinking', 'Problem solving'}
```

Ovviamente, essendo il set, un insieme non ordinato, l'ordine della list di origine non può essere mantenuto.

PYTHON SET – ACCEDERE GLI ELEMENTI

Non si possono accedere gli elementi del set facendo riferimento agli indici o alle chiavi.

Si può però:

1. ciclare gli elementi del set con un **ciclo for**

2. chiedere **se un preciso valore è presente in un set** usando la keyword **in**

```
thisset = {"apple", "banana", "cherry"}  
  
for x in thisset:  
    print(x)
```

cherry
banana
apple

```
thisset = {"apple", "banana", "cherry"}  
  
print("banana" in thisset)
```

True

PYTHON SET – Aggiungere elementi

AGGIUNGERE ELEMENTI – add()

Una volta che il set è stato creato, non si possono modificare gli elementi ma se ne possono aggiungere di nuovi.

Per farlo si usa il metodo add():

`set.add(element)`

AGGIUNGERE SETS – update()

Per aggiungere elementi da un set al set corrente, si usa il metodo update()

`Set.update(set_to_add)`

AGGIUNGERE UN QUALSIASI OGGETTO ITERABILE

L'oggetto che si vuole aggiungere al set tramite il metodo update(), non deve per forza essere un set, può essere un **qualsiasi oggetto iterabile** (tuple, list, dictionary, ecc...)

```
thisset = {"apple", "banana", "cherry"}  
  
thisset.add("orange")  
  
print(thisset)  
#return {'banana', 'apple', 'cherry', 'orange'}
```

```
#aggiungiamo elementi da tropica a thisset  
thisset = {"apple", "banana", "cherry"}  
tropical = {"pineapple", "mango", "papaya"}  
  
thisset.update(tropical)  
  
print(thisset)  
  
#return {'banana', 'cherry', 'apple', 'mango', 'pineapple','papaya'}
```

```
thisset = {"apple", "banana", "cherry"}  
mylist = ["kiwi", "orange"]  
  
thisset.update(mylist)  
  
print(thisset)  
  
#return {'banana', 'cherry', 'apple', 'orange', 'kiwi'}
```

PYTHON SET – RIMUOVERE ELEMENTI DAL SET – remove e discard

Per rimuovere un elemento da un set, si possono usare i metodi `remove()` o `discard()`

REMOVE()

Rimuove l'oggetto specificato dal set

`Set.remove(item)`

```
thisset = {"apple", "banana", "cherry"}  
  
thisset.remove("banana")  
  
print(thisset)  
#returns {'cherry', 'apple'}
```

Se l'elemento da rimuovere non esiste, il metodo `remove()` lancerà una **Key exception**.

```
thisset = {"apple", "banana", "cherry"}  
  
thisset.remove("lemon")  
  
print(thisset)  
  
Traceback (most recent call last):  
  File "./prog.py", line 3, in <module>  
KeyError: 'lemon'
```

DISCARD()

Rimuove l'oggetto specificato dal set

`Set.discard(item)`

```
thisset = {"apple", "banana", "cherry"}  
  
thisset.discard("banana")  
  
print(thisset)  
  
#returns {'cherry', 'apple'}
```

Se l'elemento da rimuovere non esiste, il set originale verrà lasciato inalterato e **non sarà lanciata alcuna eccezione**.

```
thisset = {"apple", "banana", "cherry"}  
  
thisset.discard("lemon")  
  
print(thisset)  
  
{'banana', 'cherry', 'apple'}
```

PYTHON SET – RIMUOVERE ELEMENTI DAL SET – pop, clear e del

pop()

Il metodo `pop()` **rimuoverà l'ultimo elemento**. Essendo però i set non ordinati, **non si saprà quale elemento verrà eliminato**.

Il valore di ritorno del metodo `pop()` è l'elemento eliminato.

Set.pop()

```
thisset = {"apple", "banana", "cherry"}  
  
x = thisset.pop()  
  
print(x) #removed item  
  
print(thisset) #the set after removal
```

```
apple  
{'banana', 'cherry'}
```

del()

La keyword `del` **cancellerà il set completamente**

```
thisset = {"apple", "banana", "cherry"}  
  
del thisset  
  
print(thisset) #this will raise an error because the set no longer exists
```

Il `print()` solleverà un errore perché il set non esiste più

```
Traceback (most recent call last):  
  File "demo_set_del.py", line 5, in <module>  
    print(thisset) #this will raise an error because the set no longer exists  
NameError: name 'thisset' is not defined
```

clear()

Il metodo `clear()` **svuota il set**.

Set.clear()

```
thisset = {"apple", "banana", "cherry"}  
  
thisset.clear()  
  
print(thisset)
```

```
set()
```

PYTHON SET – CICLARE I SETS – ciclo for

Si può ciclare all'interno di un set usando un ciclo loop

ESEMPIO 1:

```
thisset = {"apple", "banana", "cherry"}  
  
for x in thisset:  
    print(x)
```

```
apple  
banana  
cherry
```

ESEMPIO 2:

```
for letter in set("apple"):  
    print(letter)
```

```
e  
p  
l  
a
```

Come si può notare, la «p» appare solo una volta, in quanto i set non ammettono elementi duplicati, e l'ordine degli elementi è casuale (insieme non ordinato)

PYTHON SETS –UNIRE DUE o PIU' SETS – union e update

union()

Il metodo `union()` ritorna un nuovo set contenente tutti gli elementi DISTINTI di entrambi i set (gli elementi uguali non vengono duplicati) che si vogliono unire

A.`union(other_sets)`

Il metodo `union()` accetta 0 o più argomenti, se l'argomento non è passato, ritornerà una copia del set.

```
set1 = {"a", "b", "c"}  
set2 = {1, 2, 3}  
  
set3 = set1.union(set2)  
print(set3)
```

```
{3, 1, 2, 'a', 'c', 'b'}
```

```
A = {'a', 'c', 'd'}  
B = {'c', 'd', 2 }  
C = {1, 2, 3}  
  
print('A U B =', A.union(B))  
print('B U C =', B.union(C))  
print('A U B U C =', A.union(B, C))  
  
print('A.union() =', A.union())
```

```
A U B = {2, 'a', 'd', 'c'}  
B U C = {1, 2, 3, 'd', 'c'}  
A U B U C = {1, 2, 3, 'a', 'd', 'c'}  
A.union() = {'a', 'd', 'c'}
```

Si può ottenere l'unione di sets, anche usando l'operatore |

```
print('A U B U C =', A | B | C)
```

update()

Il metodo `update()` inserisce tutti gli elementi di un oggetto iterabile all'interno di un set.

Il set iniziale viene aggiornato dopo che vengono inseriti al suo interno gli elementi dell'altro set (o oggetto iterabile).

A.`update(B)`

```
set1 = {"a", "b", "c"}  
set2 = {1, 2, 3}  
  
set1.update(set2)  
print(set1)
```

```
{2, 'b', 'a', 3, 1, 'c'}
```

Il metodo `update()` accetta qualsiasi numero di argomenti. Ad esempio, `A.update(B, C, D)`

Dove B, C e D sono iterabili i cui elementi vengo aggiunti ad A.

Sia `union()` che `update()` escludono qualsiasi elemento duplicato

PYTHON SETS –UNIRE DUE o PIU' SETS –

intersection_update e intersection

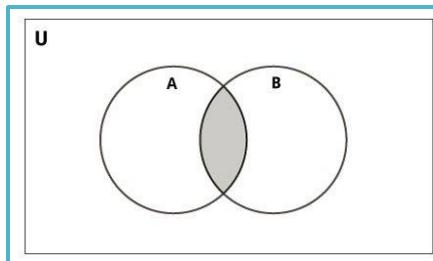
intersection_update()

Il metodo intersection_update() trova le intersezioni (elementi in comune) tra due o più set ed aggiorna il set che chiama il metodo.

A. **intersection_update(set/sets)**

Il metodo accetta uno o più argomenti

```
x = {"apple", "banana", "cherry"}  
y = {"google", "microsoft", "apple"}  
  
x.intersection_update(y)  
  
print(x)                                {'apple'}
```



intersection()

il metodo ritorna un NUOVO SET che contiene solo gli elementi presenti in tutti i set (intersezione dei sets)

X = A.**intersection(set/sets)**

Il metodo accetta uno o più argomenti

```
x = {"apple", "banana", "cherry"}  
y = {"google", "microsoft", "apple"}  
  
z = x.intersection(y)  
  
print(z)                                {'apple'}
```

Quindi, la differenza tra i due metodi è che intersection_update() aggiorna il set su cui viene chiamato il metodo, mentre intersection() crea un nuovo set.

PYTHON SETS –UNIRE DUE o PIU' SETS – differenza simmetrica

SYMMETRIC DIFFERENCE UPDATE()

Il metodo trova la differenza simmetrica(tutti gli elementi non comuni) di due set ed aggiorna il contenuto del set sul quale è stato chiamato.

A.symmetric_difference_update(B)

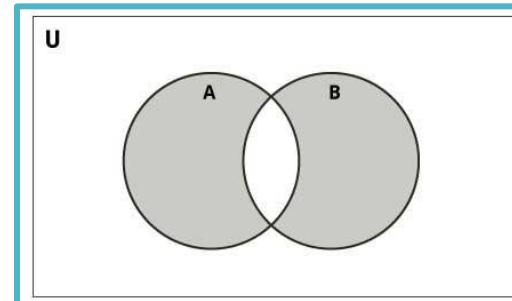
Accetta un solo parametro.

```
A = {'a', 'c', 'd'}
B = {'c', 'd', 'e' }

# updates A with the symmetric difference of A and B
A.symmetric_difference_update(B)

print(A)
# Output: {a, e}
```

Mantenere tutti gli elementi tranne quelli presenti in entrambi (o più) i set che si uniscono



SYMMETRIC DIFFERENCE()

Il metodo ritorna tutti gli elementi presenti nei sets dati, con l'eccezione degli elementi comuni (elementi della loro intersezione).

Result = A.symmetric_difference(B)

```
A = {'a', 'b', 'c', 'd'}
B = {'c', 'd', 'e' }

# returns all items to result variable except the items on intersection
result = A.symmetric_difference(B)
print(result)

# Output: {'a', 'b', 'e'}
```

PYTHON SETS – difference e difference_update

difference_update()

Il metodo `difference_update` elimina dal set **che chiama il metodo, gli elementi che esistono nei due sets(gli elementi dell'intersezione)**. In pratica calcola la differenza tra A e B ed aggiorna il set dal quale viene chiamato, con il risultato.

A.`difference_update(B)`

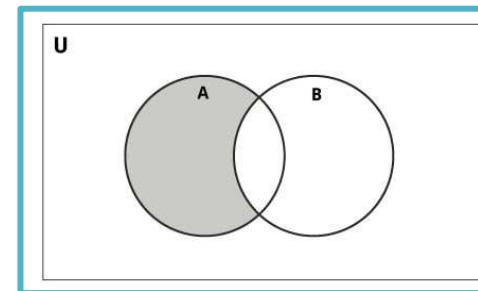
Il metodo accetta un solo argomento

```
# sets of numbers
A = {1, 3, 5, 7, 9}
B = {2, 3, 5, 7, 11}

# computes A - B and updates A with the resulting set
A.difference_update(B)

print('A = ', A)

# Output: A =  {1, 9}
```



Quindi, la **differenza** tra i due metodi è che
`intersection_update()` **aggiorna il set su cui viene chiamato il metodo**, mentre
`intersection()` **crea un nuovo set.**

difference()

il metodo **ritorna** la differenza tra due set. Contiene cioè gli elementi che esistono SOLO E SOLTANTO nel primo set, non in entrambi.

X = A.`difference(B)`

Il metodo accetta un solo argomento.

```
# sets of numbers
A = {1, 3, 5, 7, 9}
B = {2, 3, 5, 7, 11}

# returns items present only in set A
print(A.difference(B))

# Output:  {1, 9}
```

PYTHON SETS – altri metodi

isdisjoint()

Il metodo `isdisjoint` ritorna `True` se **due sets non hanno in comune alcun elemento**. Sono cioè **disgiunti**. Altrimenti ritorna `False`.

A.`isdisjoint`(B)

Il metodo accetta un solo argomento. Si può passare come parametro qualsiasi tipo di iterabile come list, tuple, dictionary o stringhe. Il metodo **prima convertirà l'iterabile in set e poi controllerà se i due set sono disgiunti o no.**

```
A = {1, 2, 3}
B = {4, 5, 6}
C = {6, 7, 8} .

print('A and B are disjoint:', A.isdisjoint(B))
print('B and C are disjoint:', B.isdisjoint(C))
```

```
A and B are disjoint: True
B and C are disjoint: False
```

issubset()

il metodo **ritorna** `True` se A è un sottoinsieme di B: tutti gli elementi del set A sono presenti in B. Altrimenti ritorna `False`.

X = A.`issubset`(B)

Il metodo accetta un solo argomento.

```
A = {1, 2, 3}
B = {1, 2, 3, 4, 5}

# all items of A are present in B
print(A.issubset(B))

# Output: True
```

PYTHON SETS – altri metodi

issuperset()

Il metodo `issuperset()` ritorna `True` se **un set possiede tutti gli elementi di un altro set (passato come argomento)**. Altrimenti ritorna `False`.

X è superset di Y se tutti gli elementi di Y sono in X

A.issuperset(B)

Il metodo controlla se A è superset di B.

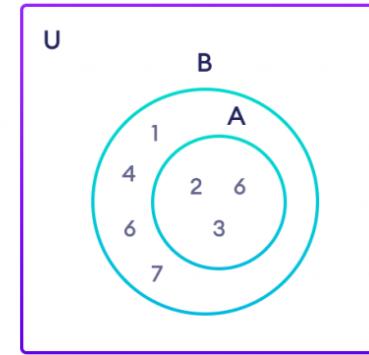
```
A = {1, 2, 3, 4, 5}
B = {1, 2, 3}
C = {1, 2, 3}

# Returns True
print(A.issuperset(B))

# Returns False
print(B.issuperset(A))

# Returns True
print(C.issuperset(B))
```

True
False
True



In questo esempio B è superset di A

PYTHON SETS - METODI

METODO	DESCRIZIONE
Add()	Aggiunge un elemento al set
Clear()	Rimuove tutti gli elementi da un set
Copy	Ritorna una copia di un set
Difference()	Ritorna un set contenente la differenza tra due o più sets
Difference_update()	Rimuove tutti gli elementi in un set che sono inclusi in un altro set specificato
Discard()	Rimuove l'elemento specificato
Intersection()	Ritorna un set che è l'intersezione di altri due sets
Intesection_update()	Nel set che lo chiama rimuove gli elementi che non sono presenti in un altro set specificato
Isdisjoint()	Controlla se due set hanno intersezioni o no
Issubset()	Controlla se un set contiene l'altro o no
Pop()	Rimuove un elemento da un set
Remove()	Rimuove l'elemento specificato
Symmetric_difference()	Ritorna un set con le differenze simmetriche tra il set che lo chiama e un altro
Union()	Ritorna un set contenente l'unione dei due set
Update()	Aggiorna il set con l'unione del set attuale e un altro

PYTHON DICTIONARIES

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

I dictionaries sono usati per memorizzare dati sotto forma di coppie chiave – valore.

Un dictionary è una collezione di dati :

- **ORDINATI** (a partire da Python 3.7): hanno un ordine definito e quest'ordine non cambia,
- **MUTABILI**: gli elementi possono essere modificati, eliminati o aggiunti anche dopo la creazione del dictionary
- **NON AMMETTE DUPLICATI**: un dictionary non può avere due elementi con la stessa chiave.

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964,  
    "year": 2020  
}  
print(thisdict)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 2020}
```

Gli **elementi** di un dictionary sono:

- scritti tra **parentesi graffe**
- sono presentati come **coppie chiave:valore**
- possono essere **riferiti usando la loro chiave**.

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
print(thisdict["brand"]) #Ford
```

PYTHON DICTIONARIES – CREARE UN DICTIONARY

Per creare un dictionary basta inserire i suoi elementi tra **parentesi graffe, separati da virgole**.

Ogni elemento ha una **CHIAVE** ed un corrispondente **VALORE** espressi come una **coppia chiave:valore**.

Mentre i **valori** possono essere di qualsiasi tipo e possono ripetersi, le **chiavi** devono essere di un tipo immutabile (string, number o tuple con elementi immutabili) e devono essere **uniche**.

METODO BUILT-IN dict()

Per definire un dictionary si può anche utilizzare il metodo built-in dict().

```
result2=dict(a=1,b=2)
```

oppure

```
result2=dict({a:1,b:2})
```

DEFINIRE UN DICTIONARY VUOTO

```
Empty_dict = {}
```

Soltanamente si definisce un dictionary vuoto prima di un ciclo e lo si popola all'interno del ciclo.

DEFINIZIONE DI UN DICTIONARY CON COPPIE CHIAVE-VALORE

```
person = {  
    'first_name': 'John',  
    'last_name': 'Doe',  
    'age': 25,  
    'favorite_colors': ['blue', 'green'],  
    'active': True  
}
```

DEFINIZIONE DI UN DICTIONARY CON IL METODO DICT()

```
My_dict = dict ({  
    1: 'apple',  
    2: 'ball'  
})
```

PYTHON DICTIONARIES - creare un dictionary – fromkeys()

METODO fromkeys()

Crea un dictionary a partire da una data sequenza di chiavi e un valore.

Dict.*fromkeys*(*keys*, *value*)

PARAMETRI:

- KEYS: sono le chiavi. Possono essere **qualsiasi iterabile** come string, set, list, ecc...
- VALUE (optional): è il valore. Può essere di **qualsiasi tipo o qualsiasi iterabile**.

Lo stesso valore è assegnato a tutte le chiavi del dictionary.

```
dict3 = dict.fromkeys(['d', 'e'],'ciao')
print(dict3)
```

```
# keys for the dictionary
alphabets = {'a', 'b', 'c'}

# value for the dictionary
number = 1

# creates a dictionary with keys and values
dictionary = dict.fromkeys(alphabets, number)

print(dictionary)

# Output: {'a': 1, 'c': 1, 'b': 1}
```

PYTHON DICTIONARIES – len() e type()

len()

Per determinare quanti elementi sono presenti all'interno di un dictionary, si usa la funzione len()

len(dictionaryname)

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964,  
    "year": 2020,  
}  
print(len(thisdict))  
  
#returns 3
```

Si ricorda che i dictionaries non ammettono elementi duplicati (stessa chiave) quindi, in caso questi siano presenti, ne viene conteggiato solo uno.

type()

Per Python i dictionaries sono oggetti di tipo «dict»

<class 'dict'>

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
print(type(thisdict))  
  
#returns <class 'dict'>
```

PYTHON DICTIONARIES – ACCEDERE GLI ELEMENTI

ACCEDERE AGLI ELEMENTI

__ se non esiste RITORNA ECCEZIONE

Si possono accedere agli elementi di un dictionary **riferendosi alla loro chiave**, inserita all'interno di parentesi quadre.

```
dictionary["key"]
thisdict = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}
x = thisdict["model"]
print(x)

#return Mustang
```

Con questo metodo, se la chiave non viene trovata, viene sollevata un **exception (KeyError)**.

Con il metodo **get()**, se la chiave non viene trovata e non è stato specificato il parametro «**value**», verrà ritornato il **valore di default «none»**

METODO **get()**

_ se non esiste NON RITORNA ECCEZIONE

Ritorna il **valore di una specifica chiave**, se la chiave è presente nel dictionary

Dictionaryname.get(key[, value])

Il metodo **get()** accetta al massimo due parametri:
Key: chiave da cercare nel dictionary
value (optional): valore da ritornare se la chiave non viene trovata. **Default = None**

```
person = {'name': 'Phill', 'age': 22}

print('Name: ', person.get('name')) #returns Name: Phillip

print('Age: ', person.get('age')) #returns Age: 22

# value is not provided
print('Salary: ', person.get('salary')) #returns Salary: None

# value is provided
print('Salary: ', person.get('salary', 0.0)) #returns Salary: 0.0
```

PYTHON DICTIONARIES – get the keys - il metodo keys()

Il **metodo keys()** ritorna la lista di tutte le chiavi del dictionary.

Dict.keys()

La lista delle chiavi è una vista del dictionary, questo vuol dire che **ogni cambiamento al dictionary, di riflesso, modificherà la keys list.**

Il metodo ritorna un oggetto view:

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

View object Keys list

```
car = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
  
x = car.keys()  
  
print(x) #before the change  
  
#returns dict_keys(['brand', 'model', 'year'])  
  
car["color"] = "white"  
  
print(x) #after the change  
  
#returns dict_keys(['brand', 'model', 'year', 'color'])
```

PYTHON DICTIONARIES – get the values - il metodo values()

Il **metodo values()** ritorna la lista di tutti i valori del dictionary.

Dict.values()

La lista dei valori è una vista del dictionary, questo vuol dire che **ogni cambiamento al dictionary, di riflesso, modificherà la values list.**

Il metodo ritorna un oggetto view:

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

View object Values list

```
# ESEMPIO 1 - Modifiche ad elementi già presenti nel dictionary
car = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}

x = car.values()

print(x) #before the change
#returns dict_values(['Ford', 'Mustang', 1964])

car["year"] = 2020

print(x) #after the change
#returns dict_values(['Ford', 'Mustang', 2020])
```

```
#ESEMPIO 2 - Aggiunta di un nuovo elemento
car = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}

x = car.values() .

print(x) #before the change
#returns dict_values(['Ford', 'Mustang', 1964])

car["color"] = "red"

print(x) #after the change
#returns dict_values(['Ford', 'Mustang', 1964, 'red'])
```

PYTHON DICTIONARIES – get items- il metodo items()

```
#ESEMPIO 1 - Modifiche al dictionary
```

```
car = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
  
x = car.items()  
  
print(x) #before the change  
#returns dict_items([('brand', 'Ford'), ('model', 'Mustang'), ('year', 1964)])  
  
car["year"] = 2020  
  
print(x) #after the change  
#returns dict_items([('brand', 'Ford'), ('model', 'Mustang'), ('year', 2020)])
```

```
#ESEMPIO 2 - aggiunta di elemento
```

```
car = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
  
x = car.items()  
  
print(x) #before the change  
#returns dict_items([('brand', 'Ford'), ('model', 'Mustang'), ('year', 1964)])  
  
car["color"] = "red"  
  
print(x) #after the change  
#return dict_items([('brand', 'Ford'), ('model', 'Mustang'), ('year', 1964)])
```

Il metodo **items()** ritorna ogni elemento del dictionary, come tuple in una list.

Dict.**items()**

La lista degli elementi è una vista del dictionary, questo vuol dire che **ogni cambiamento al dictionary, di riflesso, modificherà la lista degli elementi.**

```
dict = dict(nome='mauro', cognome='casadei')  
# caso 1  
for n, v in dict.items():  
    print(n, ':', v)  
# caso 2  
for v in dict:  
    print(v, ':', dict[v])  
  
# stampa  
# nome : mauro  
# cognome : casadei  
# nome : mauro  
# cognome : casadei
```

PYTHON DICTIONARIES – keyword in e modifica elementi

KEYWORD IN

verificare la presenza di una chiave

Per verificare se una specifica chiave è presente in un dictionary, si usa la keyword **in**

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
if "model" in thisdict:  
    print("Yes, 'model' is one of the keys in the thisdict dictionary")  
  
#returns    Yes, 'model' is one of the keys in the thisdict dictionary
```

MODIFICARE I VALORI

Si può modificare il valore di uno specifico elemento riferendosi al nome della sua chiave.

Dict [key] = newValue

```
# Change the "year" to 2018:  
  
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
  
thisdict["year"] = 2018  
  
print(thisdict)  
  
#returns    {'brand': 'Ford', 'model': 'Mustang', 'year': 2018}
```

PYTHON DICTIONARIES – modificare elementi - update()

METODO update()

Dict.**update**([items])

Il metodo update aggiorerà il dictionary con gli elementi passati come argomento.

L'argomento deve essere un dictionary o un oggetto iterabile con una coppia chiave:valore (generalmente tuples).

```
d = {1: "one", 2: "three"}  
d1 = {2: "two"}  
  
# updates the value of key 2  
d.update(d1)  
  
print(d)  
  
#returns {1: 'one', 2: 'two'}  
  
d1 = {3: "three"}  
  
# adds element with key 3  
d.update(d1)  
  
print(d)  
#returns {1: 'one', 2: 'two', 3: 'three'}
```

Il metodo update aggiunge elementi al dictionary se la chiave non è già presente. Se, invece, la chiave è già presente verrà aggiornato il suo valore.

ESEMPIO CON TUPLE PASSATA COME ARGOMENTO

```
#ESEMPIO CON TUPLES  
dictionary = {'x': 2}  
  
dictionary.update([('y', 3), ('z', 0)])  
  
print(dictionary)  
#returns {'x': 2, 'y': 3, 'z': 0}
```

In questo caso è stata passata una lista di tuples alla funzione update():

Il primo elemento della tuple è usato come chiave dell'elemento del dictionary, mentre il secondo elemento è usato come valore.

```
d1 = {'primo': 1, 'secondo': 2}  
d2 = dict(terzo=3, quarto=4)  
d1.update([('primo', 4), ('terzo', 3)]) # lista di tuple  
  
for k, v in d1.items():  
    print(k, ':', v)  
primo : 4  
secondo : 2  
terzo : 3
```

PYTHON DICTIONARIES – AGGIUNGERE ELEMENTI

Si può aggiungere un elemento al dictionary semplicemente usando una **nuova chiave ed assegnandole un valore**.

Dict [key] = value

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
thisdict["color"] = "red"  
print(thisdict)  
  
#returns  
# {'brand': 'Ford', 'model': 'Mustang', 'year': 1964, 'color': 'red'}
```

Se la chiave è già presente all'interno del dictionary, verrà semplicemente aggiornato il suo valore

METODO update()

Come visto per la modifica degli elementi di un dictionary, è possibile aggiungere un elemento ad un dictionary usando il **metodo update()** e passandogli come **argomento** un **dictionary o un oggetto iterabile** contenente una **coppia chiave:valore**.

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
thisdict.update({"color": "red"})  
  
print(thisdict)  
#returns  
# {'brand': 'Ford', 'model': 'Mustang', 'year': 1964, 'color': 'red'}
```

Rimuovere Elementi

Ci sono diversi metodi per rimuovere elementi da un dictionary:

pop() : rimuove
l'elemento con la chiave
specificata

popitem(): rimuove
l'ultimo elemento
inserito

Keyword «del»: rimuove
l'elemento con una
chiave specifica o
l'intero dictionary

clear (): rimuove il
contenuto del
dictionary.

PYTHON DICTIONARIES – RIMUOVERE ELEMENTI - pop() e popitem()

pop()

Il metodo `pop()` **rimuove e ritorna da un dictionary un elemento con una data chiave**

`Dict.pop(key[, default])`

Key = la chiave dell'elemento da rimuovere
default = optional, valore da ritornare se la chiave non esiste

```
# create a dictionary
marks = { 'Physics': 67, 'Chemistry': 72, 'Math': 89 }

element = marks.pop('Chemistry')

print('Popped Marks:', element)

# Output: Popped Marks: 72
```

Se si cerca di eliminare un **elemento non esistente**, si avrà un errore `KeyError`

popitem()

Ritorna e rimuove l'ultimo elemento inserito (nelle versioni precedenti alla 3.7 veniva rimosso un elemento casuale) nell'ordine **LIFO (Last In, First Out)**

`Dict.popitem()`

Non accetta alcun parametro.

```
person = {"name": "Cristina", "age": "39"}
print("Person:", person)

#aggiungiamo un elemento al dictionary person

person["city"] = "Riccione"
print("Person dopo aggiunta dell'elemento city:", person)

#eliminiamo l'ultimo elemento inserito
x = person.popitem()
#printiamo l'elemento che sarà rimosso (l'ultimo inserito)
print ("Elemento che sarà rimosso con popitem()", x)
print ("Person dopo l'eliminazione dell'ultimo elemento inserito:", person)|
```

```
Person: {'name': 'Cristina', 'age': '39'}
Person dopo aggiunta dell'elemento city: {'name': 'Cristina', 'age': '39', 'city': 'Riccione'}
Elemento che sarà rimosso con popitem() ('city', 'Riccione')
Person dopo l'eliminazione dell'ultimo elemento inserito: {'name': 'Cristina', 'age': '39'}
```

PYTHON DICTIONARIES – RIMUOVE ELEMENTI – del e clear()

KEYWORD «del»

La keyword del rimuove l'elemento con la chiave specificata

del dict[key]

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
del thisdict["model"]  
print(thisdict)  
  
#returns {'brand': 'Ford', 'year': 1964}
```

ATTENZIONE

La keyword «del» può anche eliminare un dictionary completamente.

del dict

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
del thisdict  
print(thisdict) #this will cause an error because "thisdict" no longer exists.
```

```
Traceback (most recent call last):  
  File "demo_dictionary_del3.py", line 7, in <module>  
    print(thisdict) #this will cause an error because "thisdict" no longer exists.  
NameError: name 'thisdict' is not defined
```

METODO clear()

Elimina tutti gli elementi del dictionary

Dict.clear()

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
thisdict.clear()  
print(thisdict)  
  
#returns {}
```

PYTHON DICTIONARIES – CICLARE UN DICTIONARY

CICLO FOR

Si può ciclare un dictionary usando il [ciclo for](#).

Quando si cicla un dictionary, il **valore ritornato sono le chiavi degli elementi** ma ci sono metodi per ritornare i valori.

1- STAMPARE TUTTE LE CHIAVI DI UN DICTIONARY, UNA PER UNA

For x in dictionary

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
for x in thisdict:  
    print(x)
```

brand
model
year

2- STAMPARE TUTTI I VALORI NEL DICTIONARY, UNO PER UNO

Dict[x]

```
for x in thisdict:  
    print(thisdict[x])
```

Ford
Mustang
1964

```
dict10 = dict(a= 'gatto', b= 12)  
dict11 = [v for (k, v) in dict10.items() if isinstance(v, int)]  
print(dict11) # [12]
```

PYTHON DICTIONARIES – CICLARE UN DICTIONARY

3 - METODO `values()` PER TORNARE I VALORI DI UN DICTIONARY

For `x` in `dict.values()`

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
for x in thisdict.values():  
    print(x)
```

Ford
Mustang
1964

4 - METODO `keys()` PER TORNARE LE CHIAVI DI UN DICTIONARY

```
for x in thisdict.keys():  
    print(x)
```

brand
model
year

5 – CICLARE ATTRAVERSO SIA CHIAVI CHE VALORI USANDO IL METODO `items()`

```
for x, y in thisdict.items():  
    print(x, y)
```

brand Ford
model Mustang
year 1964

PYTHON DICTIONARIES – COPIARE UN DICTIONARY

Non si può copiare un dictionary digitando

~~dict2 = dict1~~

Perché `dict1` sarebbe solo una riferimento a `dict1` e le modifiche fatte a `dict1` verrebbero automaticamente fatte a `dict2`.

Ci sono diversi modi per fare una copia:

1. METODO BUILT-IN `copy()`

dict.`copy()`

non accetta argomenti.

Ritorna una copia (shallow copy) del dictionary.

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
mydict = thisdict.copy()  
print("mydict:", mydict)
```

```
mydict: {'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
```

Quando si usa il metodo `copy()` viene creato un nuovo dictionary popolato con i riferimenti dal dictionary originale.

2. METODO BUILT-IN `dict()`

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
mydict = dict(thisdict)  
print("mydict:", mydict)
```

```
mydict: {'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
```

PYTHON DICTIONARIES – Dictionaries annidati

- Come già detto, un dictionary è una **collezione di elementi non ordinati**.
- Un dictionary annidato è **un dictionary dentro un altro dictionary**; è una collezione di dictionaries dentro un singolo dictionary.
- Non è possibile dividere (SLICING) un dictionary innestato
- Si può **espandere o restringere** il dictionary innestato secondo le proprie possibilità.
- I dictionary sono accedibili usano le chiavi

Nell'esempio a fianco, `nested_dict` è un **dictionary annidato** con i dizionari `dictA` e `dictB`. Questi ultimi sono due dizionari aventi le proprie chiavi e i propri valori.

```
Nested_dict = {  
    'dictA': {'key_1': 'value_1'},  
    'dictB': {'key_2': 'value_2'}  
}
```

PYTHON DICTIONARIES – dictionaries annidati - creare un dictionary annidato

CREARE UN DICTIONARY ANNIDATO

```
myfamily = {  
    "child1" : {  
        "name" : "Emil",  
        "year" : 2004  
    },  
    "child2" : {  
        "name" : "Tobias",  
        "year" : 2007  
    },  
    "child3" : {  
        "name" : "Linus",  
        "year" : 2011  
    }  
}  
  
print(mymyfamily)
```

In questo esempio «myfamily» è un **dictionary annidato**. I **dictionaries interni** «child1», «child2» e «child 3» sono assegnati a «people».

OPPURE:

```
child1 = {  
    "name" : "Emil",  
    "year" : 2004  
}  
child2 = {  
    "name" : "Tobias",  
    "year" : 2007  
}  
child3 = {  
    "name" : "Linus",  
    "year" : 2011  
}  
  
myfamily = {  
    "child1" : child1,  
    "child2" : child2,  
    "child3" : child3  
}  
  
print(mymyfamily)
```

PYTHON DICTIONARIES – Accedere gli elementi di un dictionary annidato

Per accedere gli elementi di un dictionary annidato, si usa la **sintassi degli indici** di Python:
[]

```
people = {1: {'name': 'John', 'age': '27', 'sex': 'Male'},  
          2: {'name': 'Marie', 'age': '22', 'sex': 'Female'}}  
  
print(people[1]['name'])  
print(people[1]['age'])  
print(people[1]['sex'])
```

```
John  
27  
Male
```

In questo esempio stampiamo il valore della chiave «name» usando
people[1]['name']

Dal dizionario interno 1.
Facciamo la stessa cosa con le chiavi «age» e «sex».

PYTHON DICTIONARIES – aggiungere elementi al dictionary annidato

AGGIUNGERE ELEMENTI AD UN DICTIONARY INNESTATO

```
people = {1: {'name': 'John', 'age': '27', 'sex': 'Male'},  
          2: {'name': 'Marie', 'age': '22', 'sex': 'Female'}}
```

people[3] = {} *creiamo un dizionario 3 vuoto all'interno del dictionary people*

```
people[3]['name'] = 'Luna'
```

Aggiungiamo le coppie
chiave-valore all'interno del
dictionary 3

```
people[3]['age'] = '24'
```

```
people[3]['sex'] = 'Female'
```

```
people[3]['married'] = 'No'
```

```
print(people[3])
```

AGGIUNGERE UN ALTRO DICTIONARY AL DICTIONARY ANNIDATO

```
people = {1: {'name': 'John', 'age': '27', 'sex': 'Male'},  
          2: {'name': 'Marie', 'age': '22', 'sex': 'Female'},  
          3: {'name': 'Luna', 'age': '24', 'sex': 'Female', 'married':  
              'No'}}}
```

```
people[4] = {'name': 'Peter', 'age': '29', 'sex': 'Male',  
            'married': 'Yes'}  
print(people[4])
```

PYTHON DICTIONARIES – eliminare elementi e dictionaries dal dictionary annidato

ELIMINARE ELEMENTI DAL DICTIONARY INNESTATO

Si usa la keyword del

```
people = {1: {'name': 'John', 'age': '27', 'sex': 'Male'},  
          2: {'name': 'Marie', 'age': '22', 'sex': 'Female'},  
          3: {'name': 'Luna', 'age': '24', 'sex': 'Female', 'married': 'No'},  
          4: {'name': 'Peter', 'age': '29', 'sex': 'Male', 'married': 'Yes'}}  
  
del people[3]['married']  
  
del people[4]['married']  
  
print(people[3])  
  
print(people[4])
```

```
{'name': 'Luna', 'age': '24', 'sex': 'Female'}  
{'name': 'Peter', 'age': '29', 'sex': 'Male'}
```

ELIMINARE UN DICTIONARY DA UN DICTIONARY ANNIDATO

Si usa la keyword del

```
people = {1: {'name': 'John', 'age': '27', 'sex': 'Male'},  
          2: {'name': 'Marie', 'age': '22', 'sex': 'Female'},  
          3: {'name': 'Luna', 'age': '24', 'sex': 'Female'},  
          4: {'name': 'Peter', 'age': '29', 'sex': 'Male'}}  
  
del people[3], people[4]  
print(people)
```

```
{1: {'name': 'John', 'age': '27', 'sex': 'Male'}, 2: {'name': 'Marie', 'age': '22', 'sex': 'Female'}}
```

In questo caso sono stati cancellati entrambi i dictionaries interni 3 e 4 usando «del».

PYTHON DICTIONARIES – iterazioni nel dictionary innestato

CICLO FOR

Usando il **ciclo for** possiamo ciclare ciascun elemento di un dictionary innestato.

PRIMO LOOP: il primo loop ritorna **tutte le chiavi del dictionary innestato**. Consiste nell'IDs «`p_id`» di ogni persona.

Usiamo questo Ids per estrarre le informazioni «`p_info`» di ogni persona.

SECONDO LOOP: cicla all'interno delle informazioni di ogni persona. Poi ritorna **tutte le chiavi** «`name`», «`age`», «`sex`» dal dictionary di ogni persona.

```
people = {1: {'Name': 'John', 'Age': '27', 'Sex': 'Male'},  
          2: {'Name': 'Marie', 'Age': '22', 'Sex': 'Female'}}  
  
for p_id, p_info in people.items():  
    print("\nPerson ID:", p_id)  
  
    for key in p_info:  
        print(key + ':', p_info[key])
```

```
Person ID: 1  
Name: John  
Age: 27  
Sex: Male
```

```
Person ID: 2  
Name: Marie  
Age: 22  
Sex: Female
```

PYTHON DICTIONARIES – Metodi

METODI	DESCRIZIONE
<code>clear()</code>	Rimuove tutti gli elementi da un dictionary
<code>copy()</code>	Ritorna una copia di un dictionary
<code>fromkeys()</code>	Ritorna un dictionary con le chiavi e i valori specificati
<code>get()</code>	Ritorna il valore della chiave specificata
<code>items()</code>	Ritorna una lista contenente una tuple per ogni coppia chiave:valore
<code>keys()</code>	Ritorna una lista contenente le chiavi del dictionary
<code>pop()</code>	Rimuove l'elemento con la chiave specificata
<code>popitem()</code>	Rimuove l'ultimo elemento (coppia chiave:valore) inserito
<code>Setdefault()</code>	Ritorna il valore di una chiave specificata. Se la chiave non esiste: inserisce la chiave con il valore specificato.
<code>update()</code>	Aggiorna il dictionary con le coppie chiave:valore specificate
<code>values()</code>	Ritorna una lista di tutti i valori del dictionary

Dictionary Comprehension - esempio

```
dict1 = dict(a=1, b=2)
```

```
# con loop
for k, v in dict1.items():
    print(k, ':', v)
```

```
#con comprehension
dict2 = {k: v for k, v in dict1.items()}
```

Python supporta le solite condizioni logiche matematiche:

NOME	NOTAZIONE
Uguale	$a == b$
Non uguale	$a != b$
Minore	$a < b$
Minore o uguale a	$a <= b$
Maggiore	$a > b$
Maggiore o uguale a	$a >= b$

Queste condizioni possono essere usate in molti modi, più comunemente con gli “if statement” e con i cicli.

PYTHON – IF STATEMENT - introduzione

```
if test expression:  
    statement(s)
```

Il programma **valuta la «test expression»**, vale a dire la condizione, ed **eseguirà gli statement(s) solo se la condizione è True.**

Se le condizioni danno risultato **False**, lo **statement(s) non è eseguito.**

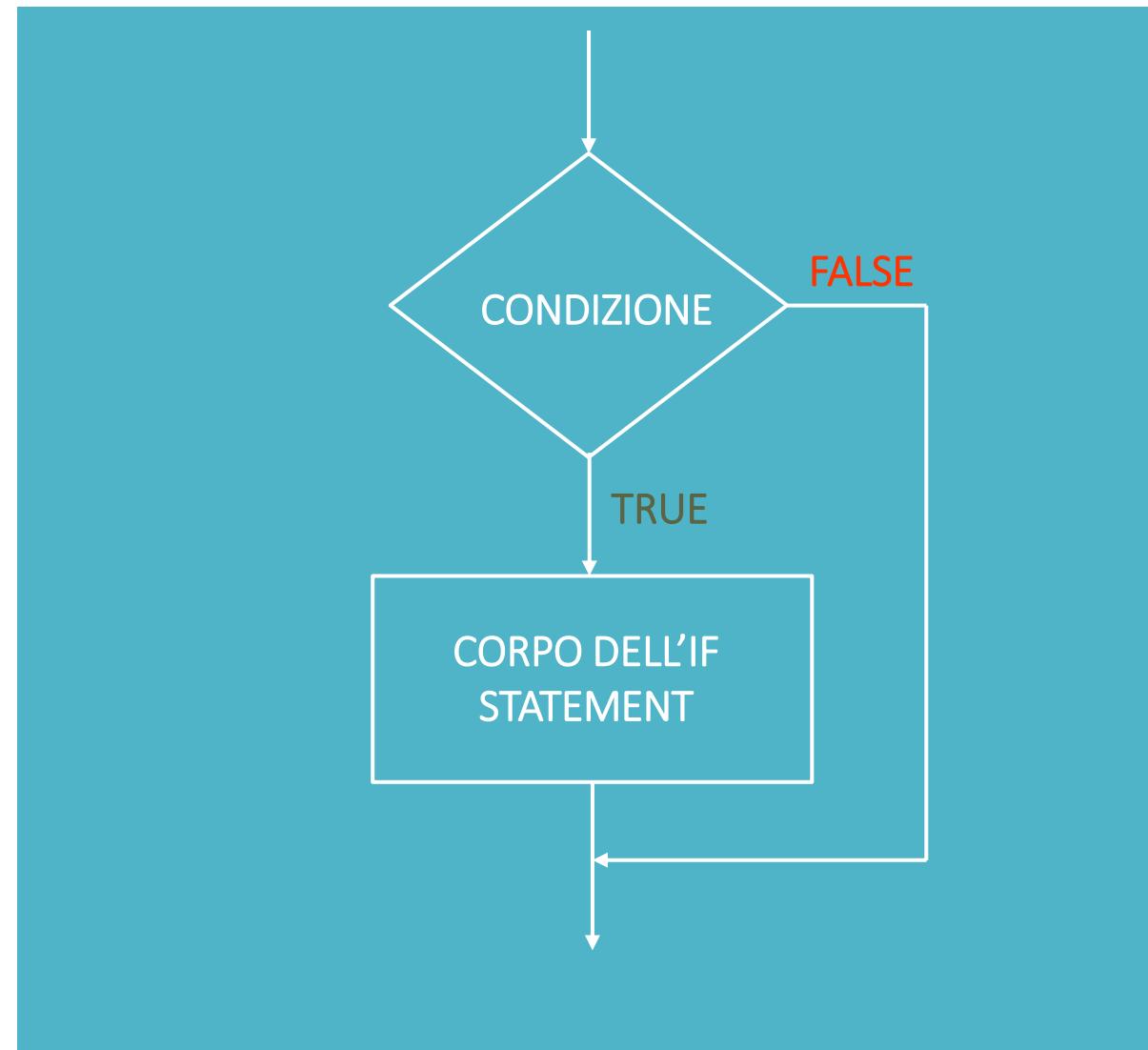
Python interpreta i valori diversi da 0 come True.

None e 0 sono interpretati come False.

In Python il body dell'if statement è indicato **dall'indentazione** (non da parentesi graffe come succede, ad esempio, in Php).

Il body inizia con un'indentazione e finisce con la prima riga non indentata.

```
a = 33  
b = 200  
if b > a:  
    print("b is greater than a")
```



PYTHON – if statement

```
# If the number is positive, we print an appropriate message
```

```
num = 3
if num > 0: → Test expression (condizione)
    print(num, "is a positive number.")
print("This is always printed.")
```

```
num = -1
if num > 0: → All'interno dell'if statement
    print(num, "is a positive number.")
print("This is also always printed.") → Fuori dall'if statement, non è
                                         indentato
```

```
3 is a positive number
This is always printed
This is also always printed.
```

Num > 0 è la **condizione da soddisfare** per procedere all'interno del body dell'if.

Infatti **il body dell'if è eseguito solo se la condizione è uguale a True.**

Quando la variabile «num» è uguale a 3, la condizione è **True** (infatti $3 > 0$ è quindi True) e gli statements all'interno del body dell'if sono eseguiti.

Quando la variabile «num» è uguale a -1, la condizione è **False** e il **body dell'if statement viene saltato**.

Il **primo print()**, che si trova all'interno del corpo dell'if (notare l'indentazione), verrà ignorato.

Il **secondo print()**, che si trova all'esterno del corpo dell'if (non è indentato), verrà sempre eseguito.

PYTHON – if statements - INDENTAZIONE

Python fa affidamento sull'indentazione per definire lo scope del codice e delle istruzioni.

Altri linguaggi di programmazione usano le parentesi graffe (curly-brackets) per questo scopo.

FARE QUINDI MOLTA ATTENZIONE A
RISPETTARE L'INDENTAZIONE, ALTRIMENTI
SI AVRANNO DEGLI ERRORI

```
a = 33
b = 200

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

Codice NON INDENTATO

```
File "demo_if_error.py", line 4
    print("b is greater than a")
               ^
IndentationError: expected an indented block
```

PYTHON – if statement – versione abbreviata

VERSIONE ABBREVIATA DELL'IF

Se il body dell'if **contiene solo un'istruzione** da eseguire in caso di condizione = True, è possibile mettere l'istruzione sulla stessa riga della condizione.

```
a = 200  
b = 33
```

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

PYTHON – if....else statement

```
if condition:  
    body of if  
else:  
    body of else
```

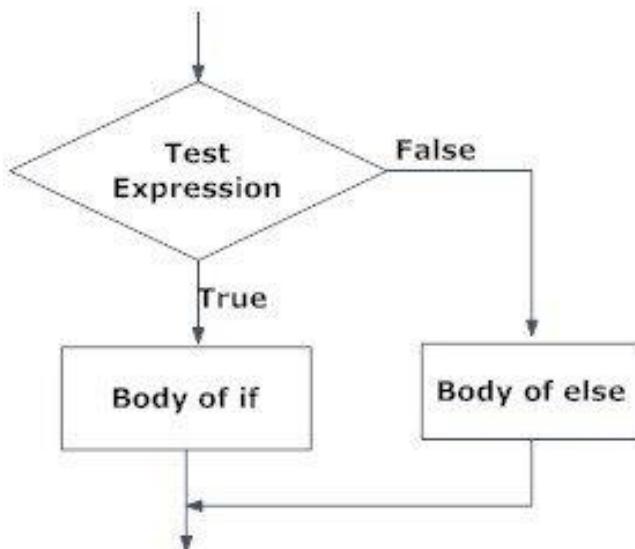


Fig: Operation of if...else statement

IF....ELSE STATEMENT

L'if...else statement valuta la condizione (test expression) ed **eseguirà il body dell'if SOLO SE la condizione è True.**

Se la condizione è **falsa**, verrà **eseguito il body dell'else.**

L'INDENTAZIONE E' USATA PER SEPARARE I BLOCCHI.

PYTHON – if...else statement – esempio

Quando `num = 3` la condizione è `True`; il `body` dell'`if` viene eseguito mentre il `body` dell'`else` viene saltato.

Quando il num = -5, la test expression è **False**. In questo caso il **body** dell'if viene saltato mentre viene eseguito quello dell'else.

Se il numeratore è uguale a 0, la test expression è **True** e verrà eseguito il **body dell'if**.

```
# Program checks if the number is positive or negative
# And displays an appropriate message

num = 3

# Try these two variations as well.
# num = -5
# num = 0

if num >= 0:
    print("Positive or Zero")
else:
    print("Negative number")
```

Positive or Zero #Con num = 3

Negative number $\#Con\ num = -5$

PYTHON – if...else statement – operatore ternario

VERSIONE ABBREVIATA DELL'IF...else

Nel caso si avessero solo un'istruzione per l'if e solo un'istruzione per l'else, si può scrivere tutto in una singola riga.

Questa tecnica è nota come **OPERATORE TERNARIO** o **ESPRESSIONE CONDIZIONALE**.

E' possibile avere anche più else statement sulla stessa riga.

```
a = 10  
x = 5 if a == 10 else 8  
print(x) # 5
```

```
a = 2  
b = 33
```

```
print("A") if a > b else print("B")
```

ISTRUZIONE IF

ISTRUZIONE ELSE

#returns B

```
a = 330  
b = 330
```

```
print("A") if a > b else print("=") if a == b else print("B")
```

Stampa A se $a > b$ altrimenti stampa = se $a == b$ altrimenti stampa B
#returns =

PYTHON – if....elif....else Statement

```
if condition:  
    body of if  
elif:  
    body of elif  
else:  
    body of else
```

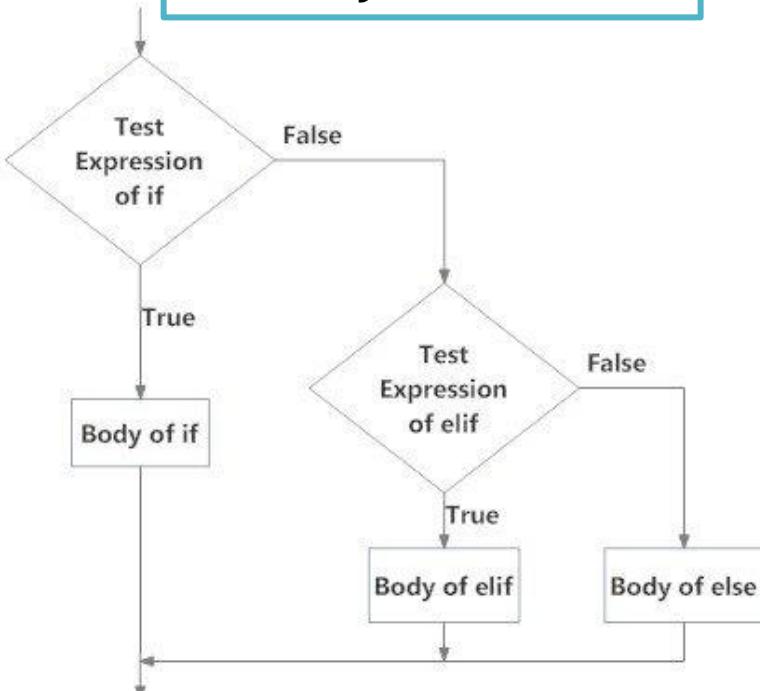


Fig: Operation of if...elif...else statement

elif = abbreviazione di else if.

Permette il controllo di più condizioni all'interno di un if statement.

1. Se la **condizione dell'if è False**, il corpo dell'if viene saltato. Viene controllata la condizione del blocco elif successivo. Se la condizione è False, si passa all'eventuale altro blocco elif successivo. Se è True si esegue il corpo dell'elif.
2. Se **tutte le condizioni sono False**, viene eseguito il body dell'else.

Solo un blocco tra i blocchi if, elif, else è eseguito.

L'if può avere **SOLO UN BLOCCO else**.
Ma può avere **PIU' BLOCCHI elif**.

```
a = 10  
if a == 9:  
    pass  
elif a == 8:  
    pass  
else:  
    print('altro')
```

PYTHON – if...elif...else – EXAMPLE

```
'''In this program,  
we check if the number is positive or  
negative or zero and  
display an appropriate message'''  
  
num = 3.4  
  
# Try these two variations as well:  
# num = 0  
# num = -4.5  
  
if num > 0:  
    print("Positive number")  
elif num == 0:  
    print("Zero")  
else:  
    print("Negative number")
```

- CASE 1 : num = 3.4

la variabile è maggiore di 0 quindi verrà eseguito il **body dell'if**.

- CASE 2 : num = 0

la variabile è uguale a 0.

condizione dell'if = False.

Condizione dell'elif = True

Viene eseguito il **body dell'elif**.

- CASE 3 : num = -4.5

condizione dell'if = false. Infatti $-4.5 < 0$

condizione dell'elif = false: Infatti $-4.5 \neq 0$

Viene eseguito il **body dell'else**.

PYTHON – if statements annidati

IF...ELSE ANNIDATI

Si può avere un if...else statement all'interno di un altro if...else statement. Questo è chiamato **annidamento (nesting)** in programmazione.

Qualsiasi numero di if...else statement può essere innestato dentro ad un altro.

Possono provocare molta confusione, per questo motivo andrebbero evitati a meno che non strettamente necessari.

```
'''In this program, we input a number  
check if the number is positive or  
negative or zero and display  
an appropriate message  
This time we use nested if statement'''  
  
num = float(input("Enter a number: "))  
if num >= 0:  
    if num == 0:  
        print("Zero")  
    else:  
        print("Positive number")  
else:  
    print("Negative number")
```

OUTPUT 1 – num = 5

```
Enter a number: 5  
Positive number
```

OUTPUT 2 – num = -1

```
Enter a number: -1  
Negative number
```

OUTPUT 3 – num = 0

```
Enter a number: 0  
Zero
```

PYTHON – if statement e operatori logici

AND

La keyword «**and**» è un operatore logico ed è utilizzato per combinare più condizioni di un if statement.

La condizione darà risultato vero, se e solo se tutte le condizioni sono vere

```
#Testa se a è maggiore di b, E se c is maggiore di a:  
a = 200  
b = 33  
c = 500  
if a > b and c > a:  
    print("Both conditions are True")
```

Both conditions are True

OR

La keyword «**or**» è un operatore logico ed è utilizzato per combinare più condizioni di un if statement. La condizione darà risultato vero, se almeno una delle condizioni è vera.

```
#Testa se a è maggiore di b, OR se a è maggiore di c:  
a = 200  
b = 33  
c = 500  
if a > b or a > c:  
    print("Almeno una delle condizioni è vera")
```

Almeno una delle condizioni è vera

PYTHON – istruzione «pass»

In Python, l'istruzione «pass» è un'istruzione nulla. La differenza tra un commento e l'istruzione «pass» è che, mentre l'interprete ignora un commento interamente, il «pass» non è ignorato.

Anche se non accade niente quando il pass è eseguito. Risulta una no operation (NOP).

Viene di solito utilizzato come segnaposto per future implementazioni di funzioni, cicli, if statements, ecc...

Supponiamo di avere un ciclo che non è ancora implementato ma che vogliamo implementare in futuro. Non può avere un body vuoto perché l'interprete darebbe errore. Si usa allora l'istruzione «pass» per sostituire un body non ancora creato

```
def function(args):
    pass
```

```
class Example:
    pass
```

```
'''pass is just a placeholder for
functionality to be added later.'''
sequence = {'p', 'a', 's', 's'}
for val in sequence:
    pass
```

ISTRUZIONE «pass» e if statement

Le istruzioni dell'if non possono essere vuote. Ma se vogliamo che non abbiano contenuto (magari per una futura implementazione), dobbiamo inserire il «pass» statement per evitare di avere un errore.

```
a = 33
b = 200

if b > a:
    pass

# having an empty if statement like this,
# would raise an error without the pass statement
```

Python ha **due cicli primitivi**:

- ciclo WHILE
- ciclo FOR

CICLI IN PYTHON – CICLO WHILE

Il ciclo while in Python è utilizzato per **iterare un blocco di codice** (delle istruzioni) **finchè una determinata condizione è vera.**

Generalmente usato quando non si conoscono in anticipo il numero di iterazioni (ripetizioni).

SINTASSI:

```
While test_expression:  
    Body of while
```

Nel ciclo while:

- I. Viene **testata la test expression** (condizione)
- II. Si accede al **body del while** solo se la **condizione è vera**.
- III. Dopo un'iterazione, la **condizione viene verificata nuovamente**.
- IV. I punti precedenti sono **ripetuti finchè la condizione non dà risultato False**.

In Python il body del while è determinato dall'**INDENTAZIONE**: inizia con la prima indentazione e finisce alla prima riga non indentata.

Ogni valore diverso da 0 è interpretato da Python come True. None e 0 sono interpretati come False.

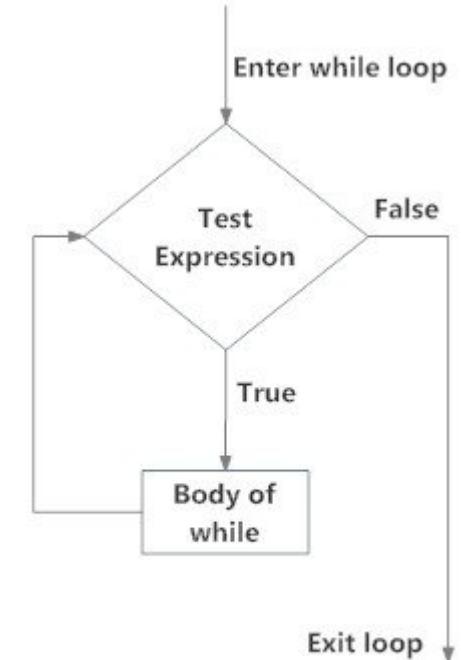


Fig: operation of while loop

CICLI IN PYTHON – CICLO WHILE - ESEMPIO

ESEMPIO 1 – stampa i finchè i è minore di 6

Ricordarsi di incrementare i,
altrimenti il ciclo
continuerà all'infinito

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

1
2
3
4
5

ESEMPIO 2 – stampa i finchè i è minore di 6

```
# Program to add natural
# numbers up to
# sum = 1+2+3+...+n

# To take input from the user,
# n = int(input("Enter n: "))

n = 10

# initialize sum and counter
sum = 0
i = 1

while i <= n:
    sum = sum + i
    i = i+1    # update counter

# print the sum
print("The sum is", sum)
```

Enter n: 10
The sum is 55

In questo esempio la test expression sarà True finche la variabile i è minore o uguale a n (10 nell'esempio).

Bisogna incrementare il valore il valore della variabile di conteggio i all'interno del ciclo.

Alla fine il risultato verrà mostrato.

CICLI PYTHON – istruzione break

In Python, le istruzioni «break» e «continue» possono alterare il flusso di un normale ciclo.

Il ciclo ripete un blocco di codice finche la condizione testata è vera. A volte però si ha la necessità di **terminare l'iterazione corrente o anche l'intero ciclo** senza verificare la condizione.

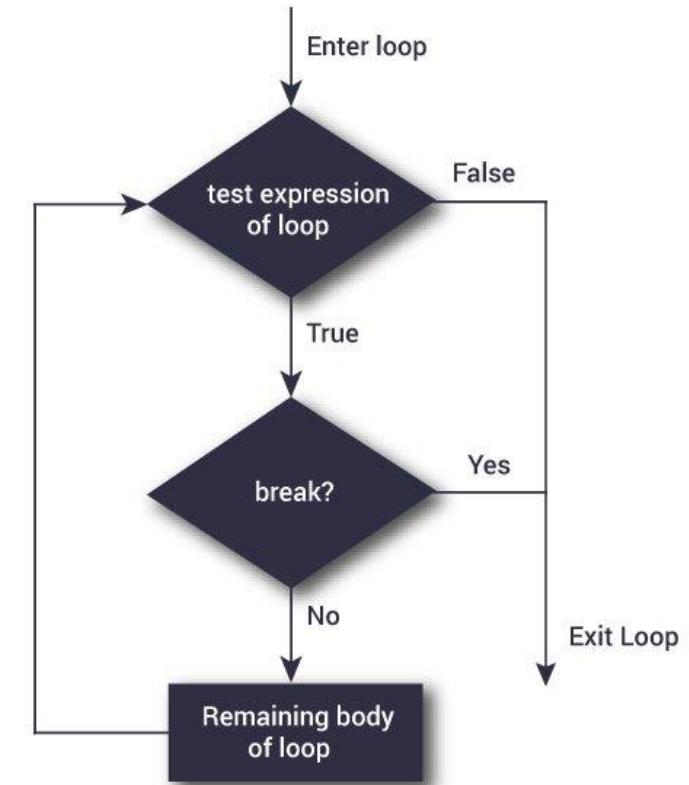
ISTRUZIONE BREAK

L'istruzione «break» **termina il ciclo che la contiene.**

Il controllo va all'istruzione immediatamente successiva al ciclo (esterna al ciclo).

Se il «break» è all'interno di un ciclo annidato, il break interromperà il ciclo interno, non quello esterno.

```
for var in sequence:  
    # codes inside for loop  
    if condition:  
        break  
        # codes inside for loop  
  
    # codes outside for loop  
  
-----  
  
while test expression:  
    # codes inside while loop  
    if condition:  
        break  
        # codes inside while loop  
  
    # codes outside while loop
```



CICLI PYTHON – istruzione break

ESEMPIO 1

```
# Use of break statement inside the loop

for val in "string":
    if val == "i":
        break
    print(val)

print("The end")
```

```
s
t
r
The end  .
```

In questo esempio si ciclano la singole lettere della parola "string".

Si controlla se la lettera è «i», lettera sulla quale interrompiamo il ciclo.

Tutte le lettere fino ad «i» sono state stampate. Arrivati alla «i», il ciclo termina.

ESEMPIO 2

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

```
1
2
3
```

In questo esempio si ciclano i numeri da 1 a 6.

Il ciclo si interromperà quando i sarà uguale a 3.

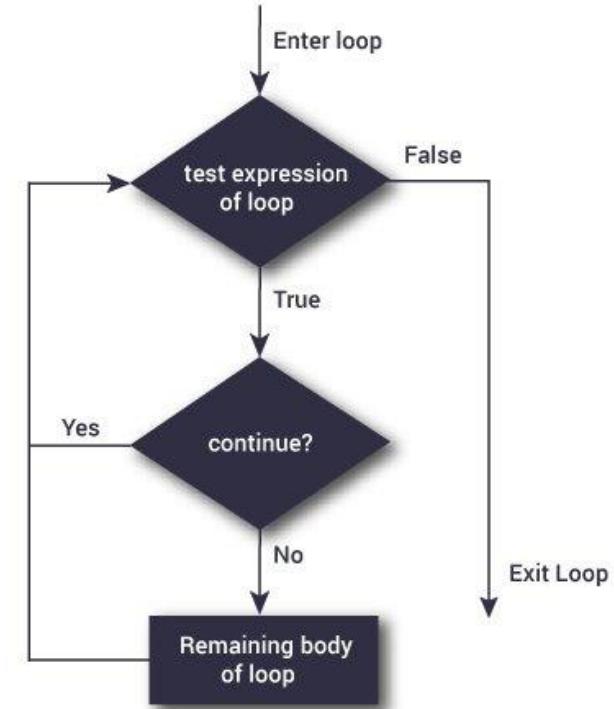
Tutti i numeri fino a «3» sono stampati. Arrivati a «3», il ciclo termina.

CICLI PYTHON – istruzione continue

ISTRUZIONE continue

L'istruzione «continue» salta il resto del codice all'interno di un'iterazione corrente. Il ciclo non termina ma continua con l'iterazione successiva.

```
for var in sequence:  
    # codes inside for loop  
    if condition:  
        continue  
    # codes inside for loop  
  
    # codes outside for loop  
  
-----  
  
while test expression:  
    # codes inside while loop  
    if condition:  
        continue  
    # codes inside while loop  
  
    # codes outside while loop
```



CICLI PYTHON - istruzione continue

```
# Program to show the use of continue statement inside loops

for val in "string":
    if val == "i":
        continue
    print(val) ←
print("The end")
```

Se val è «i» il codice dopo l'istruzione «continue» non viene eseguito e si passa all'iterazione successiva. Non si esce dal ciclo come succede con l'istruzione «break». Viene dunque bloccata solo la corrente iterazione.

```
s
t
r
n
g
The end
```

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

# Note that number 3 is missing in the result
```

```
1
2
4
5
6
```

CICLI PYTHON – else STATEMENT

L'istruzione else permette di eseguire un blocco di codice una volta che la condizione del ciclo non è più vera.

Se il ciclo viene interrotto con un «break», l'else è ignorato. Infatti l'else all'interno di un ciclo è eseguito solo se non si ha un break e se la condizione del ciclo è falsa.

```
'''Example to illustrate  
the use of else statement  
with the while loop'''  
  
counter = 0  
  
while counter < 3:  
    print("Inside loop")  
    counter = counter + 1  
else:  
    print("Inside else")
```

```
Inside loop  
Inside loop  
Inside loop  
Inside else
```

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

```
1  
2  
3  
4  
5  
i is no longer less than 6
```

PYTHON – CICLO FOR

CICLO FOR

Il ciclo for è usato per **ciclare delle sequenze** (list, tuple, string) o altri oggetti iterabili.

Con il ciclo for possiamo **eseguire una serie di istruzioni, per ciascun elemento dell'oggetto iterabile.**

SINTASSI:

```
for val in sequence:  
    body del for
```

Val è la variabile che prende il valore dell'elemento considerato dall'attuale iterazione.

Il ciclo continua finchè non viene raggiunto l'ultimo elemento della sequenza.

Il corpo del ciclo è separato dal resto del codice usando l'indentazione.

Il ciclo for **non ha bisogno di una variabile d'indice** da settare prima dell'inizio del ciclo.

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

Stampa ogni elemento (x) nella lista fruits.
PRIMA ITERAZIONE: x = apple
SECONDA ITERAZIONE: x = banana
TERZA ITERAZIONE: x = cherry

apple
banana
cherry

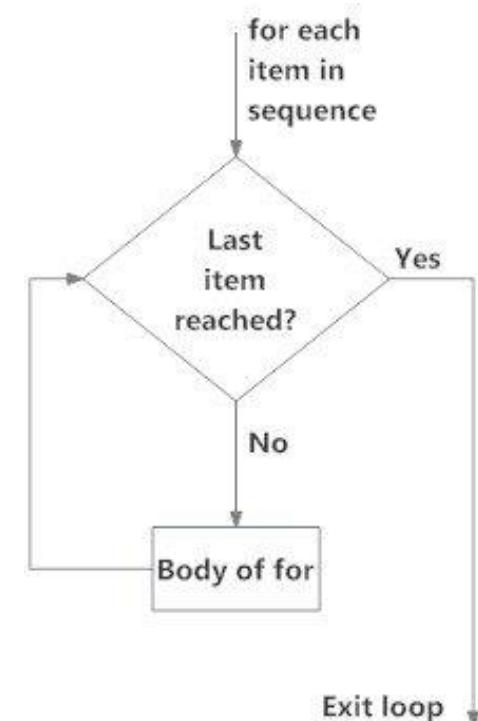


Fig: operation of for loop



CICLI PYTHON – CICLO FOR

ESEMPI DI CICLI FOR

```
# Program to find the sum of all numbers stored in a list  
  
# List of numbers  
numbers = [6, 5, 3, 8, 4, 2, 5, 4, 11]  
  
# variable to store the sum  
sum = 0  
  
# iterate over the list  
for val in numbers:  
    sum = sum+val  
  
print("The sum is", sum)
```

```
The sum is 48
```

ESEMPI DI CICLI FOR CON UNA STRINGA

Anche le stringhe sono oggetti iterabili contenenti una sequenza di caratteri.

E' dunque possibile eseguire un ciclo sulle lettere che compongono una stringa:

```
•  
b  
a  
n  
a  
n  
a  
  
for x in "banana":  
    print(x)
```

CICLI PYTHON – ciclo for - break

L'ISTRUZIONE BREAK

Con l'istruzione break possiamo interrompere il ciclo prima che abbia ciclato tutti gli elementi.

ESEMPIO 1

```
# interrompere il ciclo quando x è uguale a "banana"

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

apple
banana

ESEMPIO 2

```
# interrompe il ciclo quando x = "banana" ma questa volta
# il ciclo viene interrotto prima del print()

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

apple

L'ISTRUZIONE CONTINUE

Con l'istruzione continue possiamo fermare la corrente iterazione del ciclo ed andare alla prossima. Non esce completamente dal ciclo.

```
#il continue consente di uscire solo dall'attuale iterazione,
# non esce dall'intero ciclo.

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

apple
cherry

CICLI PYTHON – ciclo for – la funzione range()

FUNZIONE range()

Si usa per ciclare un pezzo di codice per un determinato numero di volte.

La funzione range() ritorna una sequenza di numeri che, di default, parte da 0, sempre di default, incrementa di 1 e finisce al numero specificato.

SINTASSI:

Range (**start**, **stop**, **step_size**)

Facoltativo. Valore di partenza. Se non specificato è 0.

Obbligatorio. Valore finale, escluso.

Facoltativo. Step di incremento. Se non specificato è 1.

Stampa tutti gli elementi nel range. Notare che il valore finale è escluso.

Quindi range(6) non sono i valori da 0 a 6 ma da 0 a 5.

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

0
1
2
3
4
5

```
for x in range(2, 6):  
    print(x)
```

2
3
4
5

Aggiungiamo il parametro del valore di partenza.
Stamperà tutti gli elementi da 2 a 6 (escluso)

Aggiungiamo il parametro step_size.
Stamperà tutti gli elementi da 2 a 30 (escluso) con un intervallo di 3.

```
for x in range(2, 30, 3):  
    print(x)
```

2
5
8
11
14
17
20
23
26
29

CICLI PYTHON – ciclo for – la funzione range()

L'oggetto range è «pigro» nel senso che non genera tutti i valori che «contiene» quando lo creiamo. **Non salva tutti i valori in memoria**, salva l'inizio, la fine e lo step del range e genera il numero successivo runtime.

Per forzare la funzione range() a ritornare tutti gli elementi, si usa la **funzione list()**

SINTASSI:

Range (**list(range(x))**)

```
print(range(10))
print(list(range(10)))
print(list(range(2, 8)))
print(list(range(2, 20, 3)))
```

```
range(0, 10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[2, 3, 4, 5, 6, 7]
[2, 5, 8, 11, 14, 17]
```

CICLI PYTHON – ciclo for – combinare len() e range()

COMBINARE RANGE() E LEN()

La funzione **range()** può essere combinata con la funzione **len()** per ciclare una sequenza usando gli indici.

```
# Program to iterate through a list using indexing  
  
genre = ['pop', 'rock', 'jazz']  
  
# iterate over the list using index  
for i in range(len(genre)):  
    print("I like", genre[i])
```

```
I like pop  
I like rock  
I like jazz
```

len() misura la lunghezza dell'oggetto iterabile, vale a dire il numero di elementi contenuti in quell'oggetto. Nel nostro caso misura la lunghezza della list `genre`.

$$\text{Len}(\text{genre}) = 3$$

`Range(len(genre))` crea una sequenza di numeri da 0 a 3 (escluso)

Gli elementi del range ci serviranno per accedere agli elementi della list attraverso il loro indice.

CICLI PYTHON – else nel ciclo for

KEYWORD else

La keyword else in un ciclo for specifica un **blocco di codice da eseguire quando il ciclo è finito**. Viene cioè eseguito **quando non ci sono più elementi su cui ciclare**.

```
#Stampa tutti i numeri da 0 a 5, e stampa  
#un messaggio quando il ciclo è finito:  
  
for x in range(6):  
    print(x)  
else:  
    print("Finally finished!")
```

```
0  
1  
2  
3  
4  
5  
Finally finished!
```

Il blocco else NON verrà eseguito se il ciclo è interrotto da un break.

```
for x in range(6):  
    if x == 3: break  
    print(x)  
else:  
    print("Finally finished!")
```

```
0  
1  
2
```

#Se il ciclo si interrompe con un break,
#il blocco else non sarà eseguito.

CICLI PYTHON – ciclo for – CICLI ANNIDATI

Un ciclo annidato è un **ciclo dentro un ciclo**.

Il ciclo interno sarà eseguito una volta per ogni iterazione del ciclo esterno: **per ogni iterazione del ciclo esterno, il ciclo interno eseguirà tutte le sue iterazioni.** Per ogni iterazione del ciclo esterno, il ciclo interno ricomincia e completa la sua esecuzione prima che il **ciclo esterno possa passare alla prossima iterazione.**

Il ciclo esterno e quello interno possono essere **di qualsiasi tipo**: il ciclo esterno può essere un for e contenere al suo interno un ciclo while e vice versa.

Il ciclo esterno può contenere più di un ciclo interno.

I cicli innestati sono di solito usati per lavorare con **strutture di dati multidimensionali** come, ad esempio, una lista contenente un'altra lista.

```
adj = ["red", "big", "tasty"]
fruits = ["apple", "banana", "cherry"]

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

Per ogni elemento della list adj, viene eseguito un ciclo sugli elementi della list fruits.

```
red apple
red banana
red cherry
big apple
big banana
big cherry
tasty apple
tasty banana
tasty cherry
```

CICLI PYTHON – ciclo for – istruzione pass

Il body del ciclo for non può essere vuoto.

Come visto per l'if statement, per evitare di incorrere in un errore qualora avessimo l'esigenza di lasciare vuoto il corpo del for (magari perché abbiamo intenzione di implementare il ciclo successivamente), possiamo ricorrere all'istruzione «pass».

```
for x in [0, 1, 2]:  
    pass  
  
# having an empty for loop like this,  
# would raise an error without the pass statement
```

FUNZIONI

FUNZIONI IN PYTHON

In Python una funzione è un blocco di codice definito con un nome.

Si usano le funzioni quando si ha la necessità di **eseguire la stessa operazione più volte senza riscrivere lo stesso codice.**

Una funzione può accettare argomenti e ritornare un valore.

Python, come altri linguaggi di programmazione, fa suo il principio **DRY (Don't Repeat Yourself).**

Consideriamo il caso in cui si abbia la necessità di ripetere la stessa azione o operazione molte volte.

Possiamo definire quell'azione una volta sola usando una funzione e chiamando quella funzione ogni volta che viene richiesto di eseguire la medesima attività.

Le funzioni **migliorano l'efficienza** e **riducono gli errori** grazie alla riusabilità del codice. Una volta creata una funzione, possiamo chiamarla ovunque e ogni volta che serve.

Il beneficio nell'usare le funzioni sta nel rendere riusabile e modulabile il codice

TIPI DI FUNZIONI

Python supporta due tipi di funzioni:

1. **FUNZIONI BUILT-IN** o **FUNZIONI PREDEFINITE**: sono funzioni native di Python. Alcuni esempi di funzioni built-in sono: range(), type(), id(), ecc.
2. **FUNZIONI DEFINITE DALL'UTENTE**: Sono funzioni create appositamente dai programmatore per rispondere a delle specifiche esigenze di programmazione.

FUNZIONI IN PYTHON

Python Functions

In Python, the **function is a block of code defined with a name**

- A Function is a block of code that only runs when it is called.
- You can pass data, known as parameters, into a function.
- Functions are used to perform specific actions, and they are also known as methods.
- **Why use Functions?** To reuse code: define the code once and use it many times.

```
Function Name    Parameters  
↑             ↑  
def add(num1, num2):  
    print("Number 1:", num1)  
    print("Number 2:", num1)  
    addition = num1 + num2  
  
    return addition —> Return Value  
  
res = add(2, 4) —> Function call  
print(res)
```

Function Body

PYnative

Step da seguire per creare una funzione in Python:

1. usare la **keyword def con il nome della funzione** per definire una funzione
2. Passare i **parametri** a seconda delle proprie esigenze
3. **Definire il corpo della funzione** con un blocco di codice che conterrà le azioni che vogliamo intraprendere.

In Python **non servono le parentesi graffe** per definire il corpo della funzione. La sua definizione viene affidata all'**INDENTAZIONE**.

FUNZIONI IN PYTHON – CREARE UNA FUNZIONE

```
def function_name(parameter1, parameter2):
    # function body
    # write some action
return value
```

- **Function_name**: è il nome della funzione. Possiamo dare alla funzione qualsiasi nome.
- **Parameter**: i parametri sono i **valori passati alla funzione**. Possiamo passare qualsiasi numero di parametri alla funzione, anche nessuno. La funzione usa i valori dei parametri per eseguire un'azione su di essi.
- **Function_body**: blocco di codice che esegue alcune azioni. Non è altro che l'azione che vogliamo compiere.
- **Return value**: è l'output della funzione. E' **opzionale**

ARGOMENTI

Sono le informazioni che possono essere passate alla funzione. Possono essere un valore, una variabile o un oggetto.

Sono specificati dopo il nome della funzione, tra parentesi tonde. Si possono aggiungere quanti argomenti si desideri, separandoli con una virgola.

ARGOMENTI O PARAMETRI?

I termini «argomenti» e «parametri» possono essere usati per la stessa cosa: informazioni che sono passate a una funzione.

Dal punto di vista di una funzione:

- un **PARAMETRO** è la variabile elencata tra parentesi nella **definizione di una funzione**
- un **ARGOMENTO** è il **valore inviato** alla funzione quando questa viene **chiamata**.

FUNZIONI IN PYTHON – creazione e chiamata funzione

CREAZIONE DI UNA FUNZIONE SENZA PARAMETRI

```
# function
def message():
    print("Welcome to PYnative")
```

CHIAMATA DI UNA FUNZIONE SENZA PARAMETRI

Per chiamare una funzione senza parametri basta **usare il nome della funzione seguito da parentesi tonde:**

```
# call function using its name
message()
```

OUTPUT DELLA FUNZIONE:

```
Welcome to PYnative
```

CREARE UNA FUNZIONE CON PARAMETRI

Creiamo una funzione che accetta due parametri e mostra il loro valore.

```
# function
def course_func(name, course_name):
    print("Hello", name, "Welcome to PYnative")
    print("Your course name is", course_name)

# call function
course_func('John', 'Python')
```

```
Hello John Welcome to PYnative
Your course name is Python
```

FUNZIONI PYTHON - chiamare una funzione

Una volta definita una funzione e finalizzata la sua struttura, possiamo chiamare quella funzione **usando il suo nome.**

Possiamo **anche chiamare quella funzione da un'altra funzione** oppure da un altro programma importandola.

Per chiamare una funzione, basta usare il **nome della funzione seguito da parentesi tonde** e, se la funzione accetta parametri, **passare il valore dei parametri** tra parentesi.

CHIAMATA DI FUNZIONE SENZA PARAMETRI

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

my_function()
```

Hello from a function

CHIAMATA DI FUNZIONE CON PARAMETRI

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

my_function("Emil")
my_function("Tobias")
my_function("Linus")
```

Emil Refsnes
Tobias Refsnes
Linus Refsnes

Si ha una funzione con un argomento (fname). Quando la funzione è chiamata, passiamo il nome. Questo viene usato all'interno della funzione per stampare il nome completo.

FUNZIONI PYTHON – chiamare la funzione – positional arguments

In Python esistono **4 tipi di argomenti consentiti:**

1. Positional arguments
2. Keyword arguments e **kwargs
3. Default arguments
4. Lunghezza variabile degli argomenti (*args)

POSITIONAL ARGUMENTS

Di default, una funzione deve essere chiamata con il **numero esatto di argomenti, scritti nello stesso ordine in cui compaiono nella sua definizione.**

Il primo argomento della dichiarazione della funzione, deve essere il primo argomento della chiamata. Il secondo argomento deve essere tale sia nella dichiarazione che nella chiamata e così via.

Questo significa anche che, **se la funzione si aspetta 2 argomenti, bisogna chiamare quella funzione con 2 argomenti, NON DI MENO e NON DI PIU'.**

Il numero e la posizione degli argomenti devono corrispondere.
Se si cambia l'ordine degli argomenti, potrebbe cambiare anche il risultato!!!!!!

```
def add(a, b):  
    print(a - b)  
  
add(50, 10)  
# Output 40  
add(10, 50)  
# Output -40
```

Se si provano a passare più argomenti, si avrà un errore.

```
def add(a, b):  
    print(a - b)  
  
add(105, 561, 4)
```

TypeError: add() takes 2 positional arguments but 3 were given

FUNZIONI PYTHON – chiamare la funzione – numero arbitrario di argomenti (*args)

Se non sappiamo quanti argomenti saranno passati alla funzione, possiamo aggiungere un **asterisco *** davanti al nome del parametro nella definizione della funzione.

In questo modo la funzione riceverà una tuple contenente un **numero variabile di non keyword arguments**.

```
def my_function(*kids):
    print("The youngest child is " + kids[2])

my_function("Emil", "Tobias", "Linus")
```

The youngest child is Linus

```
def adder(*num):
    sum = 0

    for n in num:
        sum = sum + n

    print("Sum:",sum)

adder(3,5)
adder(4,5,6,7)
adder(1,2,3,5,6)
```

```
Sum: 8
Sum: 22
Sum: 17
```

Nell'esempio, *num viene usato come parametro che permette di passare una lista di argomenti di lunghezza variabile alla funzione adder(). All'interno della funzione, un ciclo somma gli argomenti passati e stampa il risultato.

FUNZIONI PYTHON – chiamare la funzione – keyword arguments

KEYWORD ARGUMENTS (kwargs in Python documentations)

Un keyword argument è il **valore di un argomento passato alla funzione e preceduto dal nome della variabile e dall'uguale**.

key = value

In questo caso **l'ordine degli argomenti non importa** ma il **numero deve corrispondere**, altrimenti si avrà un errore.

Se si usano contemporaneamente positional and keyword arguments, bisogna passare come **primo argomento il positional** e poi i **keyword arguments** altrimenti si avrà un errore di sintassi.

```
def my_function(child3, child2, child1):
    print("The youngest child is " + child3)

my_function(child1 = "Emil", child2 = "Tobias", child3 = "Linus")
```

The youngest child is Linus

```
-----
```

```
def message(first_nm, last_nm):
    print("Hello..!", first_nm, last_nm)

# correct use
message("John", "Wilson")
message("John", last_nm="Wilson")

# Error
# SyntaxError: positional argument follows keyword argument
message(first_nm="John", "Wilson")
```

FUNZIONI PYTHON – chiamare la funzione – numero arbitrario di keyword arguments (**kwargs)

Se non si sa quanti keyword arguments saranno necessari nella nostra funzione, si aggiungono due asterischi ** prima del nome del parametro nella definizione della funzione.

In questo modo la funzione riceverà un dictionary di argomenti

```
def my_function(**kid):
    print("His last name is " + kid["lname"])

my_function(fname = "Tobias", lname = "Refsnes")
```

His last name is Refsnes

```
def intro(**data):
    print("\nData type of argument:",type(data))

    for key, value in data.items():
        print("{} is {}".format(key,value))

intro(F firstname="Sita", L lastname="Sharma", A ge=22, P hone=1234567890)
intro(F firstname="John", L lastname="Wood", E mail="johnwood@nomail.com",
C ountry="Wakanda", A ge=25, P hone=9876543210)
```

```
Data type of argument: <class 'dict'>
Firstname is Sita
Lastname is Sharma
Age is 22
Phone is 1234567890
```

```
Data type of argument: <class 'dict'>
Firstname is John
Lastname is Wood
Email is johnwood@nomail.com
Country is Wakanda
Age is 25
Phone is 9876543210
```

Abbiamo una funzione intro() con ** data come parametro. Abbiamo passato due dictionaries con un numero variabile di argomenti. All'interno della funzione intro() c'è un ciclo che opera sui dati del dictionary passato e ne stampa le chiavi e i valori.

FUNZIONI IN PYTHON – variabili - *args e **kwargs

- ***args** e ****kwargs** sono keyword speciali che permettono alla funzione di accettare un numero variabile di argomenti
- *args passa un numero variabile di argomenti **non-keyworded** sui quali posso essere eseguite operazioni proprie delle **tuples**
- **kwargs passa alla funzione un **dictionary** con un numero variabile di argomenti di tipo keyword sui quali eseguire operazioni proprie dei dictionaries
- * args e **kwargs rendono la funzione flessibile.

```
def somma(*args, **kwargs):
    print(args)  # (2, 3, 4, 5) tupla
    print(kwargs) # {'op1': 6, 'op2': 8} dictionary
    risultato = 0
    for o in args:
        risultato += o
    return risultato

print(somma(2, 3, 4, 5, op1=6, op2=8))
```

FUNZIONI PYTHON – chiamare la funzione – Valore di default del parametro

ARGOMENTI DI DEFAULT

Possiamo assegnare ad un parametro un **valore di default** usando l'operatore di assegnamento **=**.

Se chiamiamo una funzione senza passarle argomenti, questa userà i valori di default.

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

my_function("Sweden")
my_function("India")
my_function()
my_function("Brazil")
```

```
I am from Sweden
I am from India
I am from Norway
I am from Brazil
```

```
# function with default argument
def message(name="Guest"):
    print("Hello", name)

# calling function with argument
message("John")

# calling function without argument
message()
```

```
Hello John
Hello Guest
```

FUNZIONI IN PYTHON – variabili – passare una List come argomento

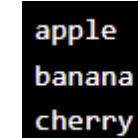
Possiamo passare qualsiasi tipo di dato ad una **funzione** (string, number, list, dictionary, ecc.) e sarà trattato come tale tipo di dato all'interno della funzione.

Ad esempio, se passiamo una List come argomento, quando raggiungerà la funzione, sarà ancora una List e trattata come tale.

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

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

my_function(fruits)
```



apple
banana
cherry

FUNZIONI IN PYTHON – ritornare valori

L'ISTRUZIONE return

Per far sì che la funzione torni un valore, si usa l'istruzione **return()**.

Permette di **uscire da una funzione** e tornare al punto in cui è stata chiamata.

Il return può contenere **un'espressione o una variabile il cui valore viene poi ritornato**. Se non c'è alcuna espressione nell'istruzione o se il return stesso non è presente all'interno della funzione, questa ritornerà un **oggetto None**.

```
def my_function(x):
    return 5 * x

print(my_function(3))
print(my_function(5))
print(my_function(9))
```

15
25
45

```
def absolute_value(num):
    """This function returns the absolute
    value of the entered number"""

    if num >= 0:
        return num
    else:
        return -num

print(absolute_value(2))

print(absolute_value(-4))
```

2
4

LE VARIABILI – Variabili Globali

VARIABILI GLOBALI

Le variabili globali sono quelle variabili che sono **create al di fuori di una funzione** e possono essere **usate da chiunque, sia dentro che fuori le funzioni.**

```
#ESEMPIO: Creiamo una variabile fuori da una funzione e la usiamo all'interno della funzione
|
x = "awesome"

def myfunc():
    print("Python is " + x)

myfunc()
```

Python is awesome

Se si crea una **variabile con lo stesso nome all'interno della funzione**, quest'ultima sarà locale e potrà essere usata solo all'interno della funzione in cui è stata dichiarata. **La variabile globale con lo stesso nome rimarrà invariata e con il valore iniziale.**

```
#Create a variable inside a function, with the same name as the global variable
x = "awesome"

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

myfunc()

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

Python is fantastic
Python is awesome

LE VARIABILI – Variabili Globali

LA KEYWORD «GLOBAL»

- Normalmente, quando si crea una variabile all'interno di una funzione, questa è locale e può essere utilizzata solo all'interno della funzione stessa.

Per **creare una variabile globale all'interno di una funzione**, si usa la parola chiave **«global»**.

```
#Se si usa la keyword "global", la variabile appartiene allo SCOPE GLOBALE

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

myfunc()

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

Python is fantastic

- **«global»** è usato anche per **modificare il valore della variabile globale all'interno di una funzione**

```
# Per modificare il valore di una variabile globale all'interno di una funzione, riferisi alla variabile usando
# la parola "global"

x = "awesome"

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

myfunc()

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

Python is fantastic

Esempio Variabili keyword global

```
mela = 1

def fun():
    global mela
    mela = 4
    print('int', mela)
```

```
fun()
print('ext', mela)
```

```
# int 4
# ext 4
```

```
mela = 1

def fun():
    mela = 4
    print('int', mela)
```

```
fun()
print('ext', mela)
```

```
# int 4
# ext 1
```

LEGB Scope

local: L'ambito locale (o funzione) è il blocco di codice o il corpo di qualsiasi funzione Python o espressione lambda

enclosing: L'ambito di inclusione (o non locale) è un ambito speciale che esiste solo per le funzioni nidificate.

global: L'ambito globale (o modulo) è l'ambito più in alto in un programma, script o modulo Python

built-in: L'ambito integrato è uno speciale ambito Python che viene creato o caricato ogni volta che si esegue uno script o si apre una sessione interattiva.

```
>>> print(list)  
<class 'list'>
```

FUNZIONI IN PYTHON - RICORSIONI

FUNZIONI RICORSIVE

Una funzione ricorsiva è una funzione che **richiama se stessa ancora e ancora.**

La ricorsione è un concetto matematico e di programmazione. Significa che una funzione chiama se stessa. Questo ha il beneficio di poter ciclare i dati per ottenere un risultato.

Gli sviluppatori dovrebbero essere **molto attenti** con le ricorsioni siccome può essere semplice incorrere nella scrittura di **una funzione che non termina mai** o una funzione che usa **un'eccessiva quantità di memoria** o di capacità di calcolo del processore.

In ogni caso, se scritta correttamente, una ricorsione può essere un efficiente e matematicamente-elegante approccio alla programmazione.

```
def tri_recursion(k):
    if(k > 0):
        result = k + tri_recursion(k - 1)
        print(result)
    else:
        result = 0
    return result

print("\n\nRecursion Example Results")
tri_recursion(6)
```

Recursion Example Results
1
3
6
10
15
21

FUNZIONI PYTHON – RICORSIONE

```
1 def fattoriale(n):
    if (n<2):
        return 1
    return n*fattoriale(n-1)

y=fattoriale(5)
print(y)
```

Definiamo la funzione ricorsiva fattoriale() che accetta in input l'argomento (n)

```
2 def fattoriale(n): ← n = 3
    if (n<2):
        return 1
    return n*fattoriale(n-1)

y=fattoriale(3)
print(y)
```

Chiamiamo la funzione una prima volta passandogli l'argomento 3

La funzione elabora il valore e alla riga 4 tenta di restituire il valore (n=3) moltiplicato per il risultato della funzione fattoriale(n-1) ossia fattoriale(2).

```
3 def fattoriale(n): ← n = 2
    if (n<2):
        return 1
    return n*fattoriale(n-1)

y=fattoriale(3)
print(y)
```

La funzione richiama se stessa fornendo l'argomento 2 (PRIMA RICORSIONE)

Il processo si ripete di nuovo durante la prima ricorsione. La funzione cerca di restituire il prodotto tra il valore corrente (n=2) e il risultato della funzione fattoriale(n-1) ossia fattoriale(1). Quindi richiama se stessa fornendo l'argomento 1 (seconda ricorsione)

4

```
def fattoriale(n): ← n = 1
    if (n<2):
        return 1
    return n*fattoriale(n-1)

y=fattoriale(3)
print(y)
```

Nella ricorsione successiva la variabile n è uguale a 1. Essendo inferiore a 2, l'istruzione if alla riga 3 termina la seconda ricorsione senza invocarsi di nuovo. La funzione ricorsiva restituisce 1 alla funzione chiamante (return 1).

5

```
def fattoriale(n):
    if (n<2):
        return 1
    return n*fattoriale(n-1)

y=fattoriale(3)
print(y)
```

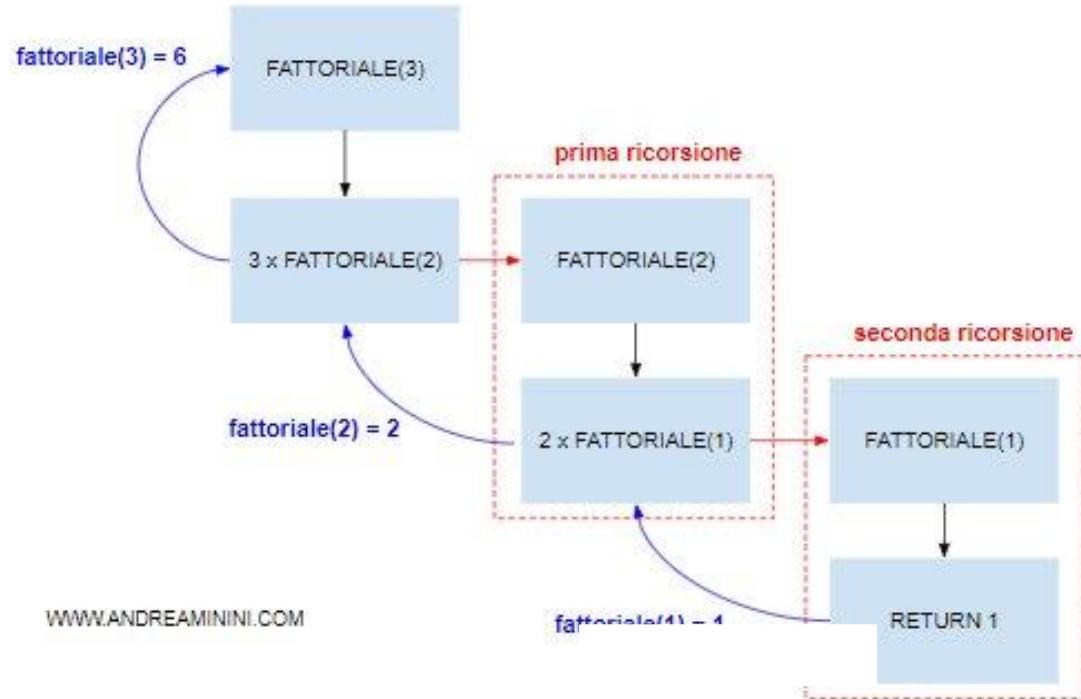
Il processo continua a ritroso fino alla prima ricorsione.

Il fattoriale è la moltiplicazione di un numero per tutti i numeri positivi interi che lo precedono: es:

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$$

$$7! = 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 5040$$

FUNZIONI PYTHON – RICORSIONE



```
def fattoriale(n):
    if n < 2:
        return 1
    else:
        return n * fattoriale(n-1)

inp = input('inserisci il numero:')
if inp.isnumeric():
    print(fattoriale(int(inp)))
else:
    print('input errato!')
```

FUNZIONI IN PYTHON – FUNZIONI LAMBDA / FUNZIONI ANONIME

FUNZIONI LAMBDA

Una funzione anonima è una **funzione che è definita senza un nome.**

Mentre una normale funzione è definita usando la keyword «`def`», le funzioni anonime sono definite usando la keyword **«lambda»**.

Per questo motivo le funzioni anonime si chiamano anche funzioni lambda.

COME USARE LE FUNZIONI LAMBDA

SINTASSI

```
lambda arguments : expression
```

L'espressione è eseguita e il risultato è tornato.

Una funzione lambda **può accettare qualsiasi numero di argomenti ma può avere solo un'espressione.**

Quest'ultima viene valutata e ritornata.

Aggiunge 10 all'argomento `a` e ritorna il risultato:

```
x = lambda a: a + 10
print(x(5))
#output 15
```

Funzione lambda
Espressione che viene valutata e ritornata
Argomento della funzione

La funzione non ha nome. Ritorna un **oggetto funzione** che è assegnato all'identificatore `x`.

```
# Più argomenti passati
# Moltiplica l'argomento a con l'argomento b e ritorna il risultato.

x = lambda a, b: a * b
print(x(5, 6))
```

FUNZIONI IN PYTHON – FUNZIONI LAMBDA - utilizzo

UTILIZZO DELLE FUNZIONI LAMBDA

Si usano le funzioni lambda quando abbiamo bisogno di una **funzione senza nome per un breve periodo di tempo.**

In Python sono **generalmente utilizzate come argomento di una funzione di ordine maggiore** (una funzione che accetta altre funzioni come argomenti) oppure all'interno di altre funzioni.

Le funzioni lambda sono spesso usate insieme a funzioni built-in come filter(), map(), etc.

Supponiamo di avere una funzione con un parametro `n` che questo parametro venga moltiplicato con un numero sconosciuto:

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

Usiamo questa definizione di funzione per creare una che raddoppia sempre il numero che le viene passato e un'altra che triplica il numero passato.

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

mydoubler = myfunc(2)

print(mydoubler(11))
```

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

mytrippler = myfunc(3)

print(mytrippler(11))
```

Oppure usare la stessa definizione di funzione per eseguire entrambe le funzioni.

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

mydoubler = myfunc(2)
mytrippler = myfunc(3)

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

22
33

```
I = [1, 2, 3, 4]
I2 = list(filter(lambda v: v > 2, I))
print(I2)
```

Object Oriented Programming in Python

PROGRAMMAZIONE ORIENTATA AGLI OGGETTI –
CLASSI E OGGETTI

OOP – OBJECT ORIENTED PROGRAMMING

Python è un linguaggio di programmazione multi-paradigma, nel senso che **supporta diversi tipi di approcci di programmazione**.

Uno degli approcci più comuni per risolvere problemi in programmazione è la creazione di oggetti. Questo approccio è conosciuto come **Programmazione Orientata agli Oggetti (OOP)**.

Un oggetto ha due caratteristiche:

1. Ha degli **attributi (o anche proprietà)**
2. Ha dei **metodi** (dei comportamenti)

ESEMPIO:

Un pappagallo è un oggetto ed ha le seguenti proprietà:

- ATTRIBUTI: nome, età, colore
- COMPORTAMENTI: cantare, ballare

Il concetto di OOP si focalizza sulla creazione di codice riusabile (DRY – Don't repeat yourself).

Il concetto di OOP include **oggetti, classi, costruttori, encapsulamento, ereditarietà e polimorfismo**.

CLASSE

La classe è come un progetto per la creazione di un oggetto. Si può pensare ad una classe come ad una carta di identità vuota. Questa contiene tutte le informazioni riguardo nome, data di nascita, ecc.

Ogni singola persona è un oggetto.

```
class Person
```

Si usa la **keyword class** per creare una classe Person vuota.

Dalla classe, possiamo **costruire le istanze**.

Un **istanza** è uno **specifico oggetto creato** da una particolare classe.

Si possono creare tanti oggetti quanti se ne desiderano utilizzando la classe.

OOP – OBJECT ORIENTED PROGRAMMING

OGGETTO

Un oggetto è **un'istanza di una classe.**

E' una **collezione di attributi (variabili) e metodi.**

Si usano gli oggetti di una classe per eseguire azioni.

Gli oggetti hanno **tre caratteristiche:**

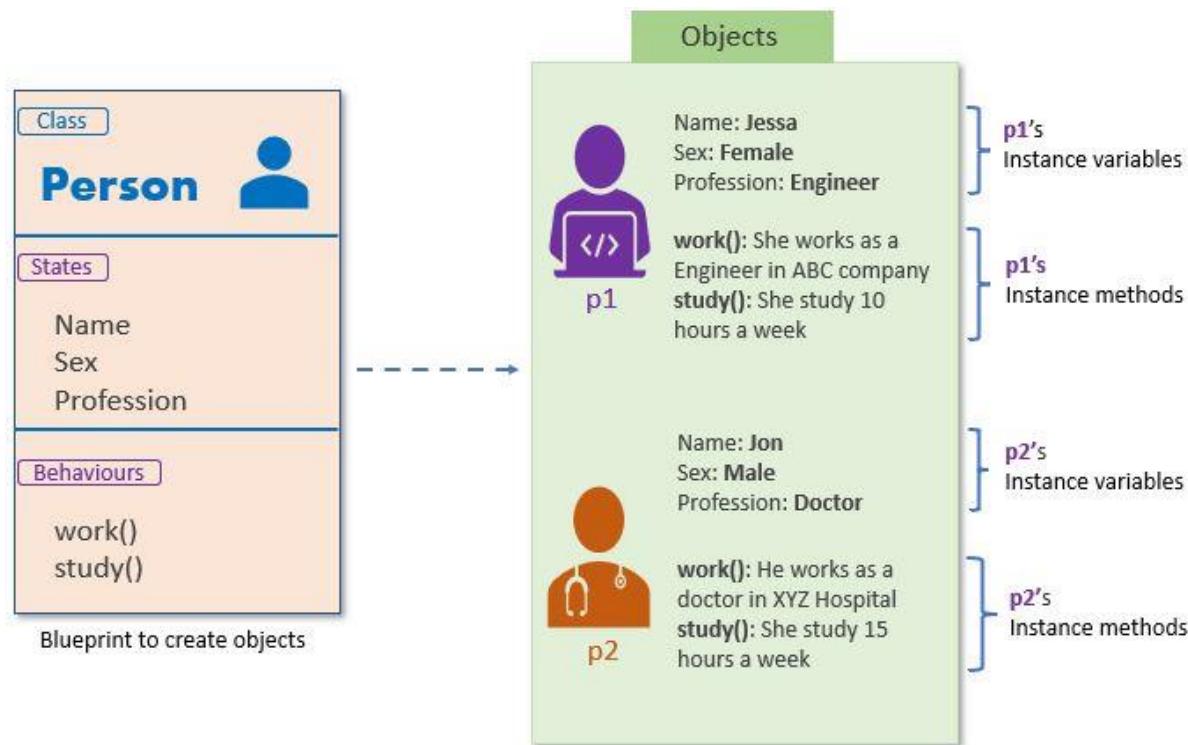
- **IDENTITA'**: ogni oggetto deve essere **univocamente identificato.**
- Hanno **PROPRIETÀ (ATTRIBUTI)**: un oggetto ha attributi che rappresentano gli stati dell'oggetto
- Hanno **COMPORTAMENTI (METODI)**

Usando i metodi di un oggetto possiamo modificarne gli attributi.

Un oggetto è una collezione di vari dati e funzioni che operano sui dati stessi.

Python è un linguaggio orientato agli oggetti quindi tutto in Python è trattato come un oggetto.

OOP IN PYTHON – CLASSI E OGGETTI



UN ESEMPIO CONCRETO DI CLASSI E OGGETTI

Class: Person

- Stati (attributi / proprietà): nome, sesso, professione
- Comportamenti (metodi): studio, lavoro

Usando la classe Person, possiamo creare oggetti diversi che rappresentano stati e comportamenti diversi. Gli oggetti vengono creati a partire dalla stessa classe ma hanno stati e comportamenti diversi tra loro.

OGGETTO 1 : Tessa

STATI (ATTRIBUTI)

- Nome: Tessa
- Sesso: femmina
- Professione: Engineer

COMPORTAMENTI (METODI)

- Lavorare: lavora come
software developer alla ABC
Company
- Studio: studia 10 ore a
settimana

OGGETTO 2 : John

STATI (ATTRIBUTI)

- Nome: John
- Sesso: male
- Professione: Doctor

COMPORTAMENTI (METODI)

- Lavorare: lavora come dottore
- Studio: studia 15 ore a
settimana

OOP PYTHON – DEFINIRE UNA CLASSE

Come già visto, la definizione di una classe inizia con la **keyword «class»**.

La prima stringa all'interno della classe è chiamata **«docstring»** ed ha una breve descrizione della classe. Anche se non è obbligatoria, è altamente raccomandata.

```
class MyNewClass:  
    '''This is a docstring. I have created a new class'''  
    pass
```

Una classe crea un nuovo namespace locale dove sono definiti tutti i suoi attributi che possono essere dati o funzioni.

Esistono anche **attributi speciali** all'interno delle classi e cominciano con **__** (doppio underscore). Ad esempio `__doc__` ritorna il docstring della classe.

Appena definiamo una classe, viene creato un nuovo oggetto avente lo stesso nome della classe.

Questo consente di accedere agli attributi e di istanziare nuovi oggetti di quella classe.

```
class Person:  
    "This is a person class"  
    age = 10  
  
    def greet(self):  
        print('Hello')  
  
    # Output: 10  
    print(Person.age)  
  
    # Output: <function Person.greet>  
    print(Person.greet)  
  
    # Output: "This is a person class"  
    print(Person.__doc__)
```

```
class Person:  
    def __init__(self, name, surname):  
        self.name = name  
        self.surname = surname  
        pass  
  
    def who_is(self):  
        print('mi chiamo {} {}'.format(self.name,  
                                       self.surname))  
  
monia = Person('Monia', 'Masini')  
carlo = Person('Carlo', 'Gallini')  
  
monia.who_is()  
carlo.who_is()
```

CONVENZIONI PER I NOMI DELLE CLASSI

- **UpperCaseCamelCase**
- Le classi delle **eccezioni** dovrebbero finire in **Error**
- Le **classi native** di Python sono generalmente in **lowercase**

```
10  
<function Person.greet at 0x7fc78c6e8160>  
This is a person class
```

OOP PYTHON – CREARE UN OGGETTO

Un oggetto è essenziale per lavorare con gli attributi di una classe.

Il processo di creazione di un oggetto è chiamato **ISTANZIAZIONE**.

L'oggetto, a sua volta, è chiamato **ISTANZA DI UNA CLASSE**.

```
class MyClass:  
    x = 5  
  
p1 = MyClass()  
print(p1.x)  
  
# OUTPUT 5
```

La procedura per la creazione di un nuovo oggetto è simile alla chiamata di una funzione.

```
>>> harry = Person()
```

Questo creerà una **nuova istanza della classe Person**, chiamata harry.

Per accedere agli attributi dell'oggetto si usa il nome dell'oggetto. Ricordiamo che gli attributi possono essere dati o metodi. I metodi sono le funzioni proprie di una classe.

Person.greet (funzione) sarà un metodo dell'oggetto.

```
class Person:  
    "This is a person class"  
    age = 10  
  
    def greet(self):  
        print('Hello')  
  
# create a new object of Person class  
harry = Person()  
  
# Output: <function Person.greet>  
print(Person.greet)  
  
# Output: <bound method Person.greet of <__main__.Person object>>  
print(harry.greet)  
  
# Calling object's greet() method  
# Output: Hello  
harry.greet()
```

```
<function Person.greet at 0x7fd288e4e160>  
<bound method Person.greet of <__main__.Person object at 0x7fd288e9fa30>>  
Hello
```

OOP PYTHON – Il parametro SELF

```
def greet(self):  
    print('Hello')
```

Guardando l'esempio precedente si può notare il **parametro «self»** nella definizione della funzione all'interno della funzione ma noi chiamiamo il metodo semplicemente come

harry.greet()

senza alcun argomento funzionando lo stesso.

Questo perché, **ogni volta che un oggetto chiama un proprio metodo, l'oggetto stesso è passato come primo argomento.**

harry.greet()
si traduce quindi in
Person.greet(harry).

In generale, chiamare un metodo con una lista di n argomenti è come chiamare la corrispondente funzione con una lista di argomenti creata inserendo l'oggetto al quale il metodo appartiene prima del primo argomento.

Per questi motivi, **il primo argomento dei metodi di una classe deve essere l'oggetto stesso.**

Questo è convenzionalmente chiamato **«self»** ma gli può essere assegnato qualsiasi nome (anche se è raccomandato seguire la convenzione).

Il **«self»** è dunque un **riferimento alla corrente istanza della classe ed è utilizzato per accedere alle variabili che appartengono alla classe.**

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

```
p1 = MyClass()  
print(p1.x)  
  
# OUTPUT      5
```

OOP PYTHON – Parametro SELF - Esempio

```
class Student:  
    # constructor  
    def __init__(self, name, age):  
        self.name = name  
        self.age = age  
  
        # self points to the current object  
    def show(self):  
        # access instance variable using self  
        print(self.name, self.age)  
  
    # creating first object  
emma = Student('Emma', 12)  
emma.show()  
  
    # creating Second object  
kelly = Student('Kelly', 13)  
kelly.show()
```

OUTPUT

```
Emma 12  
Kelly 13
```



OOP PYTHON – COSTRUTTORI – metodo `__init__()`

Nella programmazione orientata agli oggetti, il costruttore è un **metodo speciale** utilizzato per **creare ed inizializzare un oggetto di una classe**. Questo metodo è definito all'interno della classe e viene **chiamato ogni volta in cui un nuovo oggetto di quella classe è istanziato**.

- il costruttore è **eseguito automaticamente nel momento della creazione dell'oggetto**
- l'uso primario di un costruttore è quello di dichiarare ed inizializzare i **dati delle variabili delle istanze** di una classe. Il costruttore contiene una **serie di istruzioni** che vengono eseguite al **momento della creazione dell'oggetto** per inizializzarne gli attributi.

OOP PYTHON – COSTRUTTORI – metodo `__init__()`

```
class Student:

    # constructor
    # initialize instance variable
    def __init__(self, name):
        print('Inside Constructor')
        self.name = name
        print('All variables initialized')

    # instance Method
    def show(self):
        print('Hello, my name is', self.name)

# create object using constructor
s1 = Student('Emma')
s1.show()
```

```
Inside Constructor
All variables initialized

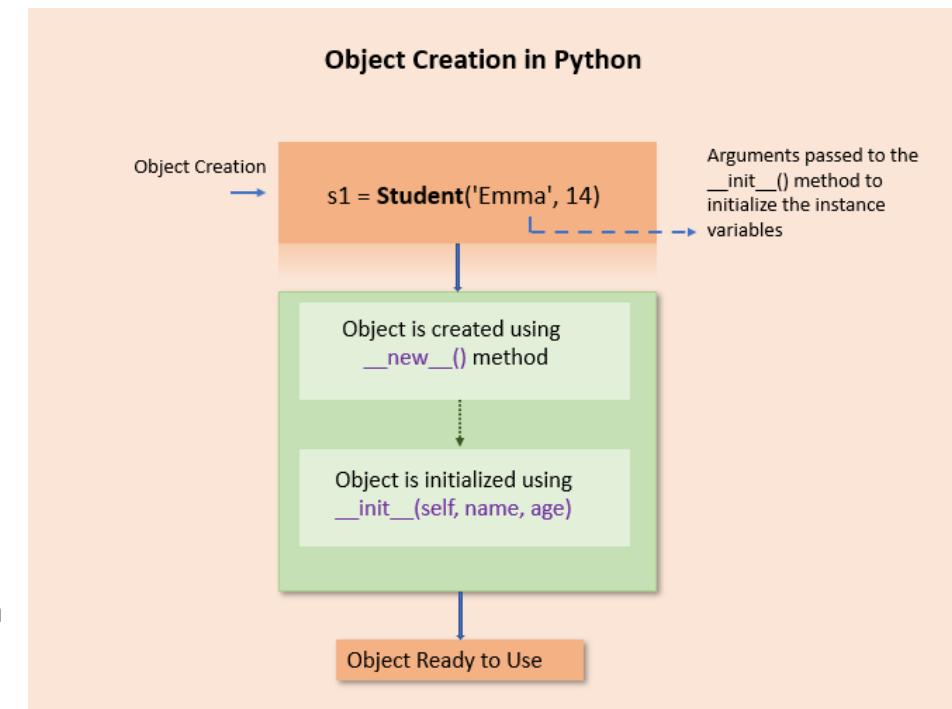
Hello, my name is Emma
```

Nell'esempio è stato creato un oggetto s1 usando il costruttore.

Nel momento in cui si crea un oggetto Student, `name` è passato come argomento al metodo `__init__()` per inizializzare l'oggetto.

In maniera analoga, tanti altri oggetti della classe Student possono essere creati passando nomi diversi come argomenti.

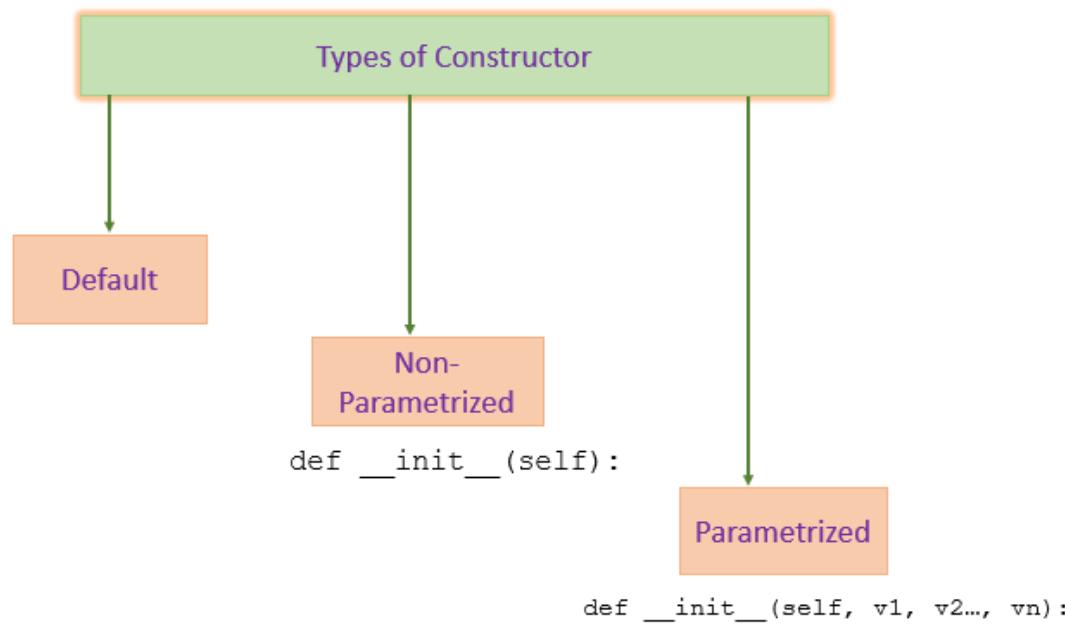
- per ogni oggetto, il costruttore sarà eseguito solamente una volta. Per esempio, se si creano quattro oggetti, il metodo `__init__()` il costruttore sarà chiamato quattro volte.
- in Python ogni classe ha un costruttore ma non è necessario definirlo esplicitamente. Definire il costruttore nella classe è opzionale
- se il costruttore non è definito, Python provvederà con uno di default.



OOP PYTHON – COSTRUTTORI – Tipi di costruttori

In Python esistono **tre tipi di costruttori**:

- COSTRUTTORE DI **DEFAULT**
- COSTRUTTORE **NON PARAMETRIZZATO**
- COSTRUTTORE **PARAMETRIZZATO**



COSTRUTTORE DI DEFAULT

Python prevede un costruttore di default se non viene definito alcun costruttore all'interno della classe.

E' un **costruttore vuoto**, senza un body e **si limita ad inizializzare gli oggetti senza prevedere ulteriori azioni**.

```
class Employee:  
  
    def display(self):  
        print('Inside Display')  
  
emp = Employee()  
emp.display()
```

Inside Display

Nell'esempio non è presente alcun costruttore ma è in ogni caso possibile creare un oggetto per la classe **Employee** perchè Python ha **aggiunto il costruttore di default durante la compilazione del programma**.

OOP PYTHON – COSTRUTTORI – Tipi di costruttori

COSTRUTTORI NON PARAMETRIZZATI

Sono i costruttori **senza alcun argomento**.

- Sono utilizzati per **inizializzare ogni oggetto con i valori di default**.
- Non accetta **argomenti** durante la creazione degli oggetti ed inizializza ogni oggetto con gli stessi valori.

```
class Company:

    # no-argument constructor
    def __init__(self):
        self.name = "PYnative"
        self.address = "ABC Street"

    # a method for printing data members
    def show(self):
        print('Name:', self.name, 'Address:', self.address)

# creating object of the class
cmp = Company()

# calling the instance method using the object
cmp.show()
```

Name: PYnative Address: ABC Street

OOP PYTHON – COSTRUTTORI – Tipi di costruttori

COSTRUTTORI PARAMETRIZZATI

Sono i costruttori con **parametri ed argomenti definiti**. Si possono passare **valori differenti ad ogni oggetto** ogni volta che ne viene creato uno, usando il costruttore parametrizzato.

Il primo parametro del costruttore è **self** che è un riferimento all'oggetto stesso. Tutti gli altri parametri verranno dichiarati dopo il self.

Può avere **qualsiasi numero di argomenti**.

Si consideri una compagnia con migliaia di impiegati. In questo caso, quando viene creato ogni singolo oggetto impiegato, **si dovranno passare name, age, and salary differenti**.

Questo è il caso in cui viene usato il costruttore parametrizzato.

```
class Employee:  
    # parameterized constructor  
    def __init__(self, name, age, salary):  
        self.name = name  
        self.age = age  
        self.salary = salary  
  
    # display object  
    def show(self):  
        print(self.name, self.age, self.salary)  
  
# creating object of the Employee class  
emma = Employee('Emma', 23, 7500)  
emma.show()  
  
kelly = Employee('Kelly', 25, 8500)  
kelly.show()
```

```
Emma 23 7500  
Kelly 25 8500
```



OOP PYTHON – COSTRUTTORI – Tipi di costruttori

COSTRUTTORI CON VALORI DI DEFAULT

I valori di default saranno utilizzati da Python **se non vengono passati argomenti al costruttore** al momento dell'istanziazione di un nuovo oggetto.

Nell'esempio a fianco non sono stati passati i valori di age e classroom, così Python farà riferimento ai valori di default.

```
class Student:  
    # constructor with default values age and classroom  
    def __init__(self, name, age=12, classroom=7):  
        self.name = name  
        self.age = age  
        self.classroom = classroom  
  
    # display Student  
    def show(self):  
        print(self.name, self.age, self.classroom)
```

```
# creating object of the Student class  
emma = Student('Emma')  
emma.show()
```

```
kelly = Student('Kelly', 13)  
kelly.show()
```

```
Emma 12 7  
Kelly 13 7
```

OOP PYTHON – METODI DELL'OGGETTO (o metodi dell'istanza)

I metodi sono **funzioni definite all'interno del corpo della definizione della classe**. Sono usati per **definire il comportamento di un oggetto**.

```
class Parrot:

    # instance attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # instance method
    def sing(self, song):
        return "{} sings {}".format(self.name, song)

    def dance(self):
        return "{} is now dancing".format(self.name)

# instantiate the object
blu = Parrot("Blu", 10)

# call our instance methods
print(blu.sing("'Happy'"))
print(blu.dance())
```

```
Blu sings 'Happy'
Blu is now dancing
```

Nell'esempio vengono definiti **due metodi**:

- sing()
- dance()

Questi sono chiamati **metodi dell'istanza** perché sono chiamati su un'istanza dell'oggetto (nel nostro caso blu).

METODI PRIVATI

Come visto per gli attributi di una classe, i modificatori di accesso **limitano la visibilità di una funzione** o di una variabile. Dichiarare una funzione o una variabile come privata, **limita l'accesso solo alla classe che la incapsula.**

In Python i metodi privati sono metodi a cui non è possibile accedere al di fuori della classe in cui sono dichiarati.

Per dichiarare un metodo privato, si inseriscono **i doppi caratteri di underscore** all'inizio del nome del metodo.

_metodo_privato(self)

Le sottoclassi non ereditano i metodi privati della classe genitore.

```
# p2.py
class P:
    def __init__(self, name, alias):
        self.name = name      # public
        self.__alias = alias  # private

    def who(self):
        print('name :', self.name)
        print('alias :', self.__alias)

    def __foo(self):      # private method
        print('This is private method')

    def foo(self):        # public method
        print('This is public method')
        self.__foo()
```

Se proviamo:

```
x = P('Alex',
       'amem')
x.__foo()
```

```
Traceback (most recent call last):
  File "C:\Users\PC\PycharmProjects\pythonProject\python_test.py", line 19, in <module>
    x.__foo()
AttributeError: 'P' object has no attribute '__foo'

Process finished with exit code 1
```

METODI PRIVATI – accedere i metodi privati

Esistono due modi per accedere ai metodi privati in maniera corretta:

NAME MANGLING:

Uno dei modi corretti di accedere ai metodi privati è ricorrere, come per gli attributi privati, al **name mangling**:

(*_classname__privatemethod*)

```
x = P('Alex', 'amem')  
x._P__foo()
```

```
This is private method
```

CHIAMARE IL METODO PRIVATO ATTRaverso UN METODO PUBBLICO:

```
def foo(self):      # public  
    method  
    print('This is public method')  
    self._foo()
```

```
x.foo()
```

```
This is public method  
This is private method
```

APPROFONDIMENTI

approfondimenti

* Python 2

sezione OOP Approfondimento 1

OOP PYTHON – PASS statement

Come già visto per funzioni, if statement etc, **la definizione di una classe non può essere vuota.**
Se si avesse la necessità di lasciarla vuota, si utilizza il **pass statement** per evitare di incorrere in errori.

```
class Person:  
    pass  
  
# having an empty class definition like this, would raise an error without the pass statement
```

PYTHON F-STRING

F-STRING

A partire da Python 3.6, è disponibile una nuova sintassi per la **formattazione delle stringhe**.

Le f-strings hanno il **prefisso *f*** e usano le **parentesi graffe** per i valori.

In via generale, la sintassi è:

f'{variable_name}'

```
name = 'Peter'  
age = 23  
  
#vecchio sistema di formattazione delle stringhe  
print('%s is %d years old' % (name, age))  
  
#formattazione stringhe con format() a partire da  
Python 3.0  
print('{} is {} years old'.format(name, age))  
  
#Python f-strings a partire da Python 3.6  
print(f'{name} is {age} years old')
```

```
Peter is 23 years old  
Peter is 23 years old  
Peter is 23 years old
```

F-string expressions

Tra parentesi graffe si possono inserire anche **espressioni che verranno poi valutate e/o calcolate**.

```
bags = 3  
apples_in_bag = 12  
  
print(f'There are total of {bags * apples_in_bag}  
apples')
```

L'espressione ***bags * apples_in_bag*** verrà calcolata prima di essere stampata.

```
There are total of 36 apples
```

APPROFONDIMENTI

approfondimenti

* Python 2

sezione F-STRING Approfondimento 1

approfondimenti

* Python 2

sezione OOP Approfondimento 2 Ereditarietà, encapsulamento e Polimorfismo

approfondimenti

* Python 2

sezione Approfondimento Iteratori

approfondimenti

* Python 2

sezione Approfondimento Moduli

PYTHON - MODULI

I moduli, anche conosciuti come librerie in altri linguaggi, sono dei file usati per raggruppare costanti, funzioni e classi, che ci consentono di suddividere e organizzare meglio i nostri progetti. Python include già una lista estensiva di moduli standard (anche conosciuti come standard library), ma è anche possibile scaricarne o definirne di nuovi.

Prima di poter accedere ai contenuti di un modulo, è necessario importarlo. Per farlo, usiamo il costrutto **import**

```
import random as rnd  
print(rnd.randint(0, 10))
```

Questa forma di import ci consente di accedere a qualsiasi nome definito all'interno del modulo, ma per farlo dobbiamo sempre usare la sintassi `modulo.nome`.

PYTHON - MODULI

È possibile importare nomi direttamente usando la sintassi from modulo import nome
in questo caso verranno importate solamente le funzioni indicate

```
from random import randint
print(randint(0, 10))
```

eventualmente si potrebbe anche assegnare un nuovo nome alla funzione

```
from random import randint as numero_random_intero, random as
numero_random
print(numero_random_intero(0, 10))
```

Questa tecnica può essere usata anche nel caso in cui i nomi siano particolarmente lunghi o per evitare conflitti tra nomi simili o uguali che appartengono a moduli diversi.

PYTHON - MODULI

Esiste un'altra forma che ci permette di importare tutti i nomi di un modulo, usando la sintassi `from modulo import *`

```
from random import *
print(randint(0, 10))
```

è però sconsigliato l'utilizzo di questa forma in quanto non si ha controllo ne sul nome ne sui nomi importati e quindi si potrebbe cadere in un conflitto di nomi tra più moduli o funzioni di moduli definite

PYTHON - MODULI

è possibile anche definire i propri moduli, in questo caso si creerà un file che verrà importato.

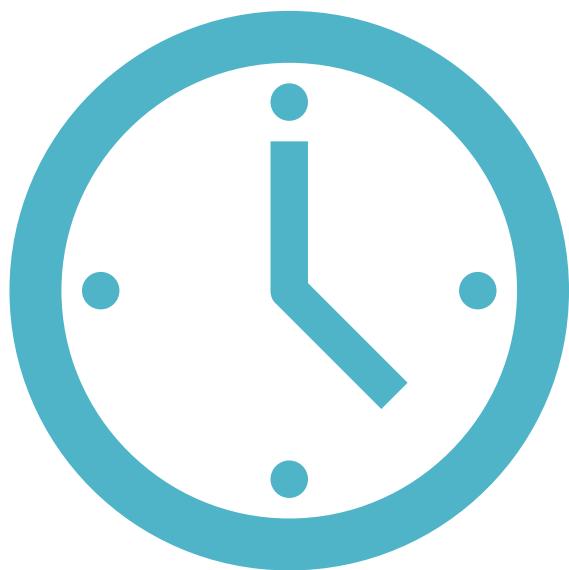
le funzioni contenute nel modulo importato hanno le stesse caratteristiche di una libreria importata dalla standard library

es: genfunction.py

```
def stampa_nome():
    print(__name__)
```

main.py

```
#importare genfunction e richiamare la funzione nom e domani
import genfunction as gf
gf.stampa_nome()
```



DATE IN
PYTHON -
datetime

DATE – MODULO DATETIME

LE DATE IN PYTHON

Una data in Python non è un tipo di dato a se' stante. Per poter lavorare con le date come oggetti date dobbiamo **importare il modulo datetime**.

Il modulo datetime possiede **molti metodi** utili che ritornano informazioni sulla data.

OTTENERE LA DATA CORRENTE – today()

Usiamo il **metodo today()** definito nella classe date, per ottenere un oggetto date che contiene la data locale.

```
import datetime  
  
date_object = datetime.date.today()  
print(date_object)
```

2022-08-04

OTTENERE DATA E ORARIO CORRENTI (.now())

ESEMPIO: importazione del modulo datetime e visualizzazione data e orario corrente

Una delle classi definite nel modulo datetime, è la **classe datetime**.

Usiamo dunque il **metodo now ()** per creare un oggetto datetime contenente l'ora e la data locali.

```
import datetime  
  
datetime_object = datetime.datetime.now()  
print(datetime_object)
```

La data contiene:

- anno
- mese
- giorno
- ora
- Minuti
- Secondi
- microsecondi

2022-08-04 12:21:07.827916

DATE IN PYTHON – contenuto di datetime module

ESAMINARE IL CONTENUTO DEL MODULO DATETIME

Possiamo usare la funzione `dir()` per ottenere una lista di tutti gli attributi di un modulo.

```
import datetime  
print(dir(datetime))
```

OUTPUT:

```
['MAXYEAR', 'MINYEAR', '__builtins__', '__cached__', '__doc__',  
 '__file__', '__loader__', '__name__', '__package__', '__spec__',  
 '__divide_and_round', 'date', 'datetime', 'datetime_CAPI', 'time',  
 'timedelta', 'timezone', 'tzinfo']
```

Le classi più usate all'interno del modulo datetime sono:

- date class
- time class
- datetime class
- timedelta class



Date in Python - Aggiungere Giorni

per calcolare con le date è possibile utilizzare la funzione timedelta

```
def domani():
    return datetime.date.today() +
    datetime.timedelta(days=1)
```

APPROFONDIMENTI ITERATORI

approfondimenti

* Python 2

sezione Approfondimento Datetime

PYTHON MATH

funzioni built in (no import math)

PYTHON MATH – funzioni matematiche built-in

Python possiede una serie di **funzioni matematiche native**, oltre ad un vasto **modulo math** che permette di effettuare **numerose operazioni sui numeri**.

□ min() e max()

Le funzioni min() e max() possono essere usate per trovare il **valore più alto o il valore più basso** in un iterabile.

SINTASSI

min/max (n1, n2, n3,)

Oppure

min /max(iterabile)

PARAMETRI:

n1, n2, n3,

Uno o più elementi da comparare

Iterabile

Un iterabile con uno o più elementi da comparare

```
x = min(5, 10, 25) #OUTPUT: 5  
y = max(5, 10, 25) #OUTPUT: 25
```

```
print(x)  
print(y)
```

```
x = (5, 10, 25)  
y = (5, 10, 25)
```

```
print(min(x)) #OUTPUT: 5  
print(max(y)) #OUTPUT: 25
```

PYTHON MATH – funzioni matematiche built-in

□ Pow()

La funzione pow (x, y) ritorna il valore di x elevato alla potenza y
(x^y)

SINTASSI

Pow(x, y, z)

x = base della potenza

y = esponente della potenza

z = optional. Il modulo.

```
x = pow(4, 3)
```

```
print(x) #OUTPUT 64
```

Se il terzo parametro è presente, la funzione ritorna x elevato alla y, modulo z

```
x = pow(4, 3, 5)
print(x) #OUTPUT 4
```

□ Round(number ,ndigits)

La funzione round() arrotonda il numero.

Ritorna un **float** che è la **versione arrotondata del numero specificato, con il numero specificato di cifre decimali.**

Il numero di decimali di default è 0, questo significa che, se non viene specificato il secondo parametro, la funzione ritorna un numero arrotondato all'intero.

PARAMETRI

Number	Required. Il numero da arrotondare
Ndigits	Optional. Il numero di decimali da usare quando si arrotonda il numero. Default è 0.

```
x = round(5.76543)
print(x)    #OUTPUT 6

x = 5.76543
print (round(x,2)) #OUTPUT 5.77
```

PYTHON MATH – funzioni matematiche built-in

❑ abs()

La funzione `abs()` ritorna il **valore assoluto di un numero**. Accetta un solo parametro (required) che è il numero di cui si vuole il valore assoluto.

```
x = abs(-7.25)
print(x) #OUTPUT: 7.25
```

```
#ritorna il valore assoluto di un numero complesso
x = abs(3+5j)
print(x) #OUTPUT 5.830951894845301
```

❑ Int()

La funzione `int()` converte il valore specificato in un **numero intero**.

SINTASSI

`int(value, base)`

PARAMETRO	DESCRIZIONE
<code>value</code>	Un numero o una stringa che può essere convertito in un numero intero
<code>base</code>	Un numero che rappresenta il formato del numero. Valore di default 10 (numero base 10).

```
print(int("0b11", 2)) # base 2 numero binario
print(int("0o1110", 8)) # base 2 numero ottale
print(int("0x11AF", 16)) # base 2 numero esadecimale
# 3
# 584
# 4527
#
```

```
x = int("12")
print(x) #OUTPUT 12
```

APPROFONDIMENTI ITERATORI

approfondimenti

* Python 2

sezione Approfondimento Math Module

PIP – Package manager for
Python

pip installare

installare pip su windows:

```
python -m ensurepip
```

PYTHON PIP

COS'E' PIP

Pip è il **manager standard** dei packages (o dei moduli) per Python.

Si usa pip per **installare pacchetti addizionali** che non sono presenti nella libreria standard di Python.

COS'E' UN PACCHETTO (PACKAGE)

Un pacchetto (package) contiene **tutti i file che servono per un modulo**.

I moduli sono librerie di codice che possono essere inclusi nei progetti Python.

COME INSTALLARE PIP

Dalla versione 3.4 PIP è incluso di default. Per verificare se pip è installato, digitare il comando

pip --version

nella console.

Se pip è già disponibile nel sistema verrà mostrata la versione:

```
pip 21.1.3 from c:\users\pc\appdata\local\programs\python\python39\lib\site-packages\pip (python 3.9)
```

Se si sta usando una versione vecchia di Python o non si ha pip installato, seguire le procedure per l'[installazione](#).

UTILIZZARE PIP

Pip è un programma a riga di comando. Dopo la sua installazione, viene aggiunto un comando pip che può essere usato nel prompt dei comandi.

La sintassi base di pip è:

```
pip <pip arguments>
```

Oltre la libreria standard, la comunità di Python contribuisce continuamente con numerosi pacchetti fatti su misura per frameworks, strumenti e librerie.

Molti di questi pacchetti sono ufficialmente pubblicati su [Python Package Index\(PyPI\)](#).

Pip consente di scaricare ed installare questi pacchetti.

INSTALLARE PACCHETTI – INSTALLAZIONE BASE

Il comando **install** è usato per installare pacchetti usando pip.

ESEMPIO:

Supponiamo di voler installare **requests**, una popolare libreria HTTP per Python:

```
pip install requests
```

Pip installerà il pacchetto indicato (nel nostro caso «**requests**») e si **occuperà anche di installare tutte le altre dipendenze** per quel pacchetto.

SPECIFICARE LA VERSIONE DEL PACCHETTO

Quando pip install viene usato nella sua forma più basilare, pip **scaricherà la versione più recente del pacchetto**.

Se si ha la necessità di specificare la versione del pacchetto, usare il seguente comando:

```
pip install requests==2.21.0
```

UTILIZZARE PIP

□ VISUALIZZARE LA LISTA DEI PACCHETTI INSTALLATI

Pip list è il comando per elencare tutti i pacchetti disponibili nell'ambiente Python corrente.

Package	Version

certifi	2019.11.28
chardet	3.0.4
idna	2.8
pip	19.3.1
requests	2.22.0
setuptools	45.0.0
urllib3	1.25.7
wheel	0.33.6

□ INFORMAZIONI SUL PACCHETTO CON PIP SHOW

Il comando pip show mostra informazioni su uno o più pacchetti installati.

```
pip show requests
```

```
Name: requests
Version: 2.22.0
Summary: Python HTTP for Humans.
Home-page: http://python-requests.org
Author: Kenneth Reitz
Author-email: me@kennethreitz.org
License: Apache 2.0
Location: c:\users\dell\desktop\venv\lib\site-packages
Requires: certifi, chardet, urllib3, idna
Required-by:
```

Il comando show mostra informazioni sulla libreria requests. Notare le voci «requires» e «required-by».

REQUIRES: sono le dipendenze di cui ha bisogno la libreria requests

REQUIRED-BY: mostra i pacchetti che richiedono il pacchetto in questione

UTILIZZARE PIP

□ DISINSTALLARE UN PACCHETTO CON PIP

Possiamo disinstallare un pacchetto usando il comando **pip uninstall**. Supponiamo di voler rimuovere il pacchetto *requests* dalla libreria del nostro attuale ambiente Python:

```
pip uninstall requests
```

```
Uninstalling requests-2.22.0:  
Would remove:  
  C:\Python37\lib\site-packages\requests-2.22.0.dist-info\*  
  C:\Python37\lib\site-packages\requests\*  
Proceed (y/n)? y  
Successfully uninstalled requests-2.22.0
```

Anche se il pacchetto specificato è stato rimosso, **i pacchetti che sono stati installati come dipendenze non vengono rimossi**.

Per rimuovere anche le dipendenze, si può usare il comando **pip show** per vedere i pacchetti installati rimuoverli manualmente.

□ USARE IL REQUIREMENT FILES

Un file contenente tutti i nomi dei pacchetti può essere usato per **installare i pacchetti di Python in blocco**.

Supponiamo di avere il file requirements.txt che ha il seguente contenuto:

```
numpy  
Pillow  
pygame
```

Possiamo installare tutti questi pacchetti e le loro dipendenze **usando un solo comando pip**:

```
pip install -r requirements.txt
```

L'argomento **-r** specifica a pip che stiamo passando un **file** dei requisiti piuttosto che il nome di un pacchetto.

CREARE UN REQUIREMENTS FILE

Come alternativa alla creazione manuale di un requirements file, pip offre il comando freeze.

Supponiamo che il nostro attuale ambiente Python abbia i seguenti pacchetti:

Package	Version
numpy	1.17.0
Pillow	6.1.0
pip	19.3.1
pygame	1.9.6
setuptools	45.0.0
wheel	0.33.6

I pacchetti che non escono preinstallati in Python vengono elencati usando il comando freeze

```
pip freeze
```

Il comando pip freeze mostra i pacchetti e le loro versioni nel formato del requiremens file.

```
numpy==1.17.0  
Pillow==6.1.0  
pygame==1.9.6
```

Questo contenuto può essere reindirizzato per creare un requirements file usando il comando:

```
pip freeze > requirements.txt
```

Viene così creato un nuovo file requirements.txt nella directoty in cui si sta lavorando. Può essere in seguito usata in altri ambienti Python per installare specifiche versioni di pacchetti.

USARE I PACCHETTI NEL CODICE – import

Una volta che il pacchetto è stato installato, è pronto per essere utilizzato.

IMPORTAZIONE ED UTILIZZO DEL PACKAGE «camelcase»

```
import camelcase

c = camelcase.CamelCase()

txt = "lorem ipsum dolor sit amet"

print(c.hump(txt))

#This method capitalizes the first letter of each word.
```

LoREM ipsum dolor sit amet

TRY...EXCEPT

Gestione delle eccezioni usando try, except ed il finally statement

LE ECCEZIONI IN PYTHON

Quando nel codice si verifica un errore (o eccezione), Python normalmente interrompe la sua esecuzione e genera un messaggio di errore.

Python possiede **molte eccezioni built-in** che sono sollevate **quando il programma incontra un errore** (cioè quando qualcosa nel programma va male).

Quando capitano queste eccezioni, l'interprete **Python interrompe l'attuale processo**, lo passa al processo di chiamata affinchè l'eccezione sia gestita. Se non succede il programma crasha.

ESEMPIO: consideriamo un programma dove una funzione A chiama la funzione B, che a sua volta chiama la funzione c. Se capita un errore nella funzione c ma non è gestito in c, l'eccezione passa a b e così via.

Se, invece, non viene mai gestito, apparirà un messaggio di errore e il programma sarà bloccato improvvisamente.

BLOCCO try:

Testa un blocco di codice per verificare la presenza di errori

BLOCCO else:

Permette di eseguire del codice quando non ci sono errori

BLOCCO except:

Permette la gestione dell'errore

BLOCCO finally:

Esegue del codice indipendentemente dalla presenza o meno di errori

CATTURARE EXCEPTIONS IN PYTHON

In Python, le **eccezioni** possono essere gestite usando lo statement

«**try**»

L'operazione critica che potrebbe sollevare un'eccezione è posta all'interno della **clausola try**.

Il codice che gestisce le eccezioni, invece, è scritto nella **except clause**.

E' così possibile scegliere quale operazione eseguire una volta che l'eccezione è stata catturata.

```
#The try block will generate an error, because x is not defined:  
  
try:  
    print(x)  
except:  
    print("An exception occurred")  
  
#OUTPUT An exception occurred
```

Siccome il **blocco try** solleva un errore, il blocco **except** sarà eseguito.

Senza il blocco **try**, il programma andrebbe in crash e solleverebbe un errore.

```
#This will raise an exception, because x is not defined:  
  
print(x)  
  
Traceback (most recent call last):  
  File "demo_try_except_error.py", line 3, in <module>  
    print(x)  
NameError: name 'x' is not defined
```

CATTURARE EXCEPTIONS IN PYTHON

Nel programma a fianco, si cicla sui valori di randomList.

La parte che può causare un'eccezione è posizionata nel blocco try.

Se nessuna eccezione si presenta, il blocco except è saltato e il normale flusso continua.

Se, però, si verifica un'eccezione, questa è catturata dal blocco except.

All'interno del blocco except viene stampato il nome dell'eccezione usando la funzione `exc_info()` del modulo `sys`.

Si può notare che «a» causa un `ValueError` e «0» causa un `ZeroDivisionError`.

```
# import module sys to get the type of exception
import sys

randomList = ['a', 0, 2]

for entry in randomList:
    try:
        print("The entry is", entry)
        r = 1/int(entry)
        break
    except:
        print("Oops!", sys.exc_info()[0], "occurred.")
        print("Next entry.")
        print()
    print("The reciprocal of", entry, "is", r)
```

```
The entry is a
Oops! <class 'ValueError'> occurred.
Next entry.
```

```
The entry is 0
Oops! <class 'ZeroDivisionError'> occurred.
Next entry.
```

```
The entry is 2
The reciprocal of 2 is 0.5
```

CATTURARE EXCEPTIONS IN PYTHON

Siccome ogni eccezione in Python eredita dalla classe base **Exception**, si può anche eseguire lo stesso compito dell'esempio precedente in questo modo, ottenendo lo stesso output:

```
# import module sys to get the type of exception
import sys

randomList = ['a', 0, 2]

for entry in randomList:
    try:
        print("The entry is", entry)
        r = 1/int(entry)
        break
    except Exception as e:
        print("Oops!", e.__class__, "occurred.")
        print("Next entry.")
        print()
print("The reciprocal of", entry, "is", r)
```

```
import sys

try:
    print(x)
except:
    print(sys.exc_info()[1])
finally:
    print('dopo try')

# oppure:

try:
    print(x)
except Exception as ex:
    print(ex.__str__())
finally:
    print('dopo try')
```

```
try:
    print(10/0)
except Exception as e:
    print(e.__str__())
else:
    print('esegue se non va in errore')
finally:
    print('esegue sempre')
```

APPROFONDIMENTI ITERATORI

approfondimenti

* Python 2

sezione Approfondimento Try Except

PYTHON USER INPUT

Prendere i dati
in input
dall'utente

USER INPUT – input()

Python consente di ottenere dati forniti dall'utente grazie al metodo `input()`.

Python 3.6 usa il metodo `input()`

Python 2.7 usa il metodo `raw_input()`.

METODO `input()`

Questa funzione prima prende l'input dall'utente e lo **converte in stringa**. Il tipo di oggetto ritornato sarà sempre <type 'str'>.

Non valuta l'espressione, si limita a ritornare l'input come stringa.

Quando la funzione `input()` viene chiamata, Python interrompe l'esecuzione e la riprende quando l'utente fornisce un input.

```
Inp = input('STATEMENT')
```

```
num = input ("Enter number :")
print(num)
name1 = input("Enter name : ")
print(name1)

# Printing type of input value
print ("type of number", type(num))
print ("type of name", type(name1))
```

```
Enter number :455
455
Enter name : Cristina
Cristina
type of number <class 'str'>
type of name <class 'str'>
```

METODO `rawinput()`

Questo metodo funziona nelle versioni di Python precedenti alla 2.7.

Prende esattamente quello che viene digitato dall'utente, lo **converte in stringa** e poi lo ritorna alla variabile nella quale lo si vuole memorizzare.

ESEMPIO:

Si chiede all'utente lo username e, una volta inserito, questo viene stampato a video

PYTHON 3.6

```
username = input("Enter username:")
print("Username is: " + username)
```

```
Enter username:Cristina
Username is: Cristina
```

PYTHON 2.7

```
username = raw_input("Enter username:")
print("Username is: " + username)
```

```
Enter username:Cri
Username is: Cri
```

GESTIONE DEI FILE

GESTIONE DEI FILE IN PYTHON – *open()*

Quando vogliamo leggere o scrivere su un file, bisogna prima aprirlo. Una volta finito di lavorare con il file, questo deve essere chiuso in modo che le risorse impegnate con quel file vengano liberate.

In Python, quindi, un'operazione su file avviene in questo ordine:

1. **Apertura** file
2. **Lettura o scrittura** (eseguire un'operazione)
3. **Chiusura** del file.

Python possiede diverse funzioni per creare, leggere, aggiornare od eliminare i file.

APRIRE FILES IN PYTHON – *open()*

La funzione built-in di Python per aprire i file è ***open()***. Questa funzione **ritorna un oggetto file** che possiede metodi ed attributi per ottenere informazioni o per eseguire modifiche sul file.

La funzione *open()* accetta **due parametri**:

1. **Filename**
2. **Mode** (optional. Default = «rt»)

Per aprire un file in sola lettura è sufficiente specificare il nome del file:

```
f = open("demofile.txt")
```

Che è uguale a:

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

Siccome «r» (read) e «t» (text) sono valori di defaults, non c'è bisogno di specificarli.

Se il file non esiste si otterrà un errore.

MODE – modalità apertura file

Mode	Descrizione
r	Apre il file in lettura e ne restituisce il contenuto sotto forma di stringa . E' modalità di default insieme a «t»
w	Apre un file in scrittura , se il file non esiste lo crea. Se il file presenta del contenuto questo viene sovrascritto .
x	Apre un file creandolo , se il file esiste già l'operazione non ha successo.
a	Apre un file consentendo di accodare dei contenuti a quelli già presenti , nel caso in cui il file non esista viene creato ex novo.
t	Apre un file in modalità testo . Modalità di default insieme a «r».
b	Apre un file in modalità binaria . Utile per files non di testo.
+	Apre un file in lettura e scrittura per l'aggiornamento .

APRIRE e LEGGERE – *open()* e *read()*

Supponiamo di avere il seguente file, nella stessa cartella di Python:

```
demofile.txt  
Hello! Welcome to demofile.txt  
This file is for testing purposes.  
Good Luck!
```

Per aprire il file usiamo la funzione built-in **open()** che restituirà l'oggetto file sul quale chiameremo il metodo **read()**. Questo permetterà di leggere il contenuto del file.

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

```
Hello! Welcome to demofile.txt  
This file is for testing purposes.  
Good Luck!
```

Se il file si trova in un luogo differente, dovremo **specificare il percorso del file**:

```
f = open("D:\\myfiles\\welcome.txt", "r")  
print(f.read())
```

```
Welcome to this text file!  
This file is located in a folder named "myfiles", on the D drive.  
Good Luck!
```

LEGGERE PARTI DI UN FILE – *read(n_char) and readline()*

□ LEGGERE SOLO PARTI DI UN FILE

Di default il metodo `read()` ritorna il testo intero, ma si può specificare **quanti caratteri si vogliono ritornare**.

Read(*numero_caratteri*)

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

Hello

Vengono ritornati solo i primi 5 caratteri del file.

□ LEGGERE RIGHE – readline()

Il metodo `readline()` permette di **ritornare una riga del file**.

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

Hello! Welcome to demofile.txt

Chiamando il metodo `readline()` **due volte**, si leggeranno le **prime due righe del file**.

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

Hello! Welcome to demofile.txt
This file is for testing purposes.

Ciclando sulle righe del file, si può leggere l'intero file una riga alla volta.

```
f = open("demofile.txt", "r")
for x in f:
    print(x)
```

Hello! Welcome to demofile.txt
This file is for testing purposes.
Good Luck!

SCRIVERE SU UN FILE – *write()*

SCRIVERE SU UN FILE ESISTENTE

Per scrivere su un file già esistente, alla funzione `open()` va aggiunto uno dei parametri tra:

- «**a**» : Append. Aggiungerà i nuovi contenuti alla fine di quelli già presenti sul file.
- «**w**»: Write. Sovrascrive il contenuto già esistente.

Si applicherà poi il metodo `write()` che accetterà come parametro il contenuto da aggiungere

APRE IL FILE demofile2.txt E AGGIUNGE IL CONTENUTO

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

#open and read the file after the appending:
f = open("demofile2.txt", "r")
print(f.read())
```

```
C:\Users\My Name>python demo_file_append.py
Hello! Welcome to demofile2.txt
This file is for testing purposes.
Good Luck!Now the file has more content!
```

APRE IL FILE demofile3.txt E SOVRASCRIVE IL CONTENUTO

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

#open and read the file after the appending:
f = open("demofile3.txt", "r")
print(f.read())
```

Woops! I have deleted the content!

Questo metodo sovrascriverà l'intero file

CREARE UN NUOVO FILE – open()

Per creare un nuovo file, chiamare il metodo **open()** con uno dei seguenti parametri:

- **x – Create**: crea un file, ritorna un errore se il file già esiste
- **a – Append**: crea un file se il file specificato non esiste
- **w – Write**: crea un file se il file specificato non esiste.

CREAZIONE DI UN NUOVO FILE «myfile.txt»

`f = open("myfile.txt", "x")`

CREARE UN NUOVO FILE SE NON ESISTE

`f = open("myfile.txt", "w")`

With* `open("test.txt",'w',encoding = 'utf-8') as f:`
`f.write("my first file\n")`
`f.write("This file\n\n")`
`f.write("contains three lines\n")`

In questo esempio, se non è già esistente, viene creato un nuovo file chiamato test.txt nella directory corrente. **Se il file esiste già, viene sovrascritto.**

Le **nuove linee vanno inserite manualmente.**

*Per lo statement with, vedi slide seguente.

LO STATEMENT with

Lo statement with è usato per la **gestione delle risorse e delle eccezioni**. E' molto facile trovarlo quando si lavora con **flussi di file o risorse esterne**.

A volte un programma continua ad impegnare queste risorse anche se non sono più necessarie.

Questo tipo di problema è chiamato **MEMORY LEAK**: la memoria disponibile si riduce ogni volta che si crea e apre una nuova istanza di una data risorsa senza chiuderne una già esistente.

Il with, ad esempio, **assicura che il flusso di file non blocchi altri processi se si verifica un errore** (exception) bensì che il processo termini adeguatamente.

Semplifica la gestione delle eccezioni encapsulando le comuni preparazioni ed operazioni di pulizia. Inoltre, **chiude automaticamente il file quando si esce dal blocco del width**.

Gestire adeguatamente le risorse richiede una fase di setup, una di cancellazione ed una che preveda alcune operazioni di pulizia come la chiusura di un file, la liberazione di un lock o la chiusura di una connessione.

Se ci si dimentica di eseguire queste operazioni di pulizia, l'applicazione continuerà ad impegnare le risorse che non sono più necessarie, compromettendo importanti risorse di sistema come la memoria o la banda della rete.

Il metodo with permette di **riusare codice di setup, cancellazione e pulizia**.

Usare with permette dunque di **astrarre molta della logica di gestione delle risorse** e di evitare ogni volta la scrittura di un esplicito try...finally statement con codice di setup e distruzione delle risorse

Molte classi della libreria standard di Python supportano il with, come ad esempio il metodo close().

GESTIONE DEI FILE IN PYTHON – close()

Quando tutte le operazioni sul file sono state effettuate, il file deve essere adeguatamente chiuso per evitare problemi sul file stesso o sprechi di risorse come la RAM.

Chiudere il file libererà le risorse che erano legate a quel file. Questo viene eseguito con il metodo `close()`.

```
f = open("test.txt", encoding = 'utf-8')
# perform file operations
f.close()
```

Questo metodo non è del tutto sicuro perché, **se un'eccezione si verifica** mentre stiamo eseguendo operazioni sul file, il codice uscirà senza chiudere il file.

Un modo più sicuro è quello di usare il blocco `try...finally`.

In questo modo avremo la certezza che il file sarà adeguatamente chiuso anche nel caso dovesse verificarsi un'eccezione e il codice di bloccasse.

```
with open("test.txt", encoding = 'utf-8') as f:
    # perform file operations
```

```
try:
    f = open("test.txt", encoding = 'utf-8')
    # perform file operations
finally:
    f.close()
```

Il modo migliore, però, per chiudere un file è usare lo **statement with**.

Viene in questo modo assicurato che il file verrà chiuso quando si uscirà dal blocco di codice del `with`.

NON c'è bisogno di specificare la chiamata al metodo `close()` perché viene fatta in automatico.

Esempio di loop su file

```
with open('file.txt', 'r') as f:  
    print('*** for ***')  
    for line in f:  
        print(line, end='')  
    print('*** while ***')  
    f.seek(0)          # riporta indietro il cursore  
    while True:  
        line = f.readline()  
        if not line:  
            break  
        print(line, end = '')
```

CANCELLARE FILE – modulo OS – os.remove() e os.rmdir()

□ CANCELLARE UN FILE

Per cancellare un file, bisogna **importare il modulo OS** (modulo che incorpora diverse funzioni utili per far **interagire il programma con il sistema operativo del computer**).

Per la cancellazione ci si avvarrà del metodo

```
os.remove(file_name)
```

ESEMPIO: REMOVE THE FILE «demofile.txt»

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

□ CANCELLARE UNA CARTELLA

Per cancellare una cartella si usa il metodo **os.rmdir()**.
E' possibile eliminare **SOLO CARTELLE VUOTE**.

```
import os  
os.rmdir("myfolder")
```

CONTROLLARE SE UN FILE ESISTE

Per evitare di incorrere in un errore, si potrebbe **cercare l'esistenza del file prima di procedere con la sua cancellazione**.

Per verificare l'esistenza di un file si utilizza il metodo

```
os.path.exists(nomefile o path)
```

Per cancellare un file si usa il metodo

```
os.remove()
```

ESEMPIO: CONTROLLARE SE UN FILE ESISTE E POI CANCELLARLO

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

FILE

METHODS

Method	Description
<code>close()</code>	Chiude un file aperto. Non produce effetti se il file è chiuso.
<code>detach()</code>	Separates the underlying binary buffer from the TextIOBase and returns it.
<code>fileno()</code>	Ritorna un numero intero (descrittore del file).
<code>flush()</code>	Svuota il buffer di scrittura del file
<code>isatty()</code>	Returns True if the file stream is interactive.
<code>read(n)</code>	Legge al Massimo n caratteri dal file. Legge fino alla fine del file se il parametron è negative o non viene specificato.
<code>readable()</code>	Ritorna True se è possibile leggere dal flusso del file.
<code>readline(n=-1)</code>	Legge e ritorna una riga del file.
<code>readlines(n=-1)</code>	Legge e ritorna una lista di righe del file. Legge al Massimo n bytes/caratteri se specificato.
<code>seek(offset,from=SEEK_SET)</code>	Cambia la posizione del numero di bytes specificato nel parametron offset, rispetto al from (star, current, end).
<code>seekable()</code>	Returns True if the file stream supports random access. Ritorna True se il file consente il cambiamento della sua posizione.
<code>tell()</code>	Ritorna un intero che rappresenta la posizione corrente del file
<code>truncate(size=None)</code>	Ridimensiona il file alla dimensione specificata. Se la dimensione non è specificata, ridimensiona om base all'attuale posizione.
<code>writable()</code>	Ritorna True se il file può essere scritto oppure no
<code>write(s)</code>	Scrive la stringa s nel file e ritorna il numero di caratteri scritti.
<code>writelines(lines)</code>	Scrive una lista di righe nel file.

PYTHON – MySQL

Database PHP MySQL

Cos'è MySQL?

MySQL è un **DBMS (Database Management System)** di tipo **relazionale**, questo vuol dire che le informazioni archiviate e gestite sono in correlazione tra i loro.

Per interagire con questo DBMS si usa il **linguaggio SQL (Structured Query Language)**.

Le istruzioni SQL possono essere inviate manualmente al Database ad esempio attraverso riga di comando oppure generate con applicazioni e script scritti in diversi linguaggi, tra i quali Python.

MySQL è **un sistema di database utilizzato sul web**

MySQL è un sistema di database che **gira su un server**

MySQL è ideale sia **per applicazioni piccole che grandi**

MySQL è molto **veloce, affidabile e facile da usare**

MySQL utilizza **SQL standard**

MySQL si compila su una serie di piattaforme

MySQL può essere scaricato e utilizzato gratuitamente

MySQL è sviluppato, distribuito e supportato da Oracle Corporation

MySQL prende il nome dalla figlia del co-fondatore Monty Widenius: My

MySQL Database

Per prima cosa è necessario avere MySQL installato sul proprio computer, se si lavora in locale, o sul server.

Per scaricare gratuitamente un database
MySQL: <https://www.mysql.com/downloads/>
pip install mysql-connector

MySQL CONNECTOR

INSTALLARE MySQL Driver – MySQL CONNECTOR

Python non si interfaccia nativamente ai database MySQL e per comunicare con essi ha bisogno **dell'installazione di un driver MySQL** come «**MySQL Connector**».

Si raccomanda l'installazione con il **package manager Pip**, che si occupa anche della gestione delle dipendenze.

Partendo dalla **directory in cui è installato Python**, digitare il seguente comando:

```
C:\Python>python -m pip install mysql-connector-python
```

TESTARE MySQL CONNECTOR

Per verificare il funzionamento del connettore è sufficiente **importarlo** come ogni modulo Python. Creiamo una pagina Python con il seguente contenuto:

```
import mysql.connector
```

Se il connettore è stato correttamente installato, l'interprete si limiterà a **non restituire alcun errore**.

NOTA: I file “.py” dai quali lanciare comandi verso il database **NON DEVONO ESSERE** salvati come **mysql.py** o **MySQL.py**, altrimenti il sistema cercherà di caricare il modulo da questi ultimi e la sua importazione fallirà.

CREARE CONNESSIONI

Una volta installato il connettore, si può procedere alla connessione con il DBMS, punto di partenza per l'interazione con esso.

Serviranno il **nome utente** e la **password** del database con il quale si vuole interagire.

CODICE D'ESEMPIO PER TEST DI CONNESSIONE

```
# Test di funzionamento del connettore MySQL

import mysql.connector

# Test di connessione a MySQL

connessione = mysql.connector.connect(
    # Parametri per la connessione
    host="localhost",
    user="nome-utente",
    password="password"
)
# Stampa dell'handle di connessione
print(connessione)
```

Il risultato dovrebbe essere:

```
Python 3.9.4 (tags/v3.9.4:1f2e308, Apr  6 2021, 13:40:21) [MSC v.1928 64
bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
=====
      RESTART: C:\Python\Progetti\mysql\test.py =====
=====
<mysql.connector.connection.MySQLConnection object at 0x000002187A630AC0>
>>> |
```

Dopo l'importazione del modulo per la connessione, il **metodo connect()** introduce i **parametri di connessione**:

- host: indirizzo del sistema su cui si trova il database.
- utente MySQL: username dell'utente MySQL
- password

APPROFONDIMENTI MYSQL

approfondimenti

* Python 2

sezione Approfondimento Mysql 1

INSERIMENTO DEI DATI NELLA TABELLA – INSERT INTO

Una volta creata una tabella con i relativi campi, è possibile popolarla con dei dati.

In SQL si usa il comando

INSERT INTO *nome_tabella*

Seguito dai **nomi dei campi separati da virgola** e delimitati da parentesi tonde.

I **valori** da assegnare ai campi vanno introdotti dalla keyword **VALUES**, anch'essi tra parentesi e separati da una virgola.

Istruzione **execute()** accetta la variabile istruzione e la tuple valori che contiene i valori da inserire.

Metodo **commit()**: affinchè l'inserimento dei valori diventi definitivo, è necessaria la chiamata al metodo commit() che **applica i cambiamenti alla tabella**. Senza l'istruzione connessione.commit() i cambiamenti non verrebbero effettuati.

Rowcount() consente il **conteggio dei record** coinvolti dall'istruzione.

```
# Importazione del modulo MySQL
import mysql.connector

# Connessione a MySQL
connessione = mysql.connector.connect(
# Parametri per la connessione
    host="localhost",
    user=<em>nome-utente</em>,
    password=<em>password</em>""
    db="anagrafiche" )

# Generazione del cursore
cursore = connessione.cursor()

# Comando SQL per l'inserimento del record
istruzione = "INSERT INTO nominativi (nome, cognome, parentela) VALUES (%s, %s, %s)"
valori = ("Rick", "Sanchez", "nonno")

# Esecuzione dell'istruzione
cursore.execute(istruzione, valori)

# Applicazione delle modifiche
connessione.commit()

# Conteggio dei record inseriti
print(cursore.rowcount)
```

ESCAPING DEI VALORI – prevenire l’SQL Injection

Nel codice precedente i valori dei campi non vengono inseriti immediatamente dopo la keyword VALUES ma viene previsto un passaggio intermedio.

Avremmo potuto avere un codice di questo tipo:

```
istruzione = "INSERT INTO nominativi (nome, cognome, parentela) VALUES ("Rick", "Sanchez", "Nonno")"
```

Sia questa sintassi che quella vista nell'esempio precedente sono corrette ma si preferisce la sostituzione dei valori con dei placeholder («%\$») e la loro introduzione come elementi di una tuple.

```
istruzione = "INSERT INTO nominativi (nome, cognome, parentela) VALUES (%s, %s, %s)"  
valori = ("Rick", "Sanchez", "nonno")
```

Il motivo è la **necessità di prevenire attacchi** basati sull'**SQL Injection** che consiste nell'iniettare del codice malevolo in un'istruzione SQL con lo scopo di violare un database o causare altri danni come l'inserimento di dati, la loro modifica o cancellazione.

I placeholder consentono l'**ESCAPING DEI VALORI**: eliminazione dai parametri di input di elementi di ambiguità che potrebbero generare dei comportamenti imprevisti.

INSERIRE RECORD MULTIPLI – executemany()

E' possibile creare script che permettono l'inserimento simultaneo di più record.

Per farlo si usa il metodo **executemany()** il cui secondo parametro è una lista di tuple ognuna delle quali presenta tanti elementi quanti sono i placeholder introdotti da VALUES.

In molti casi, l'executemany() itera sulla sequenza di parametri, passando ad ogni iterazione i correnti parametri al metodo execute().

Cursor.executemany(operation, sequenza_di_params)

```
# Comando SQL per l'inserimento dei record
istruzione = "INSERT INTO nominativi (nome, cognome, parentela, occupazione) VALUES (%s, %s, %s, %s)"
valori = [
    ('Morty', 'Smith', 'nipote', 'studente'),
    ('Summer', 'Smith', 'nipote', 'studentessa'),
    ('Beth', 'Sanchez in Smith', 'figlia', 'veterinaria'),
    ('Jerry', 'Smith', 'genero', 'disoccupato') ]

# Esecuzione dell'istruzione
cursore.executemany(istruzione, valori)

# Applicazione delle modifiche
connessione.commit()

# Conteggio dei record inseriti
print(cursore.rowcount)
```

OTTENERE L'ID DEL RECORD INSERITO – lastrowid

Si può ottenere l'id del record appena inserito la proprietà **lastrowid** l'oggetto cursore.

Se si inseriscono più records, viene ritornato l'id dell'ultimo record.

se si utilizza **lastrowid** con **executemany** verrà ritornato 0

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="myusername",
  password="mypassword",
  database="mydatabase"
)

mycursor = mydb.cursor()

sql = "INSERT INTO customers (name, address) VALUES
(%s, %s)"
val = ("Michelle", "Blue Village")
mycursor.execute(sql, val)

mydb.commit()

print("1 record inserted, ID:", mycursor.lastrowid)
```

1 record inserted, ID: 15

AGGIORNAMENTO DEI RECORD - UPDATE

A volte si ha la necessità di dover **modificare i valori di uno o più campi.**

UPDATE table_name SET field_name.....

Per farlo si usa il comando SQL **UPDATE** seguito dal **nome della tabella** e dalla keyword **SET** che introduce i campi da aggiornare e come aggiornarli.

La clausola **WHERE** permette di individuare con maggiore precisione il o i record che dovranno subire la modifica.

Quando si utilizza il comando **UPDATE** per aggiornare dei record specifici è assolutamente necessario ricordarsi di utilizzare la clausola **WHERE**, altrimenti i cambiamenti richiesti verranno applicati indiscriminatamente a tutti record.

```
# Comando per l'aggiornamento del record  
istruzione = "UPDATE nominativi SET occupazione = %s  
WHERE id_nominativi = %s"  
valori = ("scienziato", 1)
```

```
# Esecuzione dell'istruzione  
cursore.execute(istruzione, valori)
```

```
# Applicazione delle modifiche  
connessione.commit()
```

SELEZIONE DEI RECORD

Una volta popolata/e la/e tabella/e del database, è possibile estrarre i dati totalmente o sulla base di determinati criteri.

Il comando SQL di riferimento è il **SELECT** che consente di specificare la tabella dalla quale estrapolare i dati tramite la clausola **FROM**.

Altre clausole, come la **WHERE**, consentono di affinare le interrogazioni al database (query) aumentando la precisione degli output.

In Python, le query basate sul comando **SELECT** non richiedono l'utilizzo del metodo **commit()** in quanto non applicano alcuna modifica a carico dei dati o della loro struttura.

□ Metodo **fetchall()**

recupera tutti i record dall'ultima istruzione eseguita

□ Metodo **fetchone()**

Recupera la prima riga dei risultati.

□ Metodo **fetchmany()**

Ritorna il numero di record passato come parametro.

```
per estrarre i dati come dictionary e non come tupla:  
# Generazione del cursore  
cursore = connessione.cursor(dictionary=True)
```

SELEZIONE DEI RECORD – fetchall()

METODO FETCHALL()

Per selezionare tutti i campi si sostituisce al dichiarazione dei nomi dei campi con un asterisco.

Avremo quindi un

```
SELECT * FROM table_name.
```

Nel nostro esempio il

```
SELECT * FROM nominativi
```

Sostituisce

```
SELECT id_nominativi, nome, cognome, parentela,  
occupazione FROM nominativi
```

Una volta passata la query al metodo execute(), è possibile estrarre i record tramite il metodo **fetchall()** che non richiede parametri.

Fetchall genera una **lista di tuple** dove ognuna di esse corrisponde ad un document.

La lista può poi essere passata in un ciclo for per restituire i record in output.

```
# Importazione del modulo MySQL  
import mysql.connector  
  
# Connessione a MySQL  
connessione = mysql.connector.connect(  
  
# Parametri per la connessione host="localhost", user=<em>nome-  
utente</em>, password=<em>password</em>" db="anagrafiche" )  
  
# Generazione del cursore  
cursore = connessione.cursor()  
  
# Comando SQL per l'estrazione dei record  
cursore.execute("SELECT * FROM nominativi")  
query  
# Visualizzazione dei record  
risultati = cursore.fetchall()  
  
for i in risultati: print(i)  
  
(1, 'Rick', 'Sanchez', 'nonno', 'scienziato')  
(2, 'Morty', 'Smith', 'nipote', 'studente')  
(3, 'Summer', 'Smith', 'nipote', 'studentessa')  
(4, 'Beth', 'Sanchez in Smith', 'figlia', 'veterinaria')  
(5, 'Jerry', 'Smith', 'genero', 'disoccupato')
```

SELEZIONE DEI RECORD – fetchall()

Per selezionare solo alcuni dei campi di una tabella, si usa il **SELECT** seguito dai nomi dei campi della tabella.

ESEMPIO: selezionare solo i campi nome e cognome:

```
# Importazione del modulo MySQL
import mysql.connector

# Connessione a MySQL
connessione = mysql.connector.connect(
    # Parametri per la connessione host="localhost", user="<em>nome-utente</em>", password="<em>password</em>"
    db="anagrafiche" )

# Generazione del cursore
cursore = connessione.cursor()

# Comando SQL per l'estrazione dei record
cursore.execute("SELECT nome, cognome FROM nominativi")

# Visualizzazione dei record
risultati = cursore.fetchall()

for i in risultati: print(i)
```

ESTRAZIONE DI UN SINGOLO RECORD – fetchone()

Quando si ha la necessità di **estrarre UN SOLO RECORD**, Python fornisce il metodo **fetchone()** che restituisce una singola tuple (o nessuna nel caso in cui la query non produca alcun risultato).

Non essendo presente nella query alcuna clausola WHERE che specifichi il record da selezionare, viene restituito il primo record registrato in tabella.

Avendo estratto un singolo record, non serve il ciclo for per la stampa a video.

Il **fetchone()** può essere richiamato più volte all'interno dello stesso script.

Ad ogni chiamata però, il cursore generato con il metodo **cursor()** viene ricollocato in modo da occupare la posizione successiva.

Quindi la seconda chiamata a **fetchone()** restituirebbe il record con id «2» o quello successivo ancora disponibile nel caso sia stato rimosso.

```
# Importazione del modulo MySQL
import mysql.connector

# Connessione a MySQL
connessione = mysql.connector.connect(
    host="localhost", user="nome-utente", password="password",
    db="anagrafiche" )

# Generazione del cursore
cursore = connessione.cursor()

# Comando SQL per l'estrazione dei record
cursore.execute("SELECT * FROM nominativi")

# Visualizzazione del record
risultato = cursore.fetchone()

print(risultato)

(l, 'Rick', 'Sanchez', 'nonno', 'scienziato')
```

RITORNARE UN DICTIONARY E NON UNA TUPLA

con la clausola

```
mycursor = connessione.cursor(dictionary=True) # ritorna un dictionary e non una tuple
```

il fetch o fetchall ecc... ritorna un dictionary e quindi sarà possibile richiedere dal record solamente un determinato campo

```
for row in mycursor.fetchall():
    print(row['nome'])
```

ESTRAZIONE DI RISULTATI CON CLAUSOLA WHERE

Le operazioni basate sul comando SELECT possono presentare **clausole che permettono di specificare i record che devono essere coinvolti.**

Nell'esempio, la clausola WHERE determina l'estrazione dei soli record con il valore del campo cognome = Smith.

Il metodo fetchall() estrarrà solo i record in cui il campo specificato ha un determinato valore.

```
# Comando SQL per l'estrazione dei record
istruzione = "SELECT * FROM nominativi WHERE cognome = %s"
valori = ("Smith", )

# Esecuzione del comando SQL
cursore.execute(istruzione, valori)

# Visualizzazione del record
risultati = cursore.fetchall()
for i in risultati: print(i)
```

CARATTERI WILDCARD

Si possono anche selezionare i records che iniziano, contengono o finiscono con una data lettera o frase.

Per rappresentare il carattere jolly si usa **%**.

Nell'esempio a fianco si vogliono selezionare i record che contengono la parola «way».

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="myusername",
  password="mypassword",
  database="mydatabase"
)

mycursor = mydb.cursor()

sql = "SELECT * FROM customers WHERE
address Like '%way%'""

mycursor.execute(sql)

myresult = mycursor.fetchall()

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

ESTRAZIONE DEI RECORD – fetchmany()

Se si ha l'esigenza di **estrarre uno specifico numero di record**, anche senza l'utilizzo di una clausola che specifichi dei criteri di selezione, è possibile fare ricorso al metodo **fetchmany()** che accetta come argomento il cosiddetto **parametro size**, un valore numerico intero che rappresenta la quantità di record che devono essere estratti.

Nell'esempio proposto i record estratti sono 2, contenuti in altrettante tuple i cui elementi vengono stampati a video tramite un ciclo.

Nel caso non venga estratto alcun record, il metodo ritorna una lista vuota.

```
# Comando SQL per l'estrazione dei record  
cursore.execute("SELECT * FROM nominativi")
```

```
# Visualizzazione dei record  
risultati = cursore.fetchmany(2)  
for i in risultati: print(i)
```

ORDINARE I RISULTATI – Order By

Si usa lo statement **ORDER BY** per ordinare il risultato di una query in modo ascendente o discendente (se non specificato l'ordine ascendente è quello di default).

ORDINARE I RISULTATI ALFABETICAMENTE PER NOME.

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="myusername",
  password="mypassword",
  database="mydatabase"
)

mycursor = mydb.cursor()

sql = "SELECT * FROM customers ORDER BY name"

mycursor.execute(sql)

myresult = mycursor.fetchall()

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

```
(3, 'Amy', 'Apple st 652')
(11, 'Ben', 'Park Lane 38')
(7, 'Betty', 'Green Grass 1')
(13, 'Chuck', 'Main Road 989')
(4, 'Hannah', 'Mountain 21')
(1, 'John', 'Highway 21')
(5, 'Michael', 'Valley 345')
(15, 'Michelle', 'Blue Village') (2, 'Peter', 'Lowstreet 27')
(8, 'Richard', 'Sky st 331')
(6, 'Sandy', 'Ocean blvd 2')
(9, 'Susan', 'One way 98')
(10, 'Vicky', 'Yellow Garden 2')
(14, 'Viola', 'Sideway 1633')
(12, 'William', 'Central st 954')
```

ORDINARE I RISULTATI – ORDER BY DESC

Per ordinare in ordine discendente si usa la parola chiave **DESC**

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="myusername",
  password="mypassword",
  database="mydatabase"
)

mycursor = mydb.cursor()

sql = "SELECT * FROM customers ORDER BY name DESC"

mycursor.execute(sql)

myresult = mycursor.fetchall()

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

```
(12, 'William', 'Central st 954') (14, 'Viola', 'Sideway 1633')
(10, 'Vicky', 'Yellow Garden 2')
(9, 'Susan', 'One way 98')
(6, 'Sandy', 'Ocean blvd 2')
(8, 'Richard', 'Sky st 331')
(2, 'Peter', 'Lowstreet 27')
(15, 'Michelle', 'Blue Village') (5, 'Michael', 'Valley 345')
(1, 'John', 'Highway 21')
(4, 'Hannah', 'Mountain 21')
(13, 'Chuck', 'Main Road 989')
(7, 'Betty', 'Green Grass 1')
(11, 'Ben', 'Park Lane 38')
(3, 'Amy', 'Apple st 652')
```

CANCELLARE RECORD – DELETE FROM

Ricordarsi sempre che **le operazioni di cancellazione su record, tabelle e database sono IRREVERSIBILI**, quindi se non si hanno copie di backup aggiornate, i dati cancellati andranno perduti definitivamente.

CANCELLAZIONE DI RECORDS

Per effettuare la cancellazione di records, si usa il commando SQL **DELETE FROM**.

SINTASSI

DELETE FROM *table_name* **WHERE**

L'esempio mostra la **cancellazione dalla tabella “nominativi”** soltanto dei record in cui il valore del campo “parentela” corrisponde alla stringa “nipote”.

Le **modifiche vengono confermate tramite il metodo commit() privo di argomenti.**

SE non viene inserita la clausula WHERE, tutti i record verranno cancellati.

```
# Importazione del modulo MySQL
import mysql.connector

# Connessione a MySQL
connessione = mysql.connector.connect(
    host="localhost",
    user="nome-utente",
    password="password"
    db="anagrafiche" )

# Generazione del cursore
cursore = connessione.cursor()

# Comando SQL per la cancellazione dei record
istruzione = "DELETE FROM nominativi WHERE parentela = %s"
valori = ("nipote", )

# Esecuzione dell'istruzione
cursore.execute(istruzione, valori)

# Applicazione delle modifiche
connessione.commit()

# Conteggio dei record coinvolti dalla cancellazione
print(cursore.rowcount) #ritorna 2
```

ESEMPIO DI LETTURA E INSERIMENTO

```
import mysql.connector
inp = input('vuoi inserire o leggere? (I/R):')
try:
    connessione = mysql.connector.connect(
        host="localhost",
        user="root",
        password="",
        db="test"
    )
    # Stampa dell'handle di connessione print(connessione)
    if inp == "I":
        query = "INSERT into servizi (nome, descrizione) values (%s, %s)"
        mycursor = connessione.cursor(dictionary=True)
        mycursor.execute(query, ['sviluppo python', 'sviluppo in linguaggio python'])
        connessione.commit()
        print('record inserito', mycursor.lastrowid)
    elif inp == "R":
        query = "SELECT * FROM servizi"
        mycursor = connessione.cursor(dictionary=True) # ritorna un dictionary e non una tuple
        mycursor.execute(query)
        for row in mycursor.fetchall():
            print(row['nome'])
    else:
        print('operazione non prevista!')
except Exception as ex:
    print(ex.__str__())
finally:
    print('operazione conclusa con successo')
```