

# Python

# Perchè Python?

- Python è un linguaggio più semplice e più **espressivo** (con meno codice possiamo far fare al computer cose più complesse)
- Python è un linguaggio oggi utilizzatissimo in tantissimi contesti
  - data science
  - sviluppo web
  - automazione di processi
  - ....
- c'è una comunità molto attiva intorno a python!
- ci sono tantissime librerie gratuite che svolgono per voi programmatori compiti complessi.

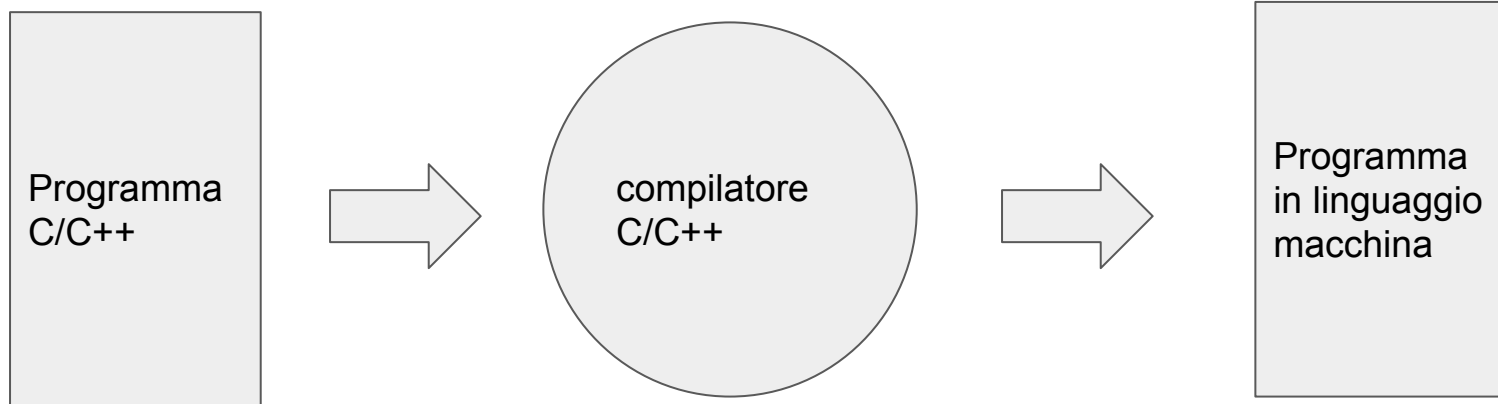
# Interpretazione ≠ compilazione

Python è un linguaggio interpretato, C++ è invece un linguaggio compilato.

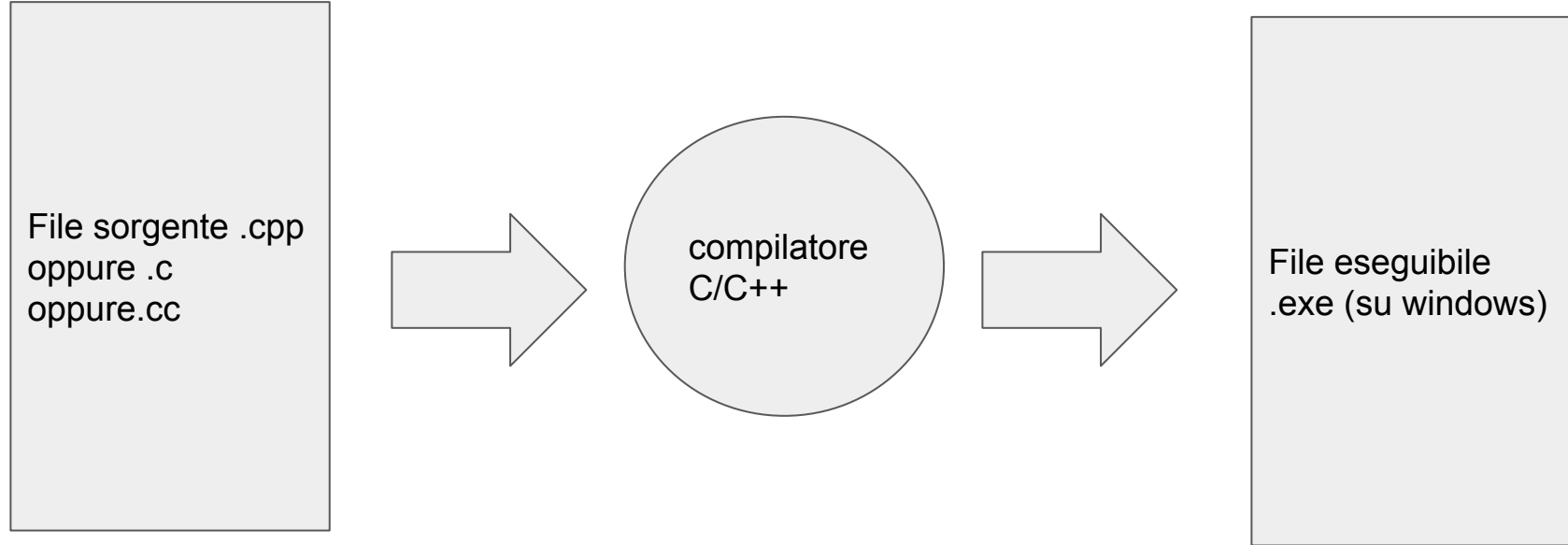
Che differenza c'è tra l'uno e l'altro?

# Compilazione

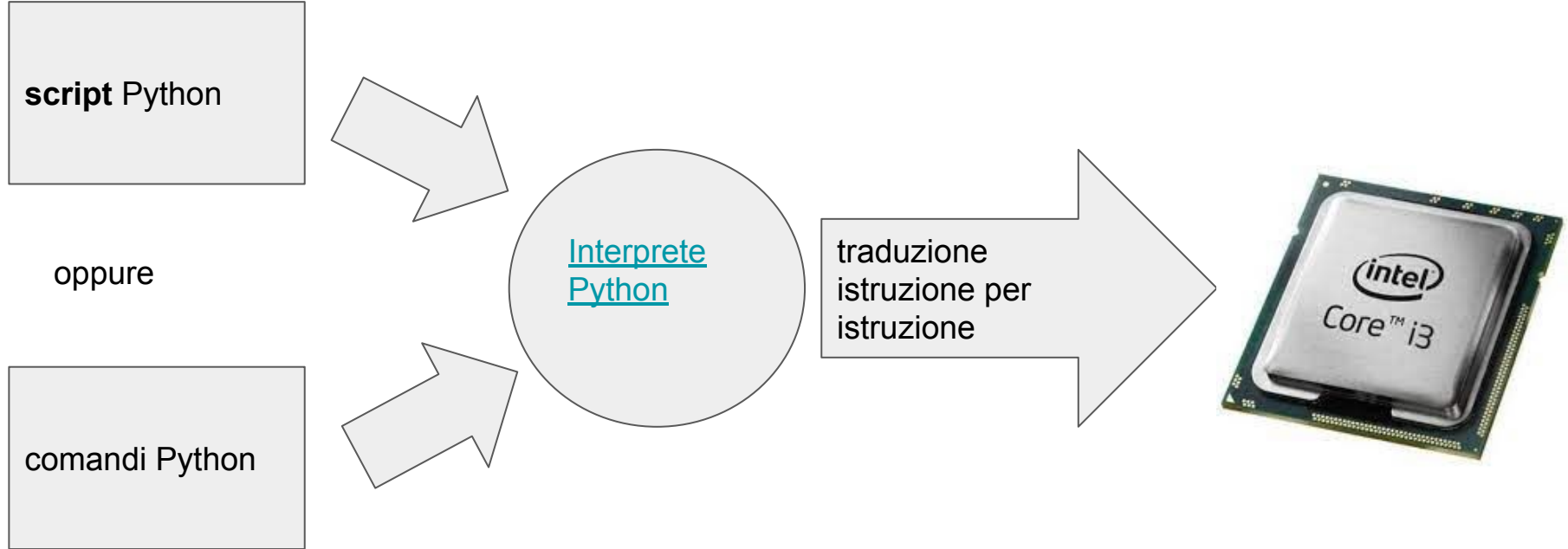
La compilazione è il processo tramite il quale il nostro programma in C/C++ viene tradotto effettivamente in **linguaggio macchina**, quel linguaggio veramente utilizzato dal processore del nostro computer per fare i calcoli. (Linguaggio molto scomodo e difficile da imparare per noi). Il procedimento di compilazione viene svolto da un programma chiamato **compilatore**.



# Compilazione



# Interpretazione



# Hands On!!!

Il miglior modo di imparare è sporcandosi le mani! pertanto cominciamo dando qualche semplice **comando** all'interprete



L'interprete python non interpreta soltanto degli **script** ma anche i nostri comandi uno per uno. Se infatti lo apriamo notiamo che comparirà uno schermo nero con le tre frecce > dove possiamo scrivere il nostro comando

```
Python 3.9.2 (tags/v3.9.2:1a79785, Feb 19 2021, 13:44:55) [MSC v.1928 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```



# Il nostro primo comando python

```
Python 3.9.2 (tags/v3.9.2:1a79785, Feb 19 2021, 13:44:55) [MSC v.1928 64 b  
Type "help", "copyright", "credits" or "license" for more information.  
>>> print("ciao mondo")  
ciao mondo  
>>>
```

# Le operazioni aritmetiche in Python

```
>>> 2 + 2
4
```

```
>>> 1 / 2
0.5
```

La divisione da  
sempre un risultato  
**float**

```
>>> 16 % 2
0
>>> 15 % 2
1
```

l'operatore di resto  
c'è sempre e  
comunque

# Le variabili in Python

In python le variabili **non si dichiarano**, creiamo una variabile nel momento in cui le dobbiamo assegnare un valore.

**Non dobbiamo specificare il tipo!**

Ci pensa l'interprete a capire di che tipo è! L'interprete inferisce in modo automatico il tipo!

# Le variabili in Python

```
>>> a = 10
>>> print(a)
10
>>> a = 'M'
>>> print(a)
M
```

# Output

La funzione print ci serve per stampare a video qualcosa! un po come cout

```
>>> a = 10
>>> b = 24
>>> print("la variabile a vale",a," la variabile b vale", b)
la variabile a vale 10  la variabile b vale 24
```

possiamo stampare quante cose vogliamo, le separiamo tra di loro con la virgola ,

# Output

La funzione print ci serve per stampare a video qualcosa! un po come cout

```
>>> a = 10
>>> b = 24
>>> print("la variabile a vale", a, " la variabile b vale", b)
la variabile a vale 10 la variabile b vale 24
```



è testo normale siccome è  
racchiuso dai doppi apici



il valore della  
variabile  
corrispondente

# Input

La funzione input ci serve per prendere in input qualcosa dall'utente! Un po come cin>> in C++

```
>>> x = int(input("inserisci un numero intero\n"))
inserisci un numero intero
12
>>> print(x)
12
```

# Input

siccome quello che inseriamo da tastiera per Python è sempre **del testo** se noi vogliamo che x sia un numero intero dobbiamo forzare la conversione ad intero

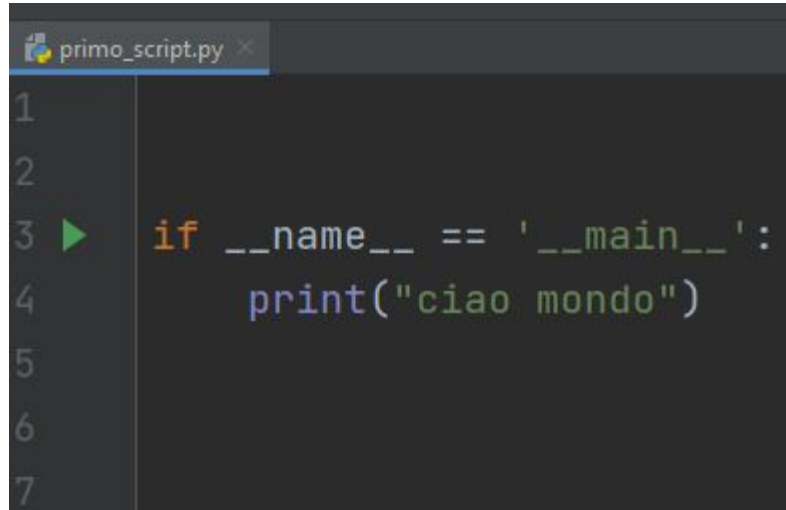
dentro alle parentesi tonde mettiamo tra doppi apici quello che vogliamo sia scritto all'utente per fargli inserire un numero

```
>>> x = int(input("inserisci un numero intero\n"))
inserisci un numero intero
12
>>> print(x)
12
```

vedete che mi mostra la scritta che ho messo tra gli apici, e mi chiede di inserire un intero.



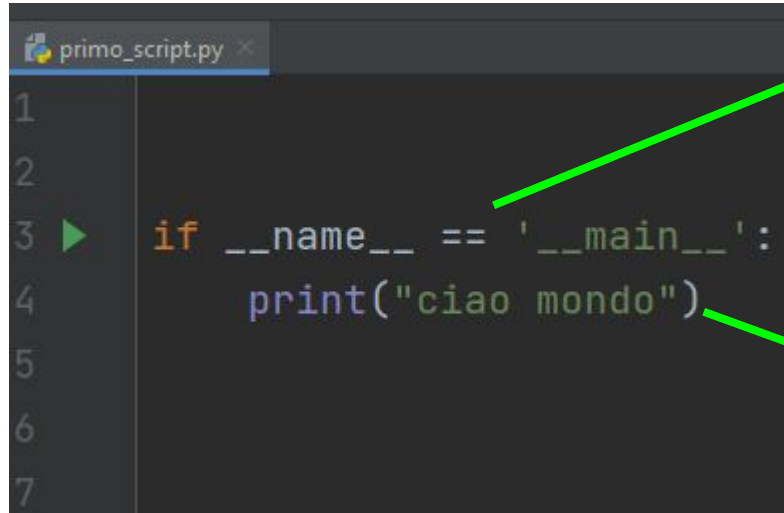
# Scriviamo il nostro primo script Python



A screenshot of a code editor window titled "primo\_script.py". The editor has a dark background with light-colored text. On the left side, there is a vertical line of numbers from 1 to 7, representing line numbers. The code is written in Python and consists of two lines: an if statement and a print statement. The if statement is on line 3 and the print statement is on line 4. The code is as follows:

```
1  
2  
3  if __name__ == '__main__':  
4      print("ciao mondo")  
5  
6  
7
```

# Il nostro primo script Python



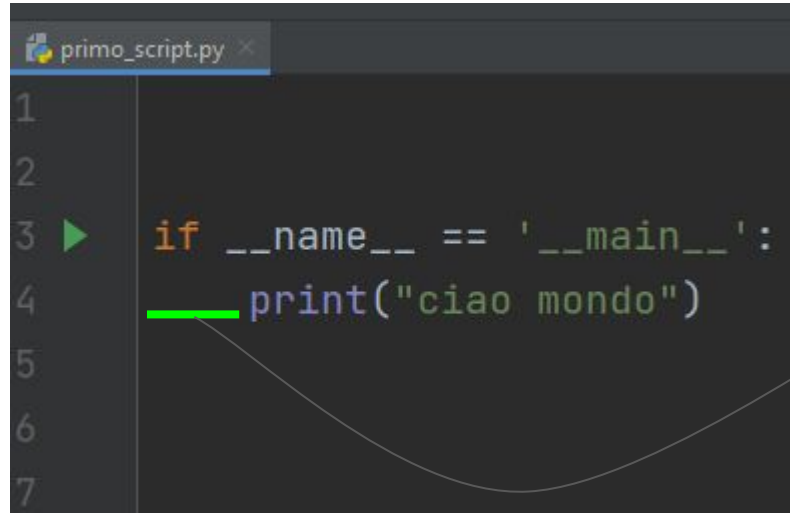
```
primo_script.py x
1
2
3 ► if __name__ == '__main__':
4     print("ciao mondo")
5
6
7
```

quello che in C++  
era: **int main()**

**cout<<"ciao mondo";**

# Indentazione

In python i blocchi di codice non si racchiudono tra parentesi graffe ma si usa l'indentazione! L'indentazione è fondamentale!!!



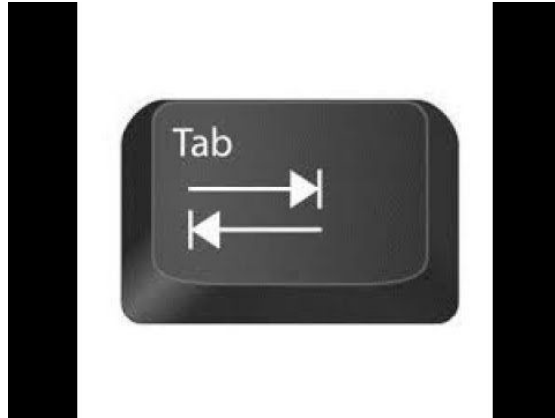
A screenshot of a code editor window titled 'primo\_script.py'. The editor shows a Python script with line numbers 1 through 7 on the left. Line 3 contains the code `if __name__ == '__main__':` and line 4 contains `print("ciao mondo")`. The `print` line is indented under the `if` line. A green horizontal line is drawn under the `print` statement. A curved line points from this green line to the text 'UNA TABULAZIONE' on the right.

```
1
2
3  ▶ if __name__ == '__main__':
4    print("ciao mondo")
5
6
7
```

UNA  
TABULAZIONE

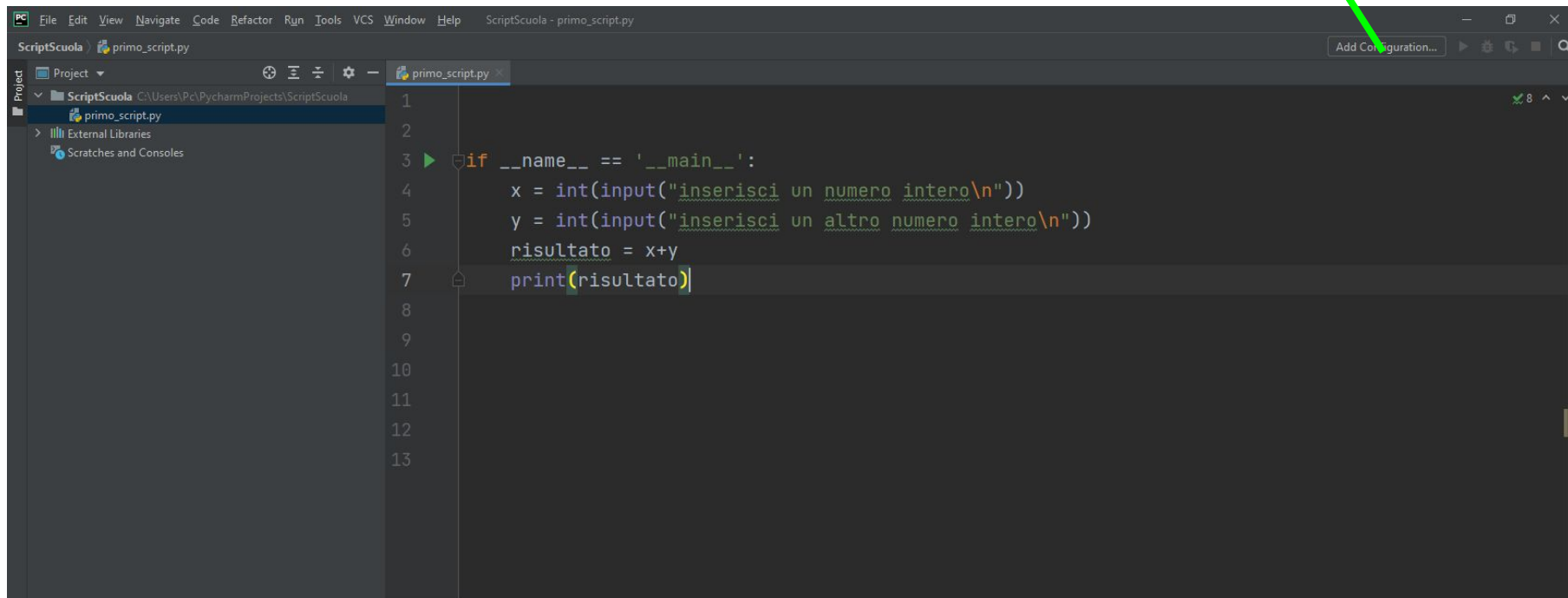
# Indentazione

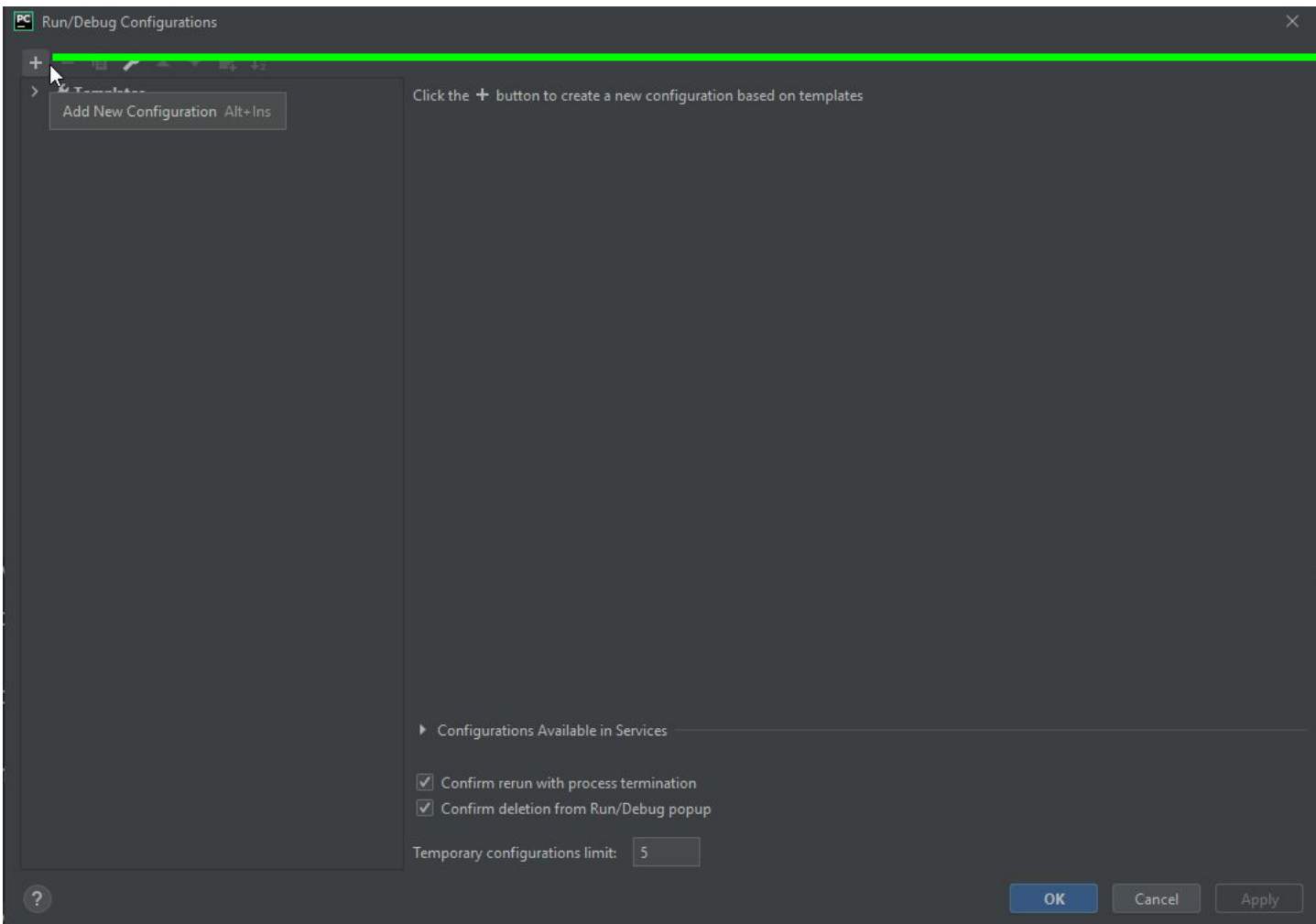
**usate il tasto tab! non gli spazi!**



# Eseguiamo il nostro primo script Python

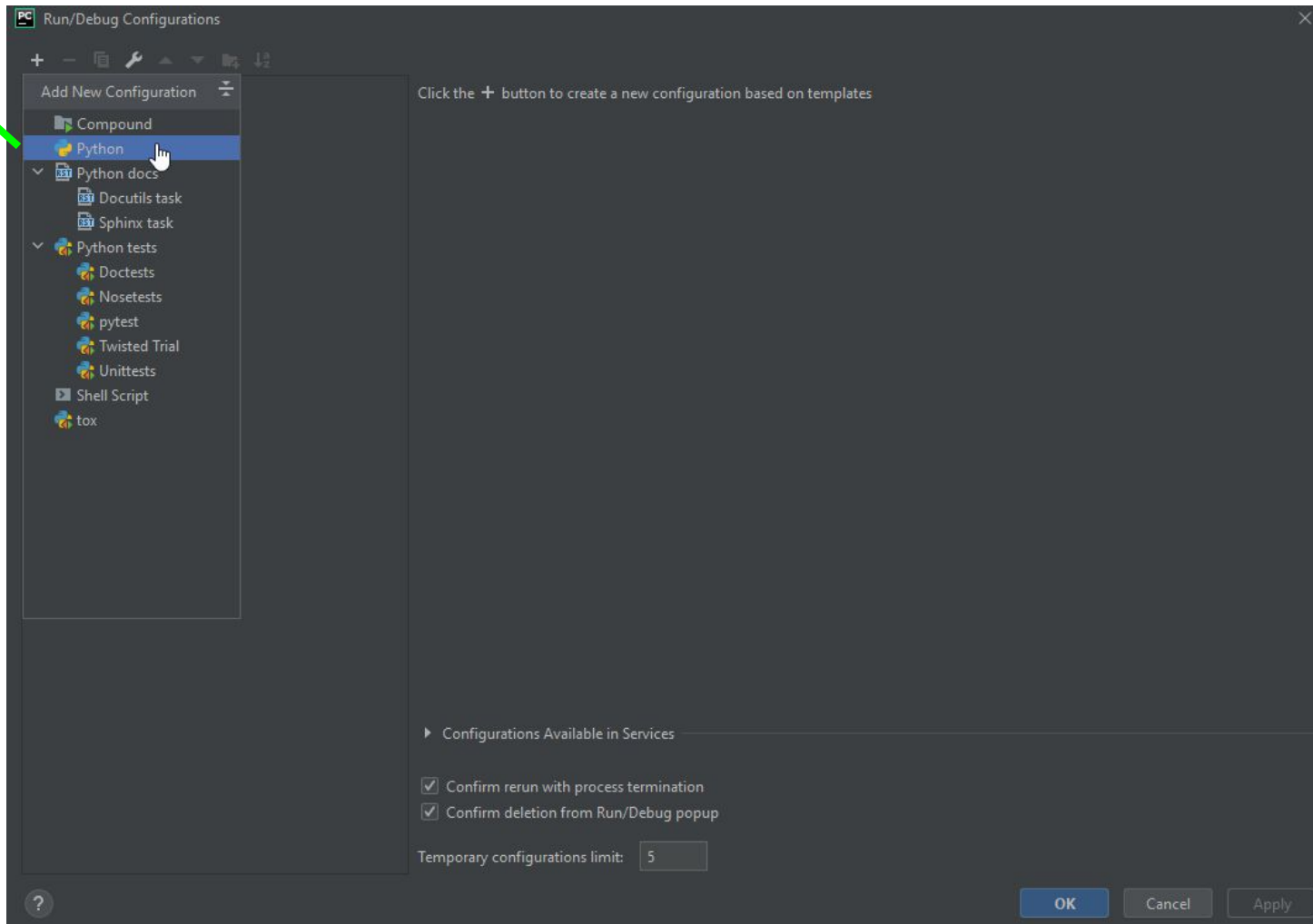
premi qui

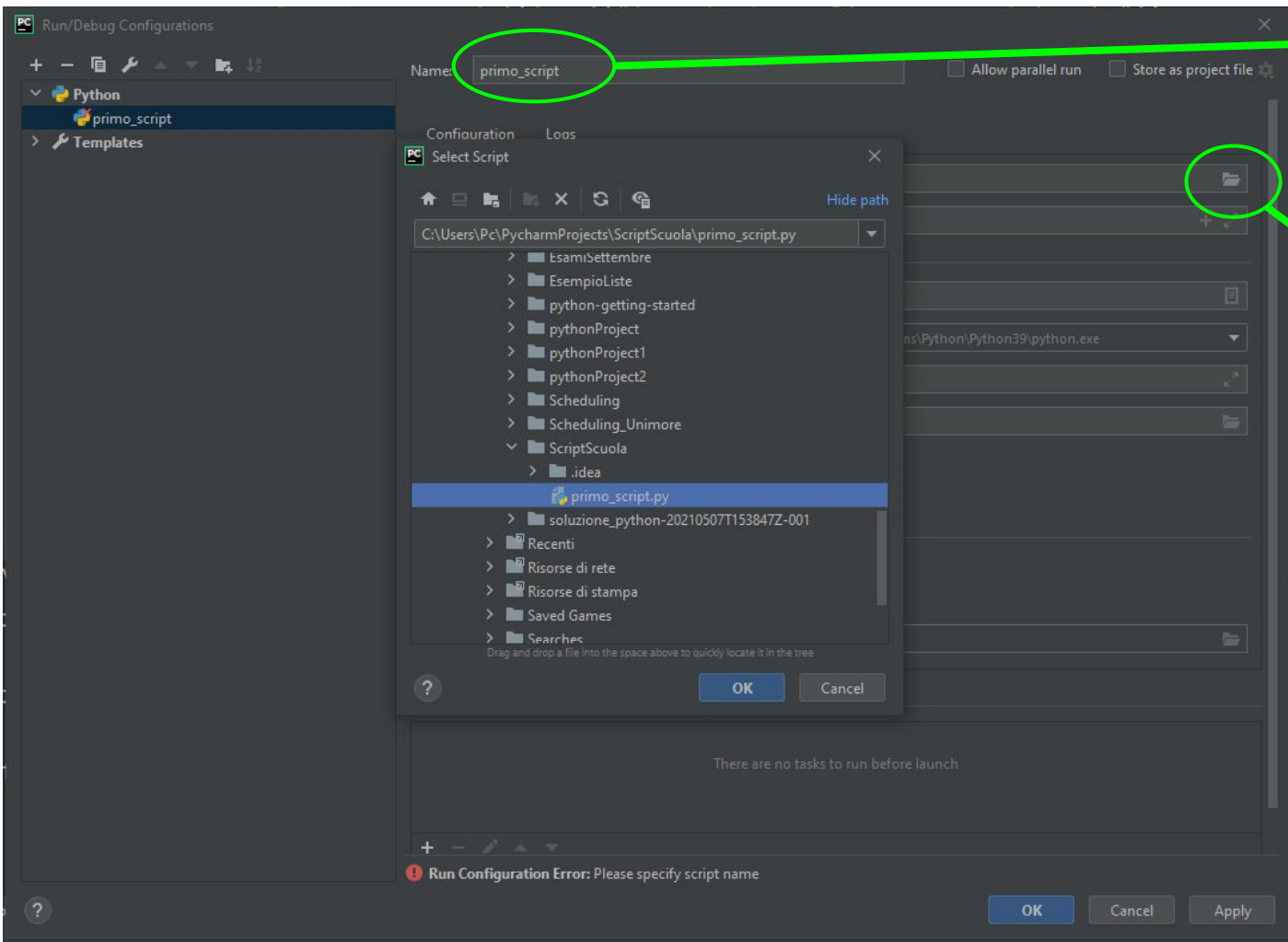




clicca qui

scegliamo  
Python





scegliamo il nome  
della nostra  
configurazione

premiamo qui per  
scegliere il file che  
vogliamo sia  
interpretato



Scriviamo un semplice programma in cui chiediamo in input un numero e mostriamo il suo successivo

```
inserisci un numero intero  
10  
11
```

secondo\_script.py x

1

2

3

▶ if \_\_name\_\_ == '\_\_main\_\_':

4

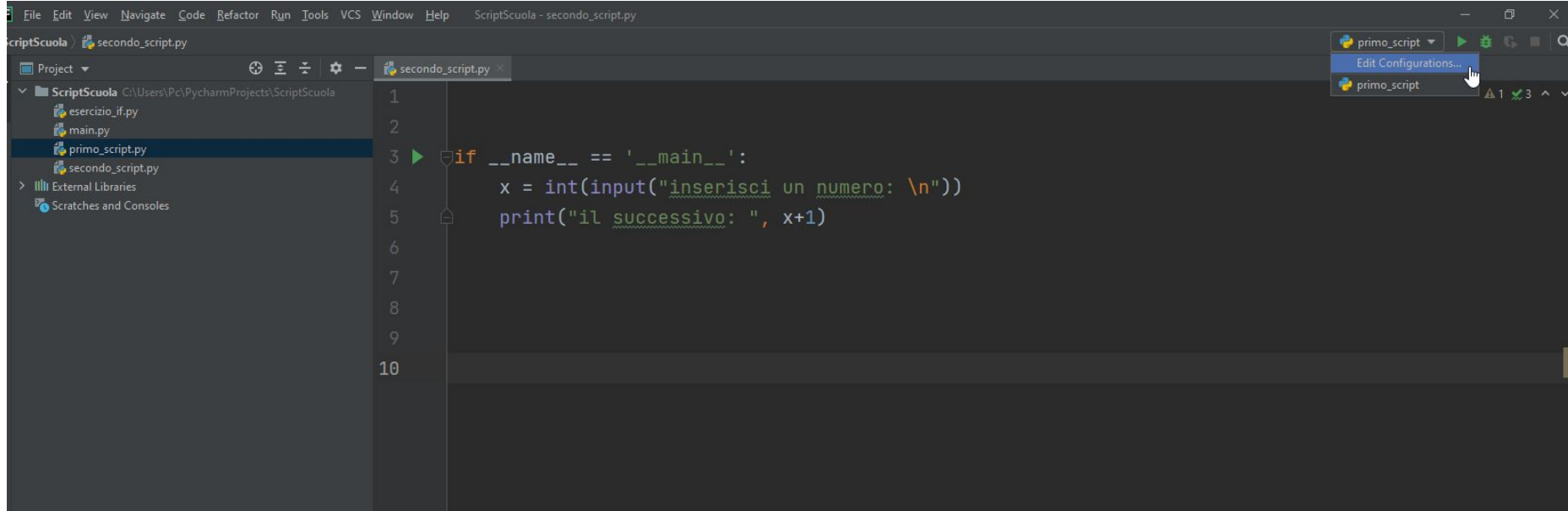
x = int(input("inserisci un numero: \n"))

5

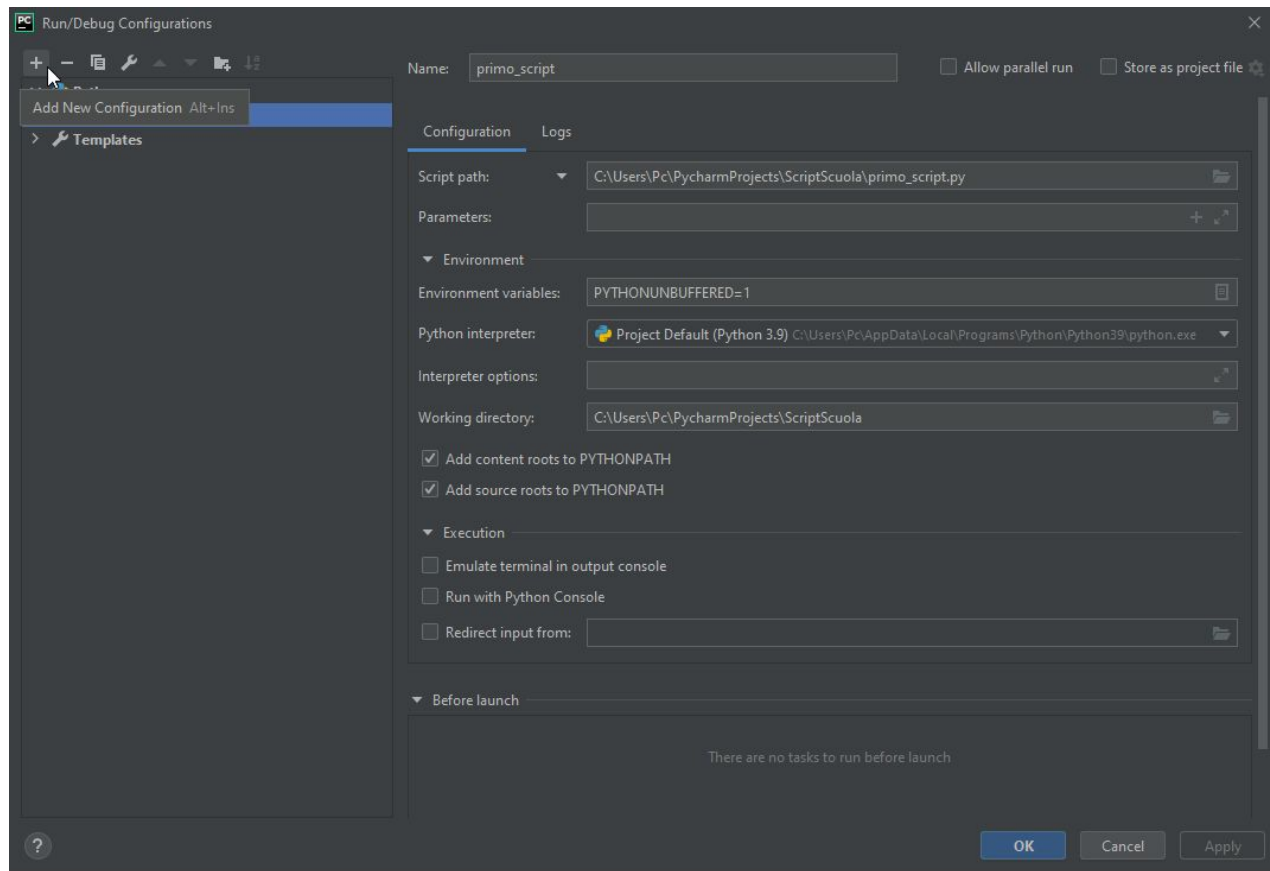
print("il successivo: ", x+1)

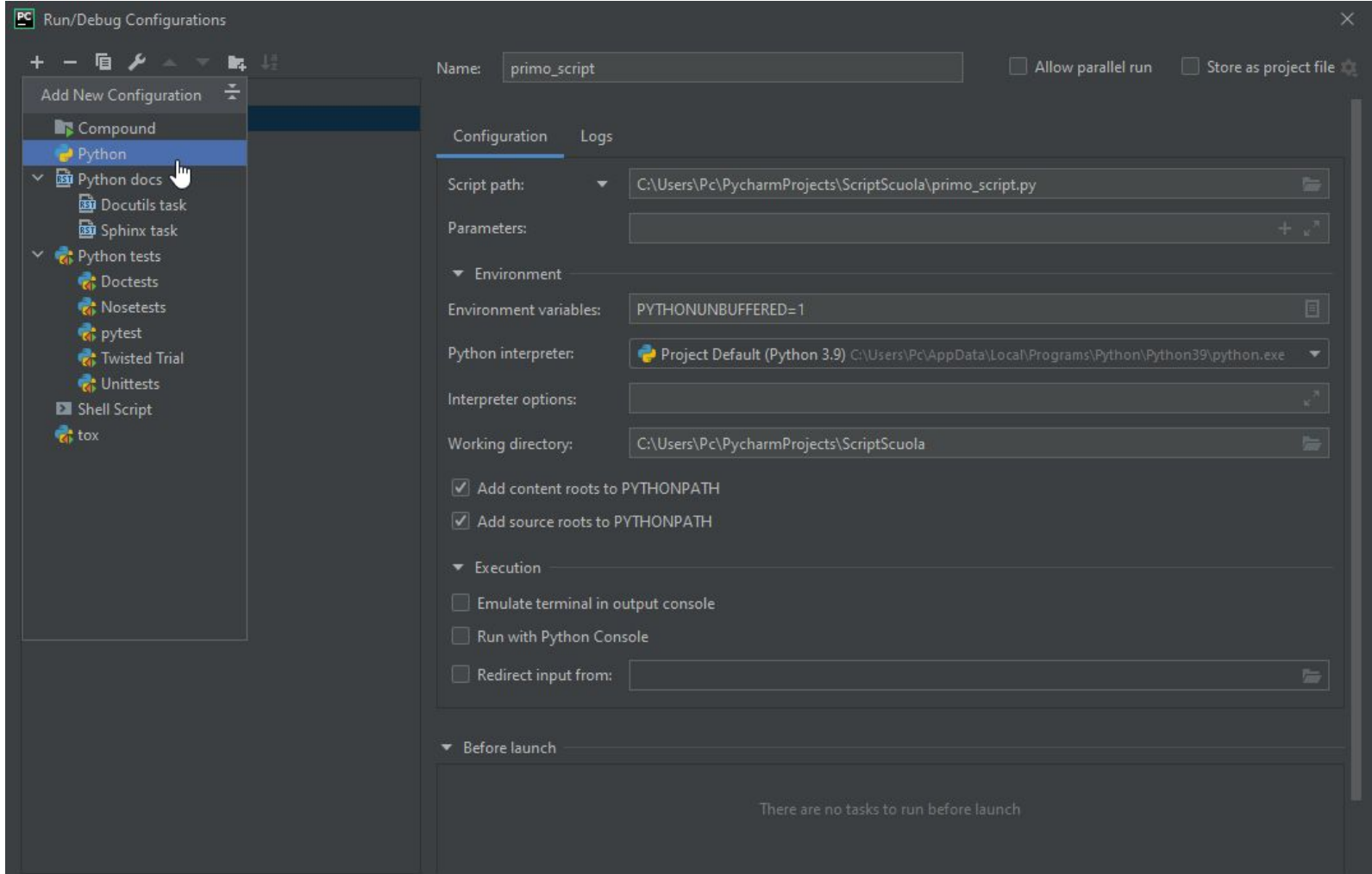
6

# Eseguiamo il nostro secondo script Python



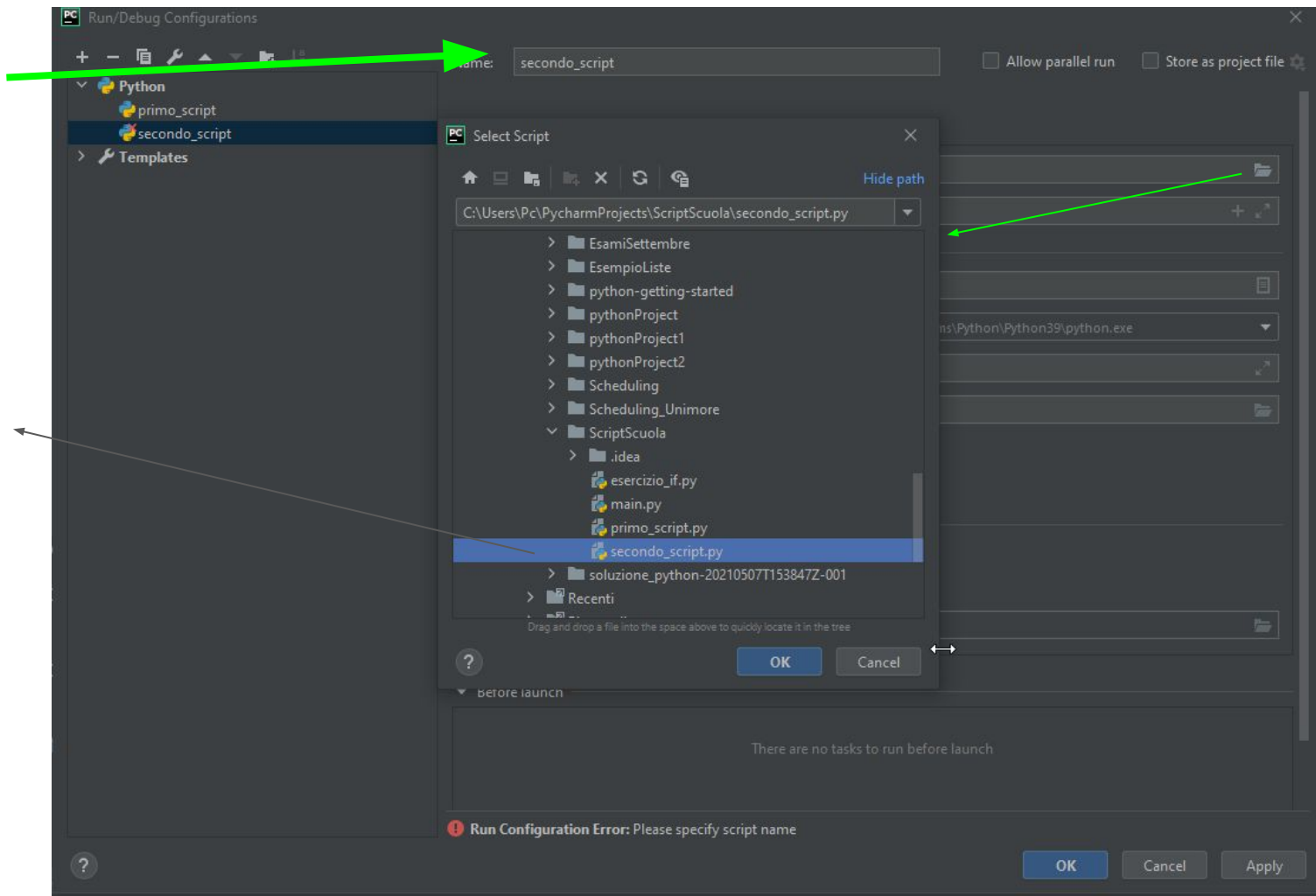
# aggiungiamo una nuova configurazione





le diamo un  
nome

scegliamo il  
file che  
vogliamo  
venga  
interpretato



scriviamo un semplice programma in cui chiediamo all'utente di inserire due numeri, il programma mostra poi la somma dei due numeri

```
inserisci un numero intero  
12  
inserisci un altro numero intero  
6  
18
```

```
primo_script.py x
1
2
3 ▶ if __name__ == '__main__':
4     x = int(input("inserisci un numero intero\n"))
5     y = int(input("inserisci un altro numero intero\n"))
6     risultato = x+y
7     print(risultato)
8
```



```
1 |
2 |
3 | ▶ if __name__=="__main__":
4 |     x = int(input("inserisci un numero: \n"))
5 |
6 |     if x % 2 == 0:
7 |         print(x, "è pari")
8 |     else:
9 |         print(x, "è dispari")
```

# selezione semplice IF

non sono  
obbligatorie  
le parentesi  
tonde

sono  
obbligatorie i  
due punti :

è obbligatoria  
l'indentazione

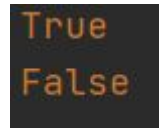
```
1 |
2 |
3 | ► if __name__=="__main__":
4 |     x = int(input("inserisci un numero: \n"))
5 |
6 |     if x % 2 == 0:
7 |         print(x, "è pari")
8 |     else:
9 |         print(x, "è dispari")
```

```
inserisci un numero:
14
14 è pari
```

# Operatori logici in Python

Gli operatori logici sono gli stessi ma hanno il loro nome inglese, ovvero **and or not**

costanti booleane:

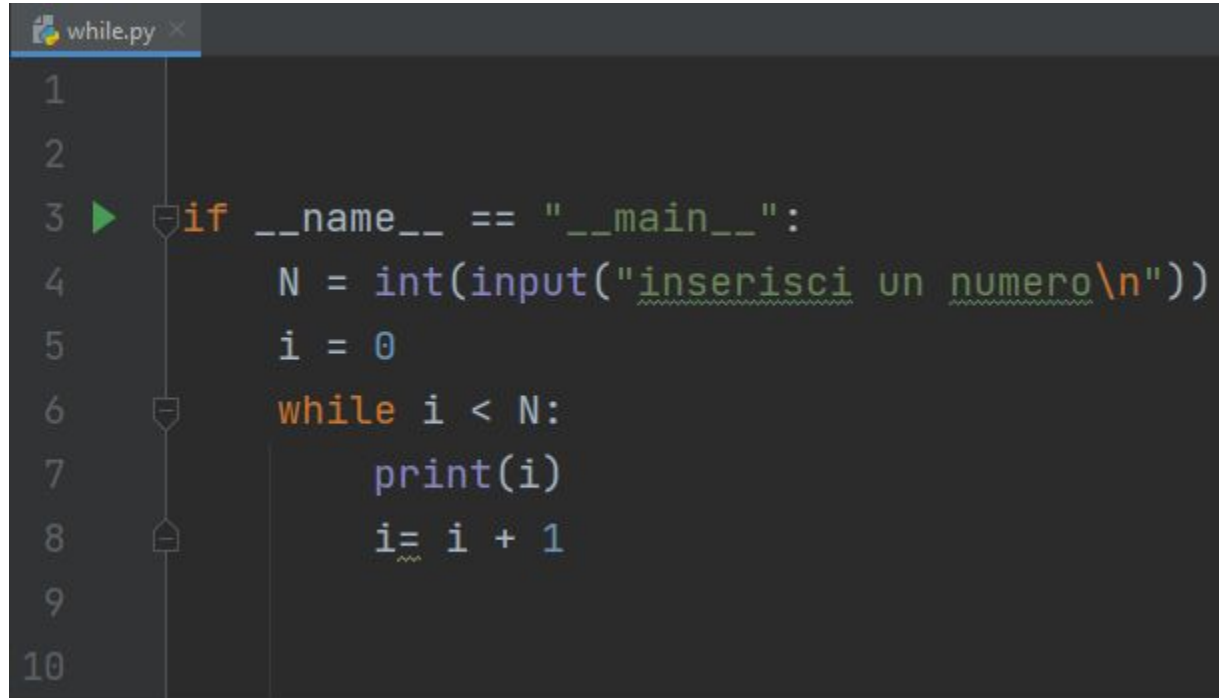


True  
False

Gli operatori di confronto sono gli stessi:

- > maggiore
- < minore
- >= maggiore o uguale
- <= minore o uguale
- != diverso
- == uguale

# Iterazione in Python. while

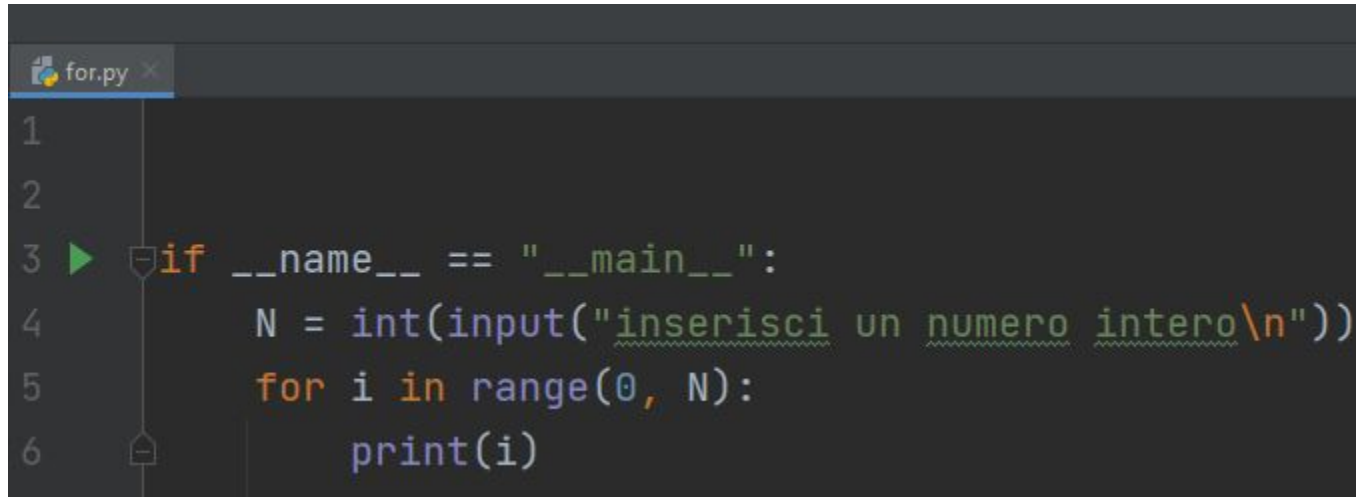


The image shows a screenshot of a Python IDE window titled "while.py". The code is as follows:

```
1  
2  
3 ▶ if __name__ == "__main__":  
4     N = int(input("inserisci un numero\n"))  
5     i = 0  
6     while i < N:  
7         print(i)  
8         i = i + 1  
9  
10
```

The code demonstrates a while loop that prompts the user to enter a number, then prints the numbers from 0 to N-1. The loop variable `i` is incremented by 1 in each iteration. The IDE interface includes a line number column on the left and a vertical cursor.

# Iterazione in python: for



A screenshot of a Python IDE window titled "for.py". The code is as follows:

```
1
2
3 ▶ if __name__ == "__main__":
4     N = int(input("inserisci un numero intero\n"))
5     for i in range(0, N):
6         print(i)
```

The code demonstrates a for loop that iterates from 0 to N-1, printing each value. The input prompt is "inserisci un numero intero\n".

# Le funzioni in Python

- non dobbiamo specificare tipo del valore di ritorno
- non dobbiamo specificare il tipo degli argomenti

```
def somma(x, y):  
    result = x + y  
    return result
```

```
if __name__ == "main":  
    A = int(input("inserisci numero:\n"))  
    B = int(input("inserisci un numero:\n"))  
  
    s = somma(A, B)  
    print("la somma: ", s)
```

main

invocazione della  
funzione

# Le funzioni in Python

```
def printNumbers(N):  
    for i in range(N):  
        print(i)  
  
if __name__ == "__main__":  
    x = int(input("inserisci un numero intero: \n"))  
    printNumbers(x)
```

La funzione nell'esempio non restituisce nulla semplicemente perchè non c'è alcuna istruzione di return al suo interno!

la invoco semplicemente senza considerare il suo valore di ritorno visto che so che non lo ha

# le funzioni in Python

```
def printNumbers(N):  
    for i in range(N):  
        print(i)  
  
if __name__ == "__main__":  
    x = int(input("inserisci un numero intero: \n"))  
    y = printNumbers(x)  
    print(y)
```



# le funzioni in python

```
inserisci un numero intero:
```

```
10
```

```
0
```

```
1
```

```
2
```

```
3
```

```
4
```

```
5
```

```
6
```

```
7
```

```
8
```

```
9
```

```
None
```

# None

In python esiste un valore che indica il **nulla**, ovvero None.

# Esercizi sulle funzioni

- 1) scrivi una funzione chiamata `isEven` che preso in ingresso un numero mi **restituisce** vero se è pari falso se è dispari
- 2) scrivi una funzione chiamata `printDouble` che preso in ingresso un numero intero **stampa** il suo doppio

# Le stringhe in Python

```
a = "Ciao" = a = 'Ciao'
```

```
>>> len(a)  
4
```

la funzione **len** mi dice  
la lunghezza di una  
stringa

```
>>> a[1]  
'i'
```

le stringhe sono array di caratteri, accedo  
ad un singolo carattere con le parentesi  
quadre.

[ posizione ]

Le posizioni partono da zero e finiscono a  
**len(a) - 1**

# Le stringhe in Python

```
a = "Ciao"
```

da 0 a **len(a)** -1

```
>>> a[0]
'c'
>>> a[1]
'i'
>>> a[2]
'a'
>>> a[3]
'o'
```

```
>>> a[-1]
'o'
>>> a[-2]
'a'
>>> a[-3]
'i'
>>> a[-4]
'c'
>>> a[-5]
```

da -1 a **-len(a)**  
per avere i  
caratteri al  
contrario

```
>>> a[4]
Traceback (most recent call last):
  File "<input>", line 1, in <module>
IndexError: string index out of range
```

# L'operatore IN

L'operatore **in** in Python serve per controllare la presenza di un elemento all'interno di una sequenza.

Lo possiamo usare per controllare la presenza di una **stringa** all'interno della nostra stringa

```
>>> 'c' in a  
True
```

# L'operatore IN

```
>>> a = "ciao mamma guarda come mi diverto"  
>>> "mamma" in a  
True
```

# la funzione index

la funzione index, chiamata su una stringa mi dice l'indice in cui si trova un carattere o una sottostringa al suo interno

```
>>> a = "ciao mamma guarda come mi diverto"
>>> a.index("ciao")
0
>>> a.index("mamma")
5
>>> a.index("a")
2
>>> a.index("x")
Traceback (most recent call last):
  File "<input>", line 1, in <module>
ValueError: substring not found
```



# Esercizio con le stringhe

scriviamo una funzione che prese in ingresso due stringhe A e B mi restituisce l'indice dove comincia B all'interno di A nel caso in cui B sia presente in A, -1 altrimenti.

Usa la funzione nel main. Il main chiede due stringhe in input e stampa l'indice dove la seconda inizia nella prima

Esempio:

A = "ciao" B = "a" la funzione deve restituire 2

A = "Pippo" B = "pluto" la funzione deve restituire -1

A = "ciao mamma guarda come mi diverto" B = "come" la funzione deve restituire 18

# Operatore + con le stringhe

L'operatore + per la somma è ridefinito per le stringhe e svolge l'operazione di concatenazione di stringhe!

```
>>> a = "ciao"  
>>> b = "mamma"  
>>> a+b  
'ciaomamma'
```

# Operatore \* con le stringhe

L'operatore \* è ridefinito per le stringhe e svolge l'operazione di **ripetizione**

```
>>> a*10  
'ciaociaociaociaociaociaociaociaociao'
```

# La stringa vuota

In Python possiamo creare una stringa di lunghezza zero, ovvero che non contiene alcun carattere.

```
string = ""
```

# str

La funzione str in python converte quello che le diamo a stringa.

Se le diamo un numero lo converte a testo!

```
x = str(3.5)
```

```
print(x)
```

x sarà la stringa "3.5"



# Iterare una stringa

Possiamo passare in rassegna tutti i caratteri che costituiscono una stringa in diversi modi:

1) `s = "ciao"`  
`for i in range(len(s)):`  
`print(s[i])`

i sarà una variabile intera che va da 0 a len(s)-1 e dovrà essere usato come indice per accedere ai diversi caratteri di s

2) `s = "ciao"`  
`for x in s:`  
`print(x)`

x sarà una variabile di tipo stringa che avrà ad ogni iterazione un valore diverso e sarà ciascuno dei caratteri di s in ordine:

1° iterazione x = "c"  
2° iterazione x = "i"  
3° iterazione x = "a"  
4° iterazione x = "o"

# Metodi sulle stringhe

Su una stringa possiamo invocare tantissime funzioni. La cui descrizione trovate [qui](#), dove potete anche provare ciascuno dei metodi presentati e vedere cosa fa.

# Intervalli di indici

Possiamo accedere ad una stringa usando anche gli intervalli di indici. La sintassi di accesso con gli intervalli di indici è la seguente:

```
s1 = s[start:stop:step]
```

s1 sarà una nuova stringa con i caratteri di s che vanno da **start** a **stop - 1** presi con un intervallo di **step**.



## Intervalli di indici

start = 2

stop = 8

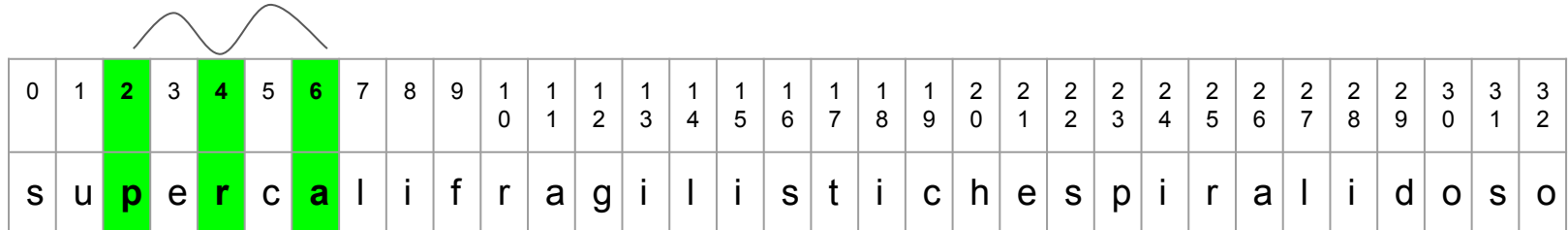
step = 2

s = "supercalifragilistichespiralidoso"

s1 = s[start:stop:step]

il programma stampa "pra"

print(s1)



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
s	u	p	e	r	c	a	l	i	f	r	a	g	i	l	i	s	t	i	c	h	e	s	p	i	r	a	l	i	d	o	s	o

# Le liste

In python per rappresentare una sequenza di elementi si utilizzano **le liste**. Gli elementi all'interno di una lista **non devono necessariamente essere dello stesso tipo** e possono quindi essere tutto quello che vogliamo.

Le liste sono sequenze di elementi racchiusi dalle parentesi quadre [ e separati tra loro dalla virgola ,

```
a = ["ciao", 245, 88.7, "sole"]
```

```
>>> len(a)  
4
```

```
mylist = []
```

```
>>> len(mylist)  
0
```

# Le liste

Le liste sono uno dei tipi di dato **fondamentali** in Python

```
>>> type(a)  
<class 'list'>
```

# Le liste

Gli elementi di una lista in python sono **accessibili mediante indici**, gli indici si mettono tra parentesi quadre `[]` e vanno da 0 a `len(a) - 1` dove `a` è la mia lista di elementi. Come per le stringhe posso usare `-1` , `-2` .... `-len(a)`

```
>>> a[0]  
'ciao'
```

```
>>> a[1]  
245
```


```
>>> a[2]  
88.7
```

```
>>> a[3]  
'sole'
```

# Le liste

una lista è **modificabile** nel senso che possiamo:

- modificare il valore di un suo elemento
- aggiungere o rimuovere un elemento



```
>>> a[2] = "nuovo valore"  
>>> print(a)  
['ciao', 245, 'nuovo valore', 'sole']
```

# Operatore in

L'operatore in funziona anche con le liste:

```
a = ["ciao", 245, 88.7, "sole"]
```

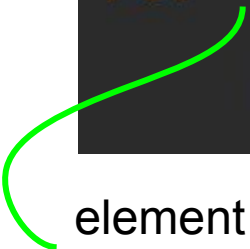
```
>>> 245 in a  
True
```

```
>>> 246 in a  
False
```

# Iterare una lista

Per “iterare” un lista, ovvero scorrere tutti i suoi elementi possiamo servirci del ciclo **for**:

```
mylist = [1, "ciao", 3.14, "pippo", 5, 0.18]  
  
for element in mylist:  
    print(element)
```



element sarà passo dopo passo  
ciascuno degli elementi nella lista

# Intervalli di indici

possiamo specificare un intervallo di indici dicendo a che indice comincia e a che indice finisce l'intervallo.

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

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

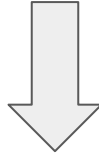
```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]  
print(thislist[5:])
```



# Intervalli di indici

possiamo anche modificare il valore di tutti gli elementi compresi in un intervallo di indici:

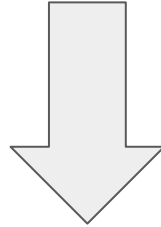
```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "mango"]  
thislist[1:3] = ["blackcurrant", "watermelon"]
```



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

```
['apple', 'blackcurrent', 'watermelon', 'orange', 'kiwi', 'mango']
```

```
thislist[1:3] = ["raspberry"]
```

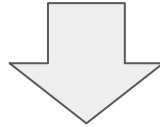


```
['apple', 'raspberry', 'orange', 'kiwi', 'mango']
```

# Inserire un elemento in una lista

Per inserire un elemento dentro ad una lista possiamo usare la funzione **insert**. La funzione **insert** se chiamata su una lista mi consente infatti di inserire un elemento ad una determinata posizione

```
thislist = ['apple', 'raspberry', 'orange', 'kiwi']  
thislist.insert(1, 'mango')
```

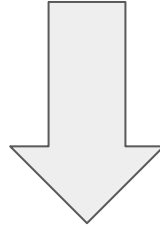


```
['apple', 'mango', 'raspberry', 'orange', 'kiwi']
```

# Inserire un elemento in fondo alla lista

Per inserire un elemento in fondo ad una lista usiamo la funzione **append**

```
thislist = ['apple', 'raspberry', 'orange', 'kiwi']  
thislist.append("mango")
```



```
['apple', 'raspberry', 'orange', 'kiwi', 'mango']
```

## rimuovere un elemento da una lista

Per rimuovere un elemento da una lista possiamo usare la funzione **remove**. La funzione infatti se chiamata su una lista rimuove l'elemento passato come argomento

```
thislist = ['apple', 'raspberry', 'orange', 'kiwi']
```

```
thislist.remove("apple")
```

```
['raspberry', 'orange', 'kiwi']
```

rimuovere un elemento dalla lista

```
>>> thislist = ['apple', 'raspberry', 'orange', 'kiwi']  
>>> thislist.remove("banana")
```

```
Traceback (most recent call last):
```

```
  File "<input>", line 1, in <module>
```

```
ValueError: list.remove(x): x not in list
```

# Moltiplicazione e addizione

Gli operatori aritmetici sono ridefiniti anche per le liste:

```
>>> thislist * 2  
['apple', 'raspberry', 'kiwi', 'apple', 'raspberry', 'kiwi']
```

```
>>> thatlist = ['banana', 'orange', 'watermelon']  
>>> thislist = ['apple', 'raspberry', 'kiwi']  
>>> thatlist + thislist  
['banana', 'orange', 'watermelon', 'apple', 'raspberry', 'kiwi']
```

# rimuovere un elemento da una lista

Per rimuovere un elemento da una lista possiamo usare anche la funzione **pop**. Se chiamata su una lista infatti mi rimuove **e mi restituisce** l'elemento all'indice datole come parametro. Se non le specifico nessun indice rimuove l'ultimo elemento.

```
thislist = ['apple', 'raspberry', 'orange', 'kiwi']  
elemento_rimosso = thislist.pop(2)
```

```
>>> print(thislist)  
['apple', 'raspberry', 'kiwi']  
>>> print(elemento_rimosso)  
orange
```



# Esercizio sulle liste

scrivi uno script Python in cui chiedi all'utente un numero intero N dopodichè chiedi all'utente N numeri che costituiranno una lista. Alla fine il programma stampa la lista costituita dagli N numeri inseriti dall'utente.

```
if __name__ == "__main__":  
    N = int(input("inserisci N: "))
```

# Esercizio sulle liste

- scrivi una funzione chiamata **raddoppia** la funzione prende in ingresso una lista e restituisce una nuova lista che contiene gli stessi elementi della lista presa in ingresso ma ciascuno è raddoppiato.
- scrivi una funzione chiamata **solo\_pari** la funzione prende in ingresso una lista e restituisce una nuova lista che contiene solamente gli elementi pari della lista presa in ingresso

scrivi uno script Python in cui chiedi all'utente un numero intero N dopodichè chiedi all'utente N numeri che costituiranno una lista. Alla fine il programma utilizza le due funzioni prima create per:

- ottenere una nuova lista con gli elementi raddoppiati
- ottenere una nuova lista con gli elementi solamente pari

# List Comprehension

La **list comprehension** è una sintassi più comoda e veloce per creare nuove liste a partire dal contenuto di liste già esistenti.

# List Comprehension

Supponiamo per esempio di voler creare una nuova lista contenente gli stessi elementi (raddoppiati) della lista **mylist**

```
mylist_doubled = [x*2 for x in mylist]
```

# List Comprehension

Supponiamo per esempio di voler creare una nuova lista contenente solo i numeri pari della lista **mylist**

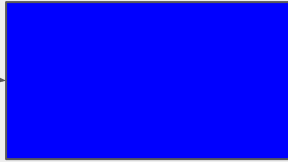
```
mylist_even_only = [x for x in mylist if x % 2 == 0]
```

# Copia

a



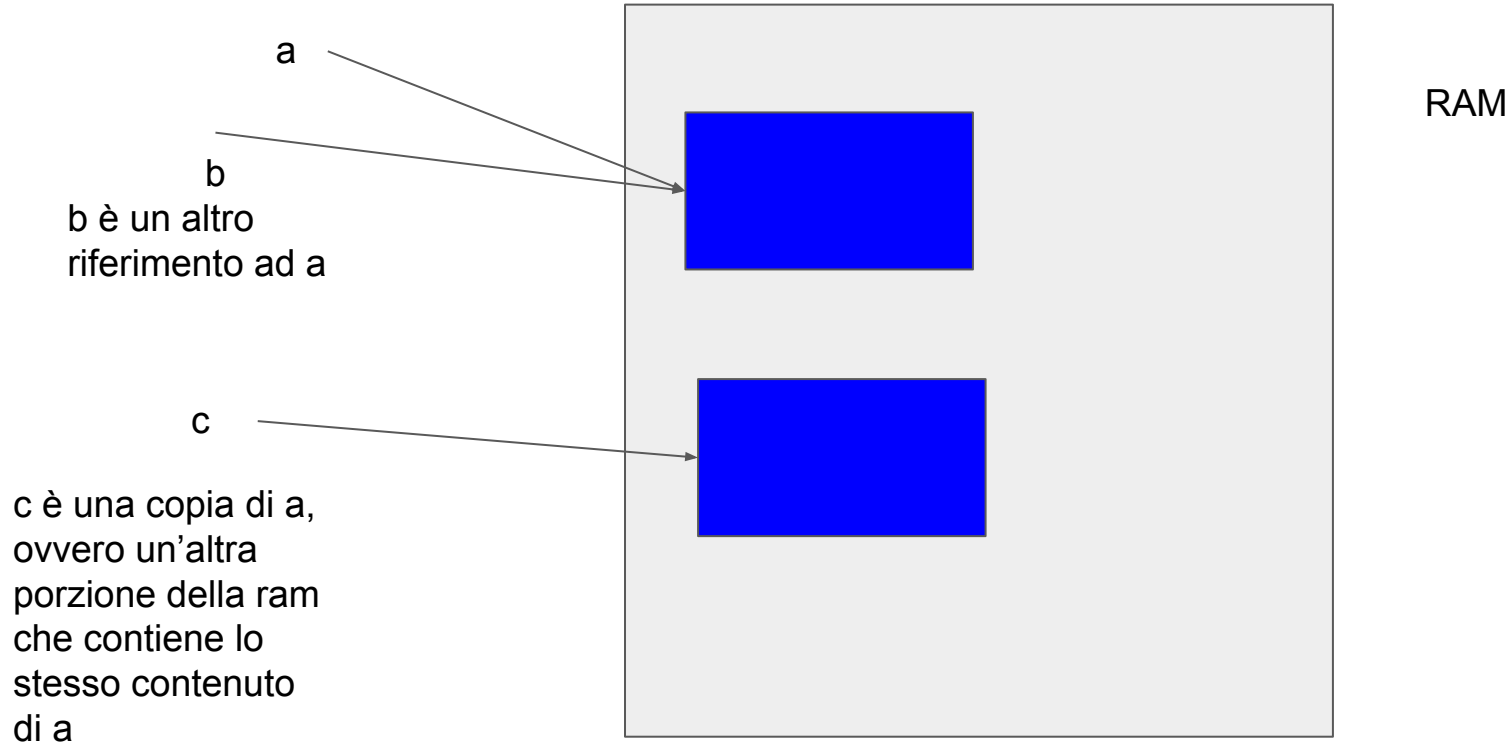
c



RAM

c è una copia di a,  
ovvero un'altra  
porzione della ram  
che contiene lo  
stesso contenuto  
di a

# Copia



# Copia

In Python le variabili di tipo **semplice** (**int**, **float**, **str**) sono assegnate **per copia**

```
def main():  
    a = 10  
    b = a # b è una COPIA di a  
    a = a + 1  
    print(a) → stampa 11  
    print(b) → stampa 10
```

```
if __name__ == '__main__':  
    main()
```



# Copia

In Python le variabili di tipo **semplice** (**int**, **float**, **str**) sono passate alle funzioni **per copia**

```
def funzione(a):  
    a = a + 1
```

```
def main():  
    var = 10  
    funzione(var)  
    print(var)
```

quando viene chiamata la  
funzione *a* diventa una  
**copia** di *var*  
se modifico *a* *var* rimane  
invariata!

→ stampa 10

# Riferimento

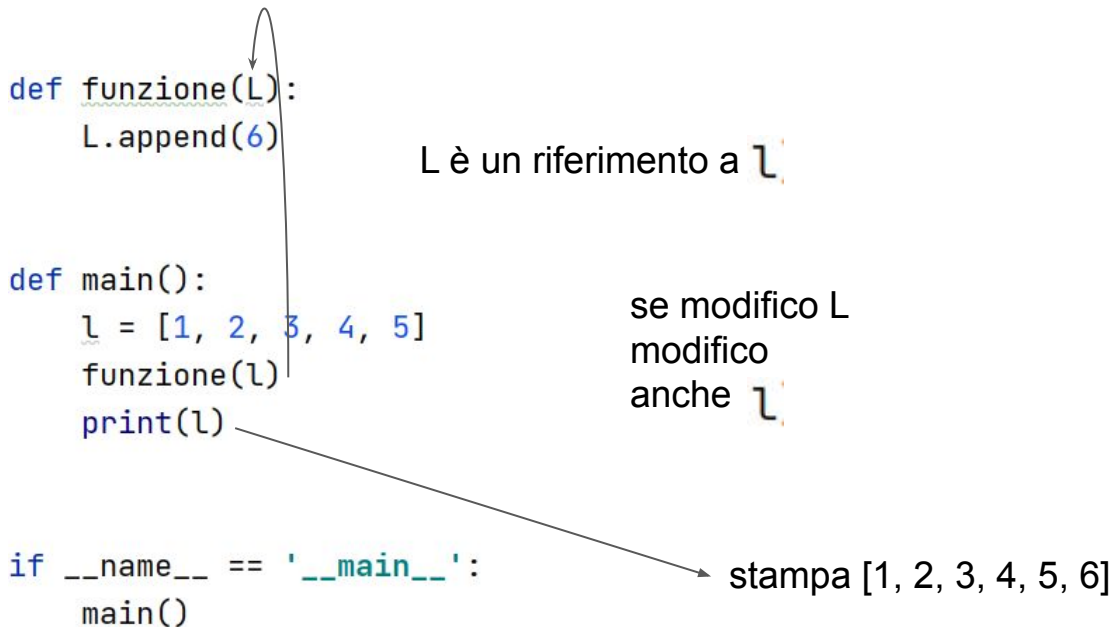
In Python le variabili di tipo strutturato sono invece assegnate per riferimento.

```
def main():  
    l = [1, 2, 3, 4, 5]  
    l2 = l # l2 non è una copia di l ma è un altro riferimento ad l  
    l.append(6)  
    print(l) → stampa [1,2,3,4,5,6]  
    print(l2) → stampa [1,2,3,4,5,6]
```

```
if __name__ == '__main__':  
    main()
```

# Riferimento

In Python le variabili di tipo strutturato sono invece passate alle funzioni per riferimento.



# Creare una copia di una lista

la funzione `copy` restituisce **una copia** della lista su cui è chiamata.

```
l = [1, 2, 3, 4, 5]
```

```
l2 = l.copy() # ora l2 è una copia di l
```

# Le Tuple

In Python le Tuple sono un tipo di dato fondamentale, utilizzato per memorizzare una collezione di valori in un'unica variabile. Una tupla è un tipo di dato **non modificabile**. Una volta creata una tupla non possiamo modificare i suoi elementi e nemmeno aggiungerne o toglierne.

# Le Tuple

Per dichiarare una tupla si racchiudono gli elementi che la costituiscono tra parentesi tonde ( , , , .... , ) separati dalla virgola.

```
a = ("a", 1, "b")
```

# Le Tuple

```
a = ("a", 1, "b")  
print(a)  
a[0] = 1
```

('a', 1, 'b')

Traceback (most recent call last):

File "<string>", line 4, in <module>

TypeError: 'tuple' object does not support item assignment

# Accesso agli elementi

Gli elementi possono essere acceduti utilizzando le parentesi quadre ed indicando il loro indice, come per le liste.

```
a = ("a", 1, "b")  
print(a[2])
```





# Le Tuple

Per conoscere quanti elementi ha una tupla possiamo usare la funzione **len()**

```
a = ("a", 1, "b")  
print(len(a))
```



3

# Le Tuple

Le tuple supportano gli intervalli di indici come le liste

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

# Unpacking

Quando creiamo una tupla, le assegniamo dei valori, questo prende il nome di *impacchettamento*. Stiamo mettendo tutti i valori insieme

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

# Unpacking

In Python possiamo anche *spacchettare* le nostre tuple! assegnando tutti i valori a diverse variabili!

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

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

```
print(green)
```

```
print(yellow)
```

```
print(red)
```

\*

se usiamo l'asterisco facciamo in modo che tutti i restanti valori finiscano dentro ad una lista.

```
fruits = ("apple", "banana", "cherry", "strawberry", "raspberry")
```

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

```
print(green)
```

```
print(yellow)
```

```
print(red)
```



["cherry", "strawberry", "raspberry"]

e così?

```
fruits = ("apple", "mango", "papaya", "pineapple", "cherry")
```

```
(green, *tropic, red) = fruits
```

```
print(green)
```

```
print(tropic)
```

```
print(red)
```

e così?

```
fruits = ("apple", "mango", "papaya", "pineapple", "cherry")
```

```
(green, *tropic, red) = fruits
```

```
print(green)
```

```
print(tropic)
```

```
print(red)
```



["mango", "papaya", "pineapple"]

# Iterare le tuple

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



# enumerate

Un modo per iterare le liste era utilizzando la funzione **enumerate**. Tale funzione se le diamo una lista restituisce **(per ogni elemento della lista)** una **coppia** (una tupla con due elementi) in cui il primo è l'indice ed il secondo è il valore corrispondente.

```
L = ["ciao", "pippo", "pluto", "mamma"]  
for x in enumerate(L):  
    print(x)
```

```
(0, 'ciao')  
(1, 'pippo')  
(2, 'pluto')  
(3, 'mamma')
```

# Unpacking di enumerate

scriviamo un programma in cui creiamo una lista di numeri interi generati casualmente tra 800 e 1000 e poi li stampiamo in coppia con i loro indici. Cercando di ***spacchettare*** la tupla che ci viene data da **enumerate**.

# I Dizionari

I dizionari sono un tipo di dato fondamentale in Python, utilizzato per rappresentare delle collezioni di coppie **chiave:valore**.

Non tutti i tipi di dato possono essere chiavi, come valore invece possiamo avere **qualsunque tipo di dato**.

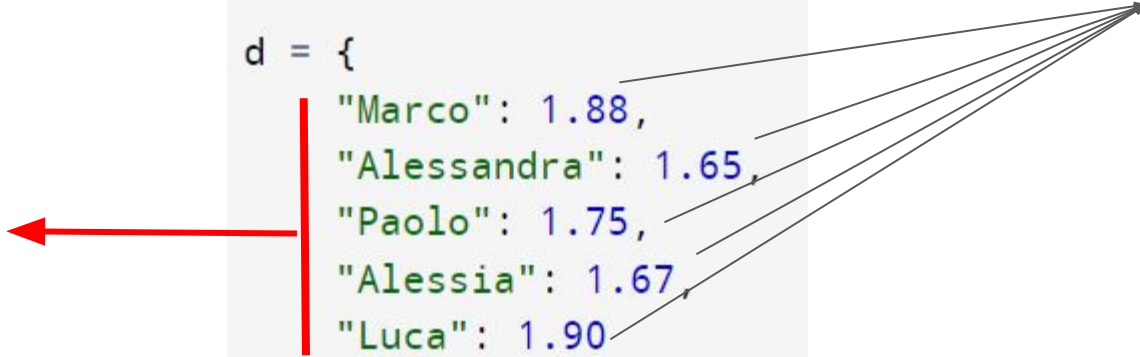
# I dizionari

chiavi

```
d = {  
    "Marco": 1.88,  
    "Alessandra": 1.65,  
    "Paolo": 1.75,  
    "Alessia": 1.67,  
    "Luca": 1.90  
}
```

```
print(d)
```

i valori associati  
ad ogni chiave



# Accesso agli elementi

Per accedere agli elementi di un dizionario si usano sempre le parentesi quadre.  
Ma invece degli indici cosa utilizzeremo?

# Accesso agli elementi

Per accedere agli elementi di un dizionario si usano sempre le parentesi quadre. Ma invece degli indici cosa utilizzeremo?

## LE CHIAVI



# Le chiavi

Ogni chiave ci serve per accedere al suo valore corrispondente!

```
d = {  
    "Marco": 1.88,  
    "Alessandra": 1.65,  
    "Paolo": 1.75,  
    "Alessia": 1.67,  
    "Luca": 1.90  
}  
  
print(d)
```

```
print(d["Paolo"])
```


Cosa succede se cerco di accedere al dizionario utilizzando una chiave che non esiste?

# KeyError

Se provo ad accedere al dizionario utilizzando una chiave inesistente Python mi da errore! **KeyError**

```
d = {  
    "Marco": 1.88,  
    "Alessandra": 1.65,  
    "Paolo": 1.75,  
    "Alessia": 1.67,  
    "Luca": 1.90  
}  
  
print(d)
```

```
print(d["pippo"])
```



```
Traceback (most recent call last):  
  File "<string>", line 11, in <module>  
KeyError: 'pippo'
```



# Chiavi duplicate

In un dizionario **non sono ammesse chiavi duplicate!**

Siccome la chiave è utilizzata per accedere ad un singolo valore in un dizionario non sono ammesse ripetizioni della stessa chiave.

# I Dizionari


I dizionari sono modificabili:

- possiamo modificare il valore di un elemento (accedendovi con la chiave)
- possiamo aggiungere un **elemento** al dizionario (una coppia chiave:valore)
- possiamo rimuovere elementi dal dizionario.

# Modificare il valore di un elemento

Per modificare il valore di un elemento ci accediamo utilizzando la sua chiave, e gli assegniamo un nuovo valore.

```
d = {  
    "Marco": 1.88,  
    "Alessandra": 1.65,  
    "Paolo": 1.75,  
    "Alessia": 1.67,  
    "Luca": 1.90  
}  
  
print(d)
```



```
d["Paolo"] = 2.00
```

in questo modo cambio il valore  
associato alla chiave "Paolo"

# Aggiungere un elemento ad un dizionario

Per aggiungere un elemento ad un dizionario basta specificare la nuova coppia chiave:valore che si desidera aggiungere al dizionario stesso.

```
d = {  
    "Marco": 1.88,  
    "Alessandra": 1.65,  
    "Paolo": 1.75,  
    "Alessia": 1.67,  
    "Luca": 1.90  
}  
  
print(d)
```



```
d["Gianluca"] = 1.79
```

In questo modo creo una nuova coppia chiave:valore in cui la chiave è la stringa "Gianluca" ed il valore è il numero 1.79


# Aggiungere più elementi al dizionario

Per aggiungere più elementi possiamo usare la funzione **update**, che, chiamata su un dizionario lo aggiorna aggiungendoci gli elementi del dizionario preso in ingresso.

```
d = {  
    "Marco": 1.88,  
    "Alessandra": 1.65,  
    "Paolo": 1.75,  
    "Alessia": 1.67,  
    "Luca": 1.90  
}
```

```
print(d)
```

```
d.update({"Gianluca":1.79, "Marcello": 1.81})
```

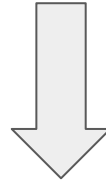


in questo modo aggiungo due  
coppie chiave valore, Gianluca:  
1,79 e Marcello 1.81

# Aggiornare un dizionario

```
d = {  
    "Marco": 1.88,  
    "Alessandra": 1.65,  
    "Paolo": 1.75,  
    "Alessia": 1.67,  
    "Luca": 1.90  
}  
  
print(d)
```

```
d.update({"Gianluca": 1.79, "Marcello": 1.81, "Marco": 1.99})
```

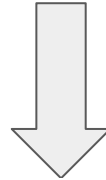


?

# Aggiornare un dizionario

```
d = {  
    "Marco": 1.88,  
    "Alessandra": 1.65,  
    "Paolo": 1.75,  
    "Alessia": 1.67,  
    "Luca": 1.90  
}  
  
print(d)
```

```
d.update({"Gianluca": 1.79, "Marcello": 1.81, "Marco": 1.99})
```

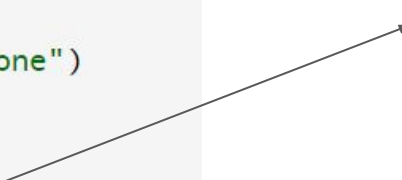


```
{  
    "Marco": 1.99,  
    "Alessandra": 1.65,  
    "Paolo": 1.75,  
    "Alessia": 1.67,  
    "Luca": 1.9,  
    "Gianluca": 1.79,  
    "Marcello": 1.81  
}
```

# Eliminare elementi da un dizionario

Per eliminare gli elementi da un dizionario possiamo usare il metodo **pop**. Questo metodo, se chiamato su un dizionario, elimina la coppia chiave:valore con la chiave che gli viene passata come argomento.

```
d = {  
    "Alessio": 7.5,  
    "Marcello": 4.25,  
    "Giorgia": 10,  
    "Simone": 3.5  
}  
  
d.pop("Simone")  
  
print(d)
```



```
{'Alessio': 7.5, 'Marcello': 4.25, 'Giorgia': 10}
```



# Iterare le chiavi di un dizionario

Per iterare le chiavi di un dizionario possiamo utilizzare il ciclo **for** in questo modo:

```
d = {  
    "Alessio": 7.5,  
    "Marcello": 4.25,  
    "Giorgia": 10,  
    "Simone": 3.5  
}
```

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

In questo modo **x** sarà  
ciascuna delle chiavi del  
dizionario!

Alessio  
Marcello  
Giorgia  
Simone

# Iterare i valori di un dizionario

Per iterare i valori di un dizionario possiamo utilizzare il ciclo **for** e la funzione **values** in questo modo:

```
d = {  
    "Alessio": 7.5,  
    "Marcello": 4.25,  
    "Giorgia": 10,  
    "Simone": 3.5  
}  
  
for x in d.values():  
    print(x)
```

In questo modo x sarà  
ciascuno dei valori del  
dizionario!

7.5  
4.25  
10  
3.5

# Iterare gli elementi di un dizionario

Per iterare gli elementi di un dizionario possiamo utilizzare il ciclo **for** e la **funzione items**. La funzione **items** restituisce per ogni elemento del dizionario una **coppia** (una **tupla** composta da due elementi):

1. chiave
2. valore

**unpacking** della tupla  
(coppia) in due  
variabili!

```
d = {  
    "Alessio": 7.5,  
    "Marcello": 4.25,  
    "Giorgia": 10,  
    "Simone": 3.5  
}
```

```
for key, value in d.items():  
    print(key, value)
```

```
Alessio 7.5  
Marcello 4.25  
Giorgia 10  
Simone 3.5
```

# Dictionary comprehension

Anche con i dizionari esiste il costrutto di **Dictionary Comprehension**. Possiamo creare un dizionario al volo inserendo (questa volta tra parentesi graffe) una sintassi simile a quella utilizzata in list comprehension. Ricordiamoci però che un elemento di un dizionario è sempre una coppia **chiave-valore**.

```
d1 = {...}
```

```
d2 = {key: value for key, value in d1.items()}
```

# Set

In Python il **set** (*set in inglese significa insieme*) è un tipo di dato fondamentale. E' utilizzato per memorizzare una collezione di oggetti **distinti**.

Gli elementi all'interno di un set:

- non sono posti in alcun ordine (**non sono quindi accessibili mediante indici**)
- non sono quindi modificabili.
- sono presenti singolarmente! Non sono ammesse ripetizioni di uno stesso elemento.

# Set

sequenza di elementi racchiusa da parentesi graffe.



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

```
len(thisset)
```

per conoscere il numero di  
elementi che costituiscono un set  
possiamo utilizzare la funzione  
len()

# Accesso agli elementi

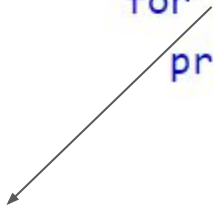
Gli elementi di un set non sono accessibili direttamente (con un indice o una chiave per esempio).

Ma possiamo iterare l'intero **set** utilizzando un ciclo **for** per passare in rassegna tutti i suoi elementi, oppure controllare la presenza di un elemento mediante l'operatore **in**.

# Iterare un set

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

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



x sarà ad ogni  
iterazione  
ciascuno dei  
valori del set



# IN

Possiamo utilizzare l'operatore `in` per controllare la presenza di un elemento all'interno di un set.

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

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

# Modificare un set

Gli elementi all'interno di un set **non sono modificabili**, dal momento che non sono accessibili. Possiamo tuttavia modificare il set stesso, per esempio aggiungendo/rimuovendo elementi!

# Aggiungere un elemento ad un set


Per aggiungere un elemento ad un set utilizziamo la funzione **add**.

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

```
thisset.add("orange")
```

```
print(thisset)
```

la funzione **add** la  
invochiamo su un set.  
Aggiungerà l'elemento  
passatole come  
parametro al set sul  
quale viene invocata.  
La funzione non  
restituisce nulla e  
modifica il set sul quale  
è chiamata.



# Rimuovere un elemento da un set


Per rimuovere un elemento da un set utilizziamo la funzione **remove**.

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

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

```
print(thisset)
```

la funzione **remove** la invochiamo su un set. Rimuoverà l'elemento passato come parametro al set sul quale viene invocata. La funzione non restituisce nulla e modifica il set sul quale è chiamata.



# Unione di due insiemi

Possiamo mediante il metodo **union** fare l'unione di due insiemi (set).

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

la funzione **union** la  
invochiamo su un set.  
**Restituisce** l'unione tra  
il set su cui è chiamata  
(x) ed il set passato  
come parametro (y).

# Intersezione di due insiemi

possiamo mediante il metodo **intersection** fare l'intersezione di due insiemi.

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

la funzione  
**intersection** la  
invochiamo su un set.  
**Restituisce**  
l'intersezione tra il set  
su cui è chiamata (x)  
ed il set passato  
come parametro (y).

# Differenza tra due insiemi

Possiamo mediante il metodo **difference** calcolare la differenza insiemistica tra due **set**.

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


la funzione **difference** la invochiamo su un set. **Restituisce** la differenza insiemistica tra il set su cui è chiamata (x) ed il set passato come parametro (y). Restituisce un set contenente tutti gli elementi di **x** ma non in **y**.

# funzioni **list**, **dict**, **set**

Per ogni tipo di dato in Python, esiste una funzione che **lo costruisce**.

Per esempio la funzione **list** prende come parametro un qualunque oggetto **iterabile** ne **costruisce** una lista e la restituisce.

```
s = "prova"  
L = list(s)  
print(L)
```



```
['p', 'r', 'o', 'v', 'a']
```



# dict

la funzione **dict** costruisce un dizionario partendo da una lista di coppie chiave valore

```
L = [["k1",123],["k2",4]]  
s = dict(L)  
print(s)
```



```
{'k1': 123, 'k2': 4}
```

```
L = [("k1",123),("k2",4)]  
s = dict(L)  
print(s)
```



```
{'k1': 123, 'k2': 4}
```

# dict

oppure da una sequenza di parametri con un nome che sarà la chiave.

```
x = dict(name = "John", age = 36, country = "Norway")  
print(x)
```



```
{'name': 'John', 'age': 36, 'country': 'Norway'}
```

# set

La funzione **set** costruisce un set a partire da un qualunque tipo di dato **iterabile**.

```
L = ["ciao", 1, "RAGAZZI", 3.14]  
s = set(L)  
print(s)
```



```
{'RAGAZZI', 1, 3.14, 'ciao'}
```

# File

Il termine **file** deriva dall'inglese e significa **archivio** o **schedario**. E' un insieme di informazioni memorizzate su un supporto di memorizzazione di massa con un nome ed eventualmente una **estensione**.

Le informazioni in un computer sono sempre memorizzate in binario. Possiamo quindi pensare ad un file come ad una porzione di memoria di massa in cui sono scritti “degli zeri e degli uni” che se letti ed interpretati nella maniera corretta rappresentano il contenuto del mio file, sia esso un'immagine, del testo, una canzone, una tabella excel, un disegno su paint o qualunque altra cosa vi venga in mente.

# File



```
01001010101010  
10101010101010  
10101001010101  
11111010101010  
11100101010101  
01010101010101  
01010101010101  
001010101010...
```

questi 0 e 1 possono significare qualunque cosa, dipende tutto da come li interpretiamo.

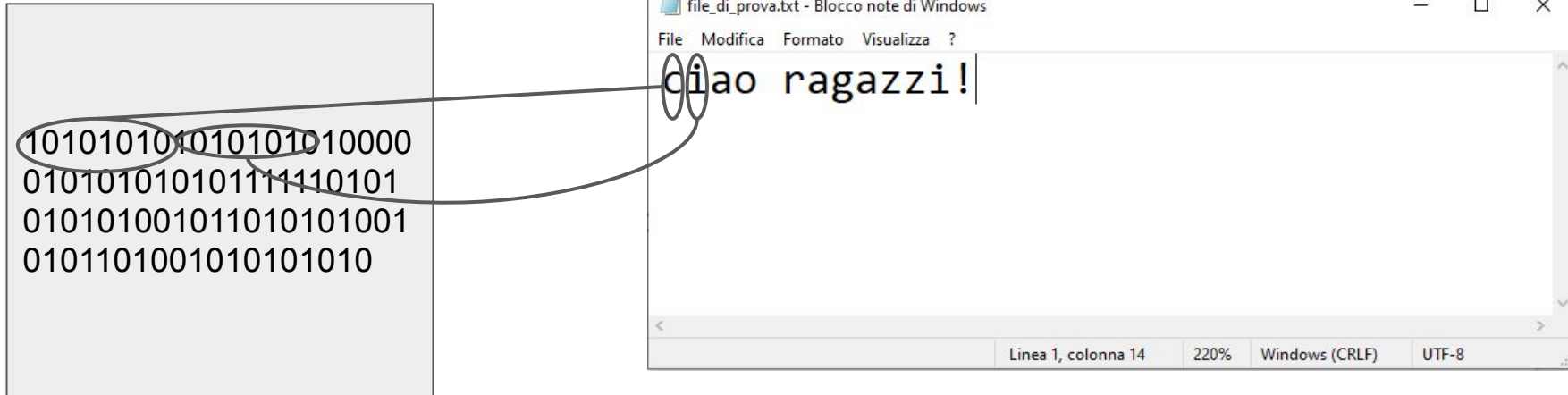
# File e programmi

Il contenuto di un file viene sempre mostrato da un programma. Un file viene sempre aperto da un programma che è in grado di leggere il suo contenuto ed interpretare quegli 1/0 in modo corretto. Il sistema operativo “capisce” con quale programma aprire un file grazie all'estensione del file!

- **.txt** → file di testo, verrà aperto con programmi di videoscrittura (editor di testo) semplici come il blocco note (su windows)
- **.jpg .png .svg** → immagine, verrà aperto con programmi per la visualizzazione o l'editing di immagini
- **.xlsx** → foglio di calcolo, verrà aperto con fogli di calcolo elettronico (Excel, Open Office...)
- **.mp3 .wav** → file audio, verrà aperto con programmi che riproducono audio (Windows Media Player, VLC..)
- **.docx .rtf** → file di testo arricchito/documenti di testo. Sono file di testo non semplice ma con info anche sul colore del testo, sull'organizzazione dei paragrafi, più in generale sullo stile del foglio. Verranno aperti da programmi di videoscrittura più complessi come Word oppure Open Office.

# File di testo

Se per esempio apriamo un file di testo semplice (.txt) su windows, in realtà stiamo aprendo il programma **blocco note** (o un qualunque equivalente editor di testo) dicendogli di **aprire** quel file di testo. Esso interpreterà gli zeri e gli uni scritti su quel file a gruppi di 8 bit (1 byte) come caratteri secondo la codifica [UTF-8](#) e ce li mostrerà a video!



# Immagine

Se per esempio apriamo un'immagine stiamo aprendo in realtà un qualunque programma di visualizzazione o di editing di immagini che altro non farà che **aprire** quel file e visualizzare per noi il suo contenuto, interpretando i suoi “zeri e uni” come informazioni riguardanti il colore dei pixel da dover mostrare a video!

```
10101010101010  
01000001010111  
11111110101011  
01010111010100  
01010101010110  
10101010101010  
01010101011010  
10101010101101  
001010101010...
```

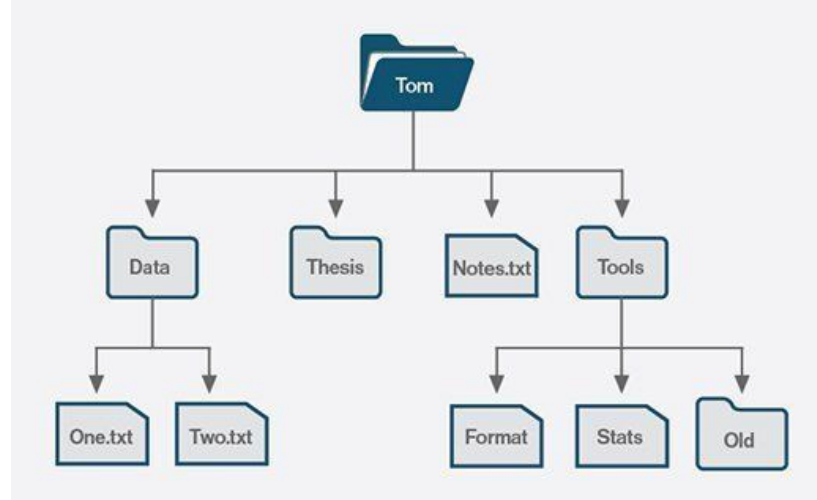


Il programma **Foto** su windows è in grado di interpretare in modo corretto gli uni e gli zeri di un file **.jpg** e di disegnare a video i pixel in modo da riprodurre l'immagine codificata.



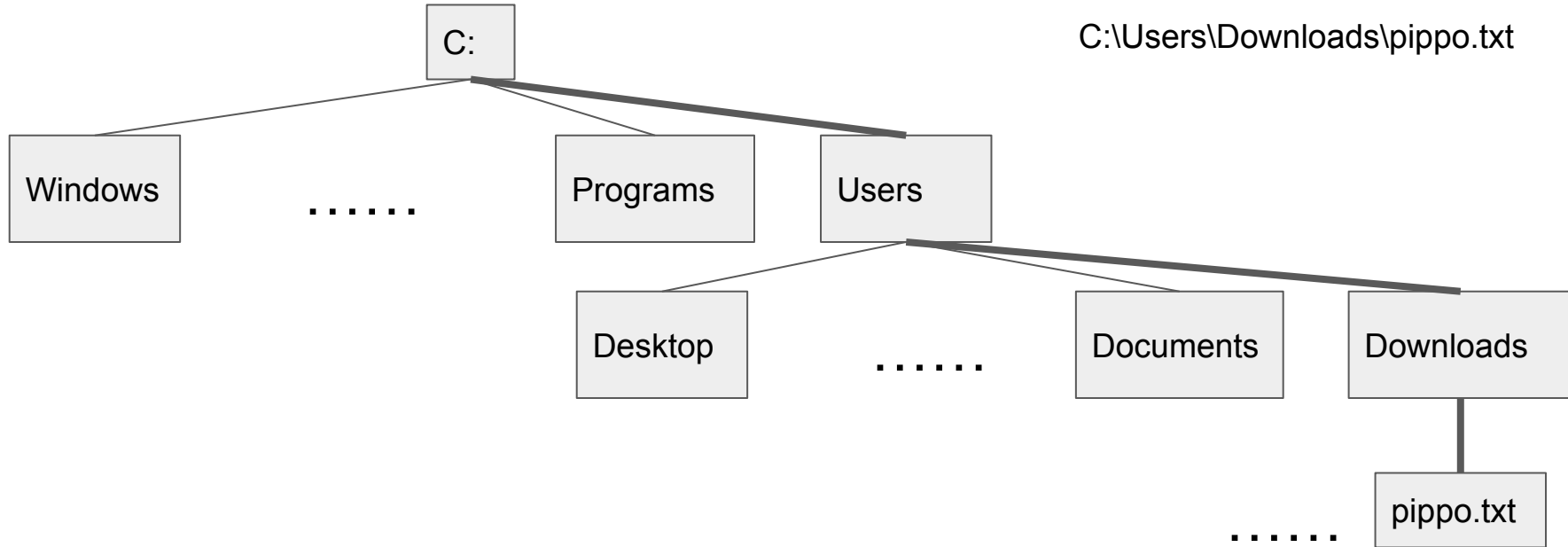
# File System

Le memorie di massa sono organizzate da un componente del sistema operativo chiamato **file system**. Esso si occupa di gestire i dispositivi di memorizzazione organizzando il loro spazio in file e cartelle, secondo una struttura gerarchica e ad albero.



# Percorso di un file

Il percorso di un file è la sequenza di cartelle dentro alle quali è memorizzato il file.



# Windows / Unix

Il carattere per separare le cartelle cambia da sistema operativo a sistema operativo. Su windows il carattere che separa le diverse cartelle in un percorso è \ mentre sui sistemi Unix-Like (MacOs, Ubuntu) è /

# Percorso relativo e Percorso Assoluto di un file.

Il **percorso assoluto** di un file è la sequenza di cartelle (ed il nome del file) che in modo **assoluto** (indipendente dalla cartella in cui io mi trovo) mi consente di riferirmi al file in questione.

Il **percorso relativo** di un file è la sequenza di cartelle (ed il nome del file) che mi consente di riferirmi al file in questione **in relazione** alla cartella in cui mi trovo.

Windows

..\

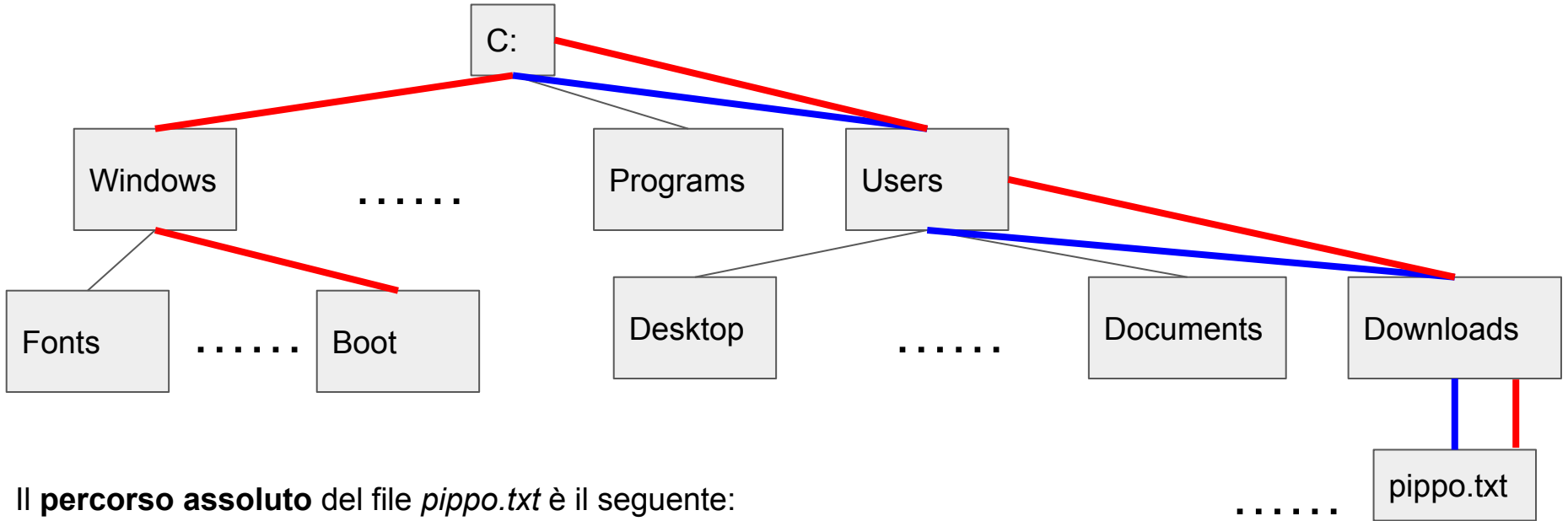
Unix

../



indietro di una cartella

# Percorso relativo e Percorso Assoluto di un file.

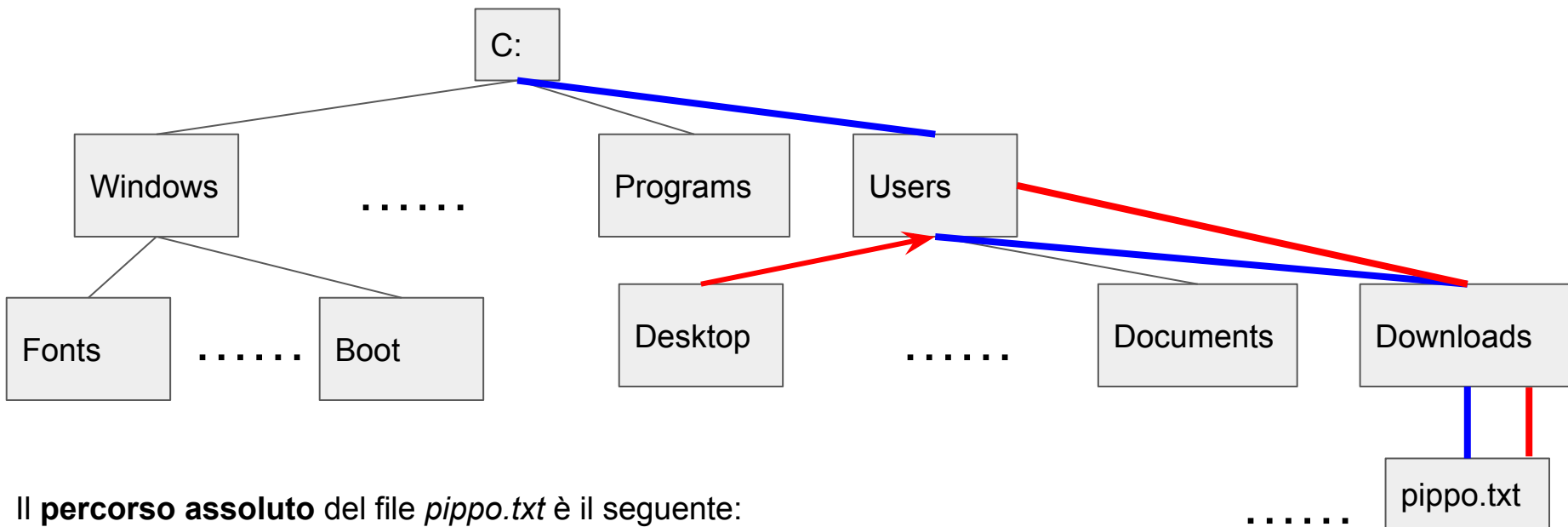


Il **percorso assoluto** del file *pippo.txt* è il seguente:

**C:\Users\Downloads\pippo.txt**

Il **percorso relativo** del file *pippo.txt* se mi trovo nella cartella **Boot** è il seguente: **..\..\Users\Downloads\pippo.txt**

# Percorso relativo e Percorso Assoluto di un file.

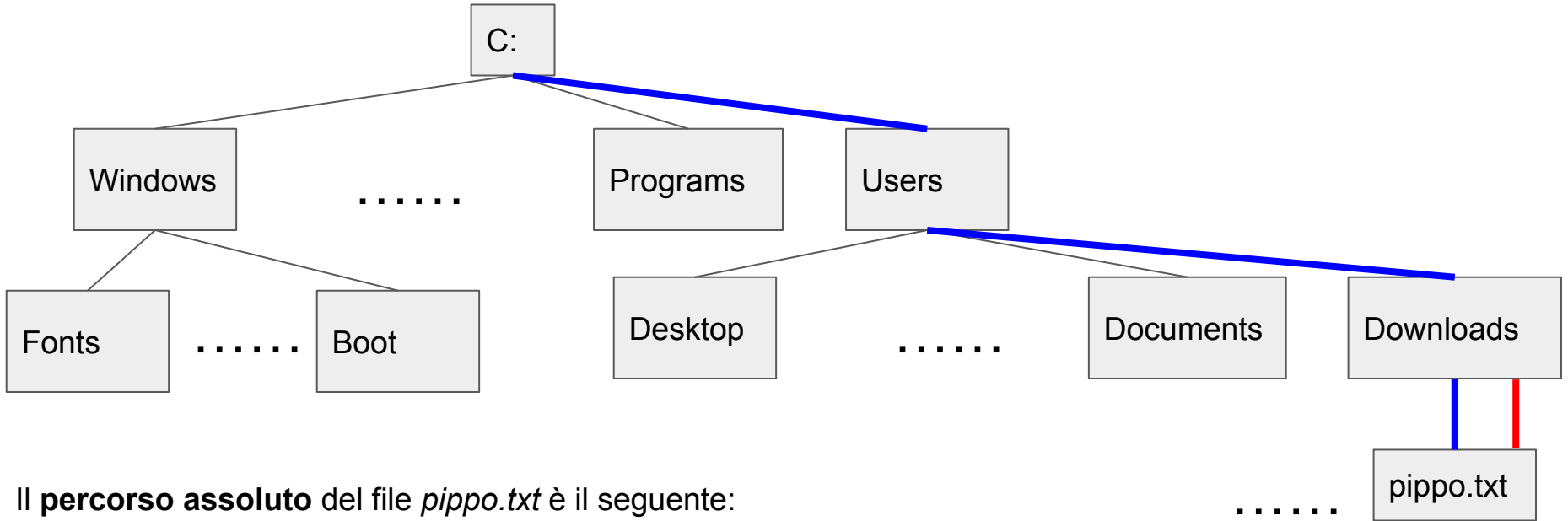


Il **percorso assoluto** del file *pippo.txt* è il seguente:

**C:\Users\Downloads\pippo.txt**

Il **percorso relativo** del file *pippo.txt* se mi trovo nella cartella Desktop è il seguente: **..\Downloads\pippo.txt**

# Percorso relativo e Percorso Assoluto di un file.

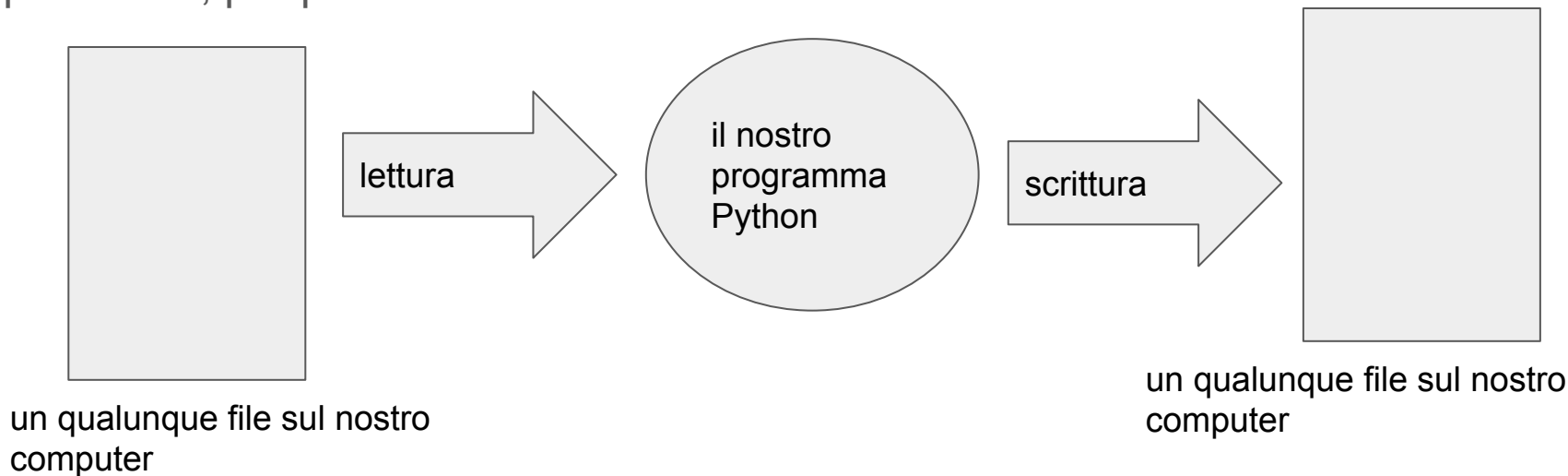


Il **percorso assoluto** del file *pippo.txt* è il seguente:  
**C:\Users\Downloads\pippo.txt**

Il **percorso relativo** del file *pippo.txt* se mi trovo nella cartella Download  
è semplicemente il nome del file ovvero **pippo.txt**

# Lavorare con i File

Vogliamo rendere i nostri programmi in grado di **lavorare** con i file che sono sul nostro computer. Vogliamo che un nostro programma sia in grado di **aprire** un file per poter **leggere** il suo contenuto o per poterci **scrivere** sopra qualcosa oppure perchè no, per poter fare entrambe le cose.





## open

La funzione chiave per lavorare con i file è la funzione **open**. La funzione ha due parametri:

1. **filename**: una stringa che rappresenta il nome (o il percorso) del file che vogliamo aprire
2. **mode**: una stringa che rappresenta la modalità di apertura del file.

La funzione ci restituisce un file, ovvero una variabile “di tipo file” sulla quale possiamo fare operazioni di lettura e/o di scrittura!!

# Modalità

ci sono 4 modalità diverse per l'apertura di un file:

- **“r” (Read)**, ovvero lettura (è il valore di default se non specifichiamo la modalità)  
Apre il file specificato dal parametro filename in **sola lettura**. Potremo leggere il contenuto del file solamente. Se il file non esiste avremo un errore
- **“a” (Append)**, apre il file specificato dal parametro filename per **aggiungerci** qualcosa, crea il file se esso non esiste.
- **“w” (Write)**, ovvero scrittura. Apre il file specificato dal parametro filename per **scriverci** qualcosa. Crea il file se esso non esiste. cancella il contenuto del file.
- **“x” (Create)**, ovvero in modalità di creazione. Crea un file con il nome specificato da filename. Se esiste già abbiamo un errore.

# Binario o Testo?

Possiamo anche sempre nel parametro modalità specificare se vogliamo aprire il file come file **di testo** “t” oppure **in binario** “b”.

Che differenza c'è?

# Binario o Testo?

Possiamo anche sempre nel parametro modalità specificare se vogliamo aprire il file come file **di testo** “t” oppure **in binario** “b”.

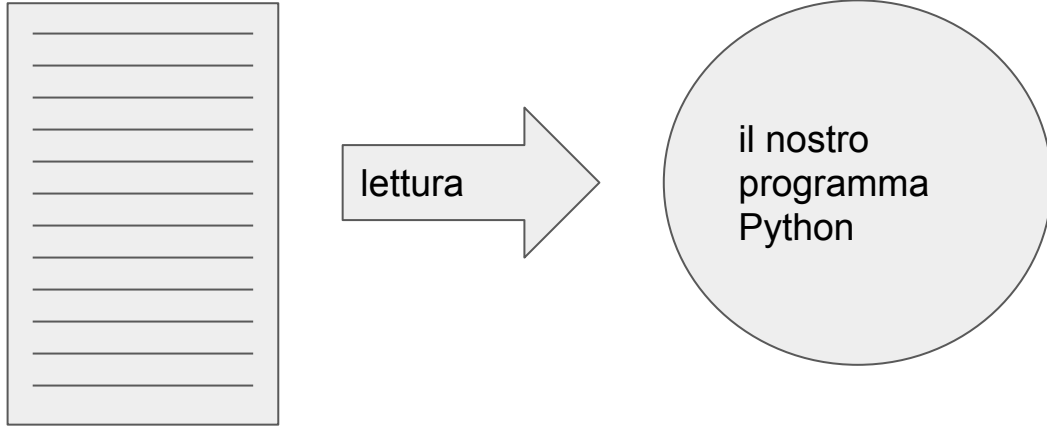
Che differenza c'è?

Se lo apriamo come **file di testo** (opzione di default) potremo visualizzare il suo contenuto come caratteri testuali (La funzione **open** interpreta per noi il contenuto del file direttamente a gruppi di 8 bit (1 byte) come caratteri secondo la codifica **UTF-8**).

Noi apriremo i file sempre come **file di testo**, per cui non ci sarà bisogno di specificare se in binario “b” o come testo “t”.

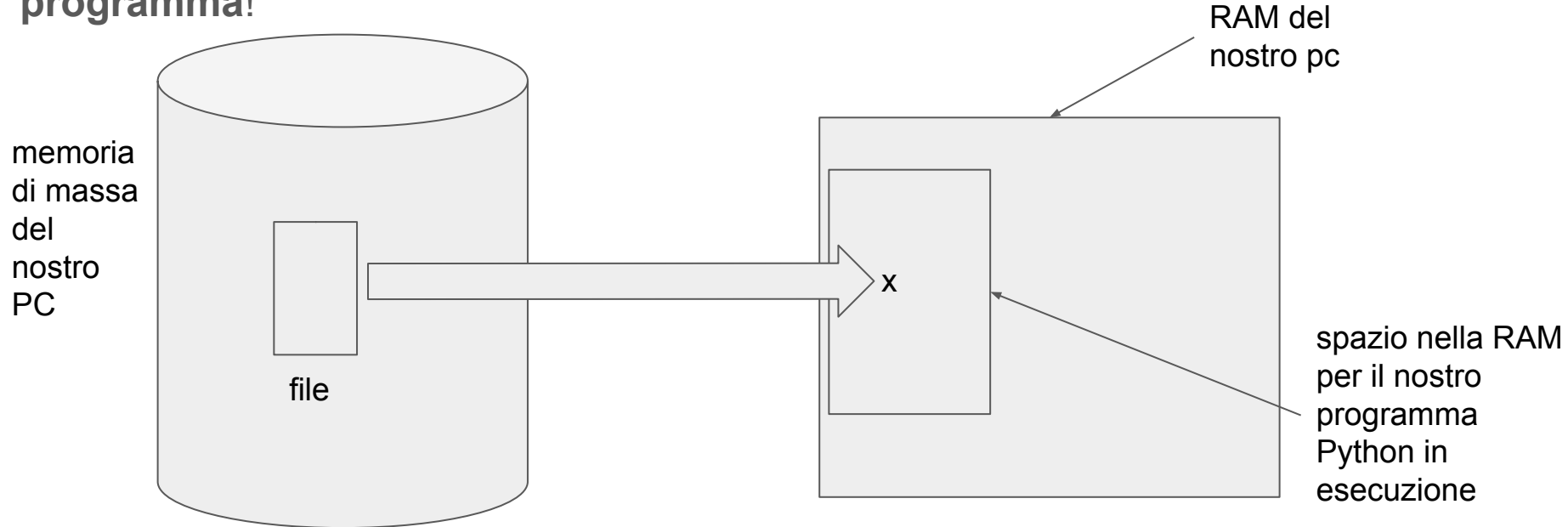
# Lettura di un file

Se apriamo un file in lettura è perché vogliamo **leggere il suo contenuto**.  
Vogliamo cioè **mettere il suo contenuto dentro ad una variabile del nostro programma!**



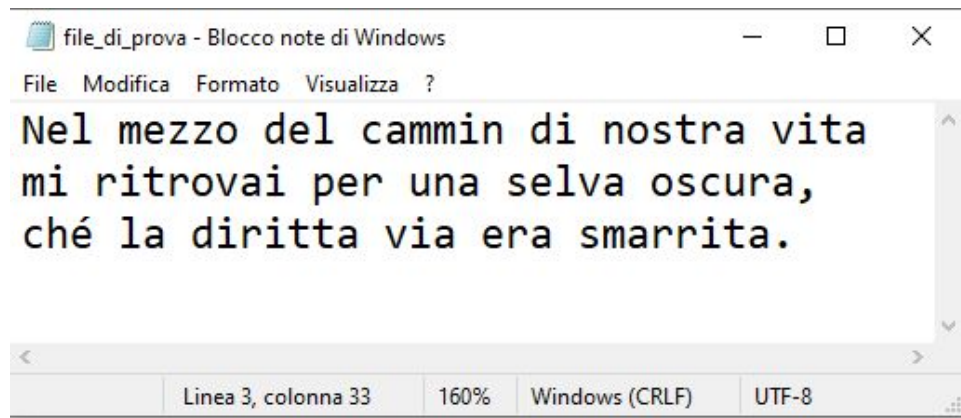
# Lettura di un file

Se apriamo un file in lettura è perché vogliamo **leggere il suo contenuto**.  
Vogliamo cioè **mettere il suo contenuto dentro ad una variabile del nostro programma!**



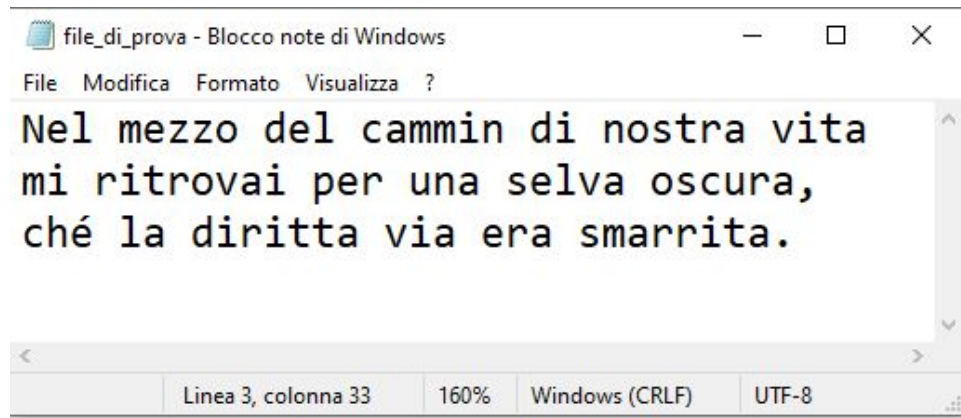
# read

```
def main():  
    f = open("file_di_prova.txt", "r")  
    x = f.read()  
    print(x)  
  
if __name__ == '__main__':  
    main()
```



# read

```
def main():  
    f = open("file_di_prova.txt", "r")  
    x = f.read()  
    print(x)  
  
if __name__ == '__main__':  
    main()
```



x sar  una **stringa** che contiene tutti i caratteri contenuti nel file "file\_di\_prova.txt"; **compresi gli spazi e i caratteri che indicano di andare a capo "\n"**

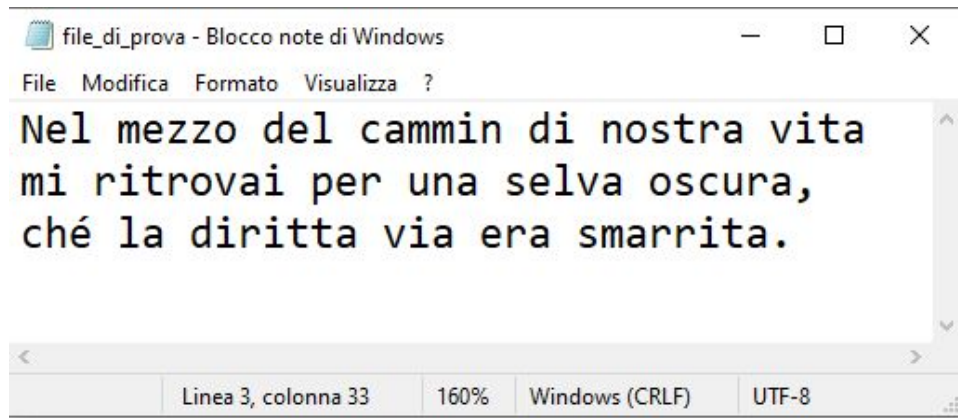


# readline

```
def main():  
    f = open("file_di_prova.txt", "r")  
    x = f.readline()  
    print(x)
```

```
if __name__ == '__main__':  
    main()
```

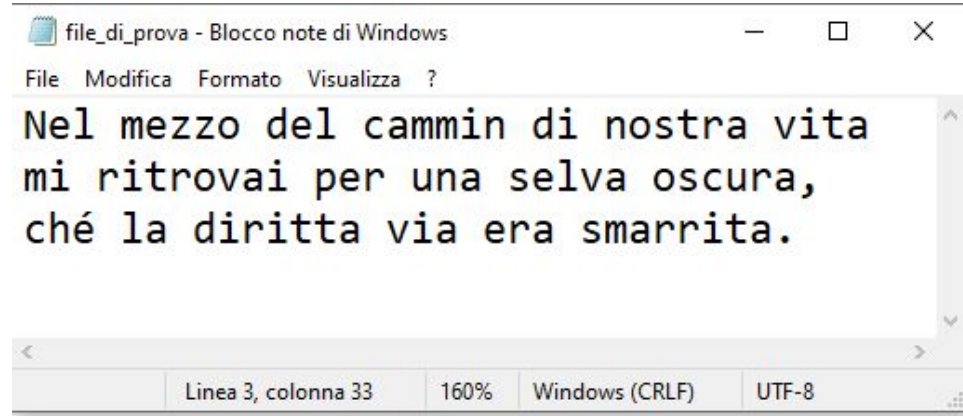
x sarà una **stringa** che contiene tutti i caratteri **della prima riga** del file "file\_di\_prova.txt"; **compresi gli spazi e i caratteri che indicano di andare a capo** "\n". Se viene chiamata una seconda volta leggerà la seconda riga e così via...



# Iterare le righe del file

```
def main():  
    f = open("file_di_prova.txt", "r")  
    for x in f:  
        print(x, end='')  
  
if __name__ == '__main__':  
    main()
```

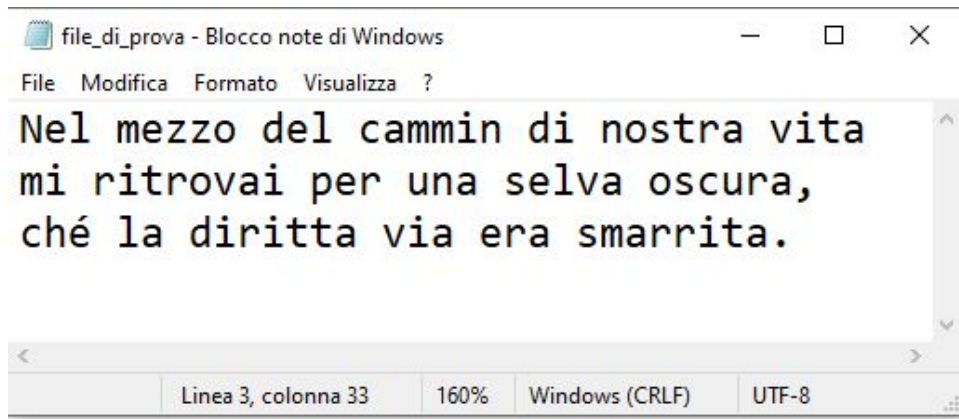
x sarà una **stringa** che contiene ad ogni iterazione tutti i caratteri **in ciascuna delle righe** del file "file\_di\_prova.txt"; **compresi gli spazi e i caratteri che indicano di andare a capo "\n"**



# readlines

```
def main():  
    f = open("file_di_prova.txt", "r")  
    x = f.readlines()  
    print(x)
```

```
if __name__ == '__main__':  
    main()
```



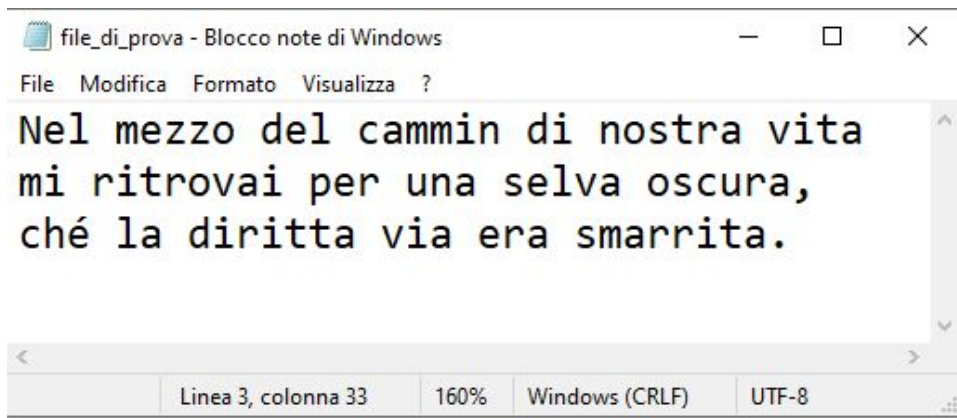
x sarà una **lista di stringhe** con  
**ciascuna delle righe** del file  
"file\_di\_prova.txt"; **compresi gli spazi e i**  
**caratteri** che indicano di andare a capo  
"**\n**"

# write

```
def main():  
    f = open("file_di_prova.txt", "w")  
    f.write("nuovo contenuto")  
  
if __name__ == '__main__':  
    main()
```

aprendo il file in modalità “w” possiamo scrivere sul file tramite la funzione **write**. Essa ha come parametro una stringa che rappresenta il testo che vogliamo scrivere sul file.

prima



dopo

