

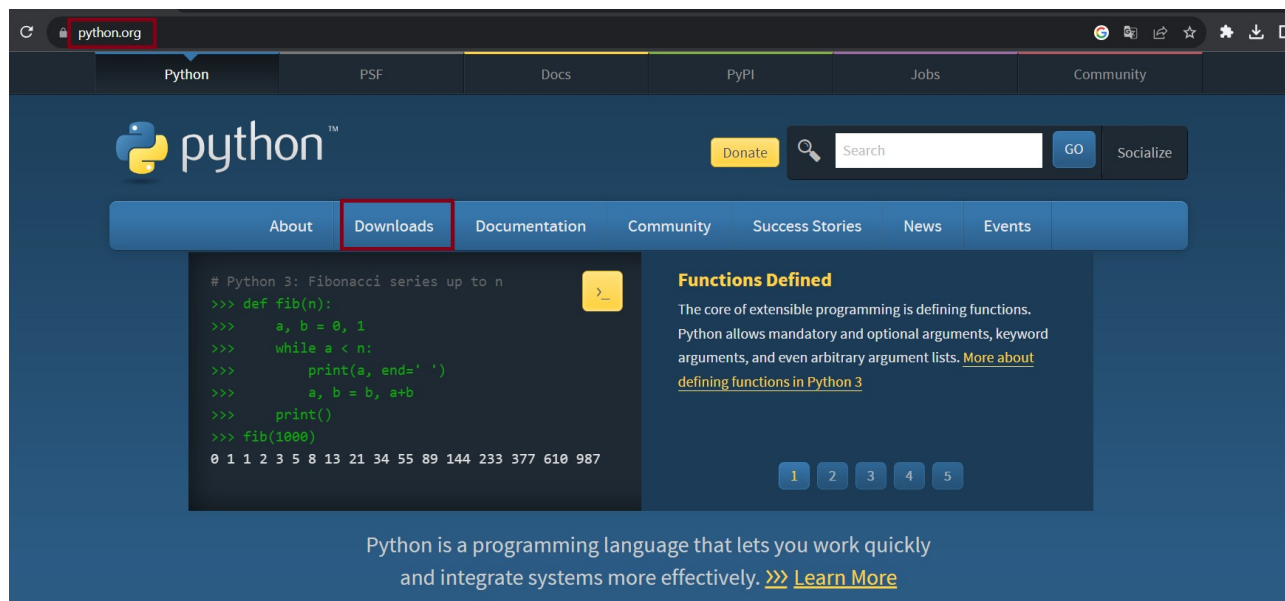
# APPUNTI PYTHON

## Indice generale

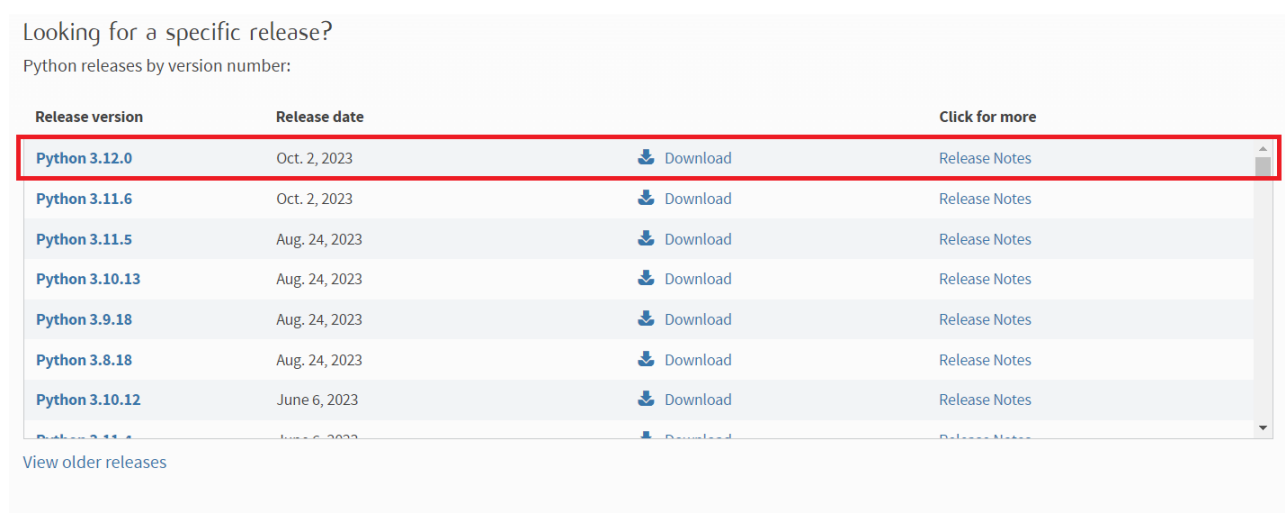
INSTALLAZIONE PYTHON SU WINDOWS.....	3
ESECUZIONE CODICE PYTHON.....	4
INPUT E OUTPUT.....	5
VARIABILI E TIPI DI DATI.....	8
GESTIRE LE ECCEZIONI.....	10
FORMATTAZIONE DELLE STRINGHE.....	11
LISTE E TUPLE.....	13
SET E FROZENSET.....	16
DIZIONARI.....	18
CICLO FOR.....	20
CICLO WHILE.....	21
ISTRUZIONI CONDIZIONALI E OPERATORI LOGICI.....	23
FUNZIONI.....	24
CLASSE E BASI DI PROGRAMMAZIONE.....	25
MODULI.....	26
COMANDO PIP.....	29

# INSTALLAZIONE PYTHON SU WINDOWS

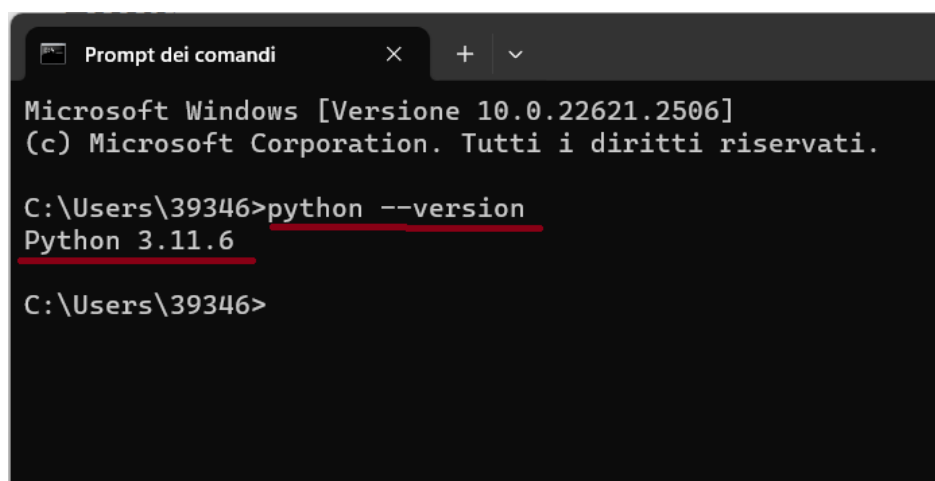
Nel caso in cui non sia installato Python su Windows, ci si deve recare sul sito ufficiale [python.org](https://python.org).



Ci rechiamo sulla sezione Downloads e scarichiamo l'ultima versione di Python disponibile.



Una volta scaricato l'eseguibile, lo possiamo installare. Per verificare la corretta installazione di Python, basta digitare sul prompt dei comandi il comando "python --version" e vedere se ci restituisce la versione.



# ESECUZIONE CODICE PYTHON

Per eseguire del codice Python possiamo avviare l'interprete interattivo di Python, digitando semplicemente "python" sul prompt dei comandi.

```
C:\Users\39346>python
Python 3.11.6 (tags/v3.11.6:8b6ee5b, Oct 2 2023, 14:57:12) [MSC v.1935 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> |
```

Adesso siamo all'interno dell'interprete interattivo di Python e possiamo digitare le istruzioni che ci servono. Ad esempio, possiamo creare una variabile e stamparne il risultato:

```
>>> hello = "Hello world"
>>> print(hello)
Hello world
```

Per uscire dall'interprete interattivo di Python, basta digitare il comando quit().

```
>>> quit()
```

```
C:\Users\39346>|
```

L'altro modo per eseguire del codice Python è inserirlo all'interno di un file di testo con estensione .py.

```
C:\Users\39346\Desktop\AppData Python\Script Python\script.py - Sublime Text (UNREGISTERED)
File Edit Selection Find View Goto Tools Project Preferences Help
script.py x
1 hello = "ciao python"
2 print(hello)
```

Dopodichè da terminale si può eseguire lo script, digitando il comando "python nome\_file.py"

```
C:\Users\39346>cd C:\Users\39346\Desktop\AppData Python\Script Python
C:\Users\39346\Desktop\AppData Python\Script Python>python script.py
ciao python
```

# INPUT E OUTPUT

L'input ci permette di inserire dei dati all'interno del nostro programma.

L'output, invece, ci permette di stampare su schermo il risultato del nostro programma, oppure in generale qualsiasi tipo di dato.

Con Python possiamo stampare dei dati su schermo, utilizzando la funzione print().

Parleremo più avanti di cosa è e come si realizza una funzione. Per adesso possiamo considerare una funzione come un programma già realizzato, alla quale possiamo eventualmente passare dei dati per ottenere un risultato.

In questo caso, i dati che passiamo alla funzione print() sono appunto quelli che vogliamo stampare su schermo.

```
print("ciao python")
```

```
C:\Users\39346\Desktop\AppData Python\Script Python>python print.py  
ciao python
```

In questo caso, il dato che abbiamo utilizzato è una stringa, cioè del testo. Con Python possiamo identificare una stringa, racchiudendola tra doppi apici.

Possiamo utilizzare la funzione print() anche per stampare dei numeri. In questo caso non vanno utilizzati i doppi apici.

```
print(5)
```

```
C:\Users\39346\Desktop\AppData Python\Script Python>python print.py  
5
```

Possiamo anche eseguire tutte le varie operazioni di base sui numeri all'interno della funzione print()

```
print(5 + 3)
```

```
C:\Users\39346\Desktop\AppData Python\Script Python>python print.py  
8
```

Qualsiasi linguaggio di programmazione ci permette di inserire dei commenti all'interno del programma, che non avranno alcuna utilità al fine della logica del programma, ma ci permetteranno di inserire delle annotazioni per ricordarci quello che stiamo facendo e farlo comprendere anche ad altri programmatori che dovranno lavorare sul nostro codice. Con Python possiamo inserire i commenti utilizzando il carattere # e poi scrivendo il commento.

```
#somma  
print(5 + 3)
```

```
C:\Users\39346\Desktop\AppData Python\Script Python>python print.py  
ciao python  
8
```

Quando svolgiamo l'operazione di divisione e il suo risultato è un numero con la virgola, possiamo andare a prendere anche soltanto la parte intera del risultato utilizzando //.

```
#divisione (solo intero)
print(10 // 3)
```

```
C:\Users\39346\Desktop\AppData Python\Script Python>python print.py
ciao python
3
```

Possiamo anche avere il resto della divisione utilizzando %.

```
#resto
print(10 % 3)
```

```
C:\Users\39346\Desktop\AppData Python\Script Python>python print.py
ciao python
1
```

Abbiamo anche l'elevamento a potenza utilizzando \*\*.

```
#potenza
print(10**2)
```

```
C:\Users\39346\Desktop\AppData Python\Script Python>python print.py
ciao python
100
```

Se proviamo ad utilizzare la somma su due stringhe, il risultato sarà l'unione tra le due stringhe.

```
#unione stringhe
print("5" + "2")
```

```
C:\Users\39346\Desktop\AppData Python\Script Python>python print.py
ciao python
52
```

Possiamo unire due stringhe anche tramite una virgola e, in questo caso, ci restituirà l'unione delle due stringhe con uno spazio in mezzo, senza doverlo inserire noi.


```
#unione stringhe
print("ciao", "python")
```

```
C:\Users\39346\Desktop\AppData Python\Script Python>python print.py
ciao python
```

Possiamo anche prendere un input da tastiera semplicemente utilizzando la funzione input().

```
#input  
name = input("Come ti chiami?")
```

```
C:\Users\39346\Desktop\AppData Python\Script Python>python print.py  
ciao python  
Come ti chiami? Stefano
```



Input inserito da  
tastiera

# VARIABILI E TIPI DI DATI

Una variabile ci permette di immagazzinare in memoria dei dati, che poi possono essere elaborati dal programma e modificati.

Come abbiamo già visto, possiamo utilizzare una variabile ad esempio per immagazzinare l'input che inseriamo da tastiera.

```
name = input("Come ti chiami? ")  
print(name)
```

```
C:\Users\39346\Desktop\AppData Python\Script Python>python variabili.py  
Come ti chiami? Stefano  
Stefano
```

Il nome della variabile va scelto da noi ed è buona norma utilizzare nomi che ci permettano di intuire il contenuto di una variabile. L'utilizzo del trattino basso per separare le parole che compongono il nome di una variabile è una convenzione di Python.

```
grandpa_age = input("Quanti anni ha tuo nonno? ")
```

Python è un linguaggio non tipizzato. Questo vuol dire che non bisogna specificare a priori il tipo di dato che una variabile deve contenere e possiamo anche passare da un tipo di dato ad un altro. Possiamo vedere che tipo di dato contiene una variabile utilizzando la funzione `type()`

```
var = "ciao"  
print(type(var))  
print(var)
```

```
C:\Users\39346\Desktop\AppData Python\Script Python>python variabili.py  
<class 'str'>  
ciao
```

Abbiamo già visto i principali tipi di dati di Python, quindi gli interi, i numeri in virgola mobile (float) e le stringhe. Un altro tipo di dato che spesso si rivela essere utile è il tipo booleano, che può assumere soltanto i valori `True` e `False`.

```
var = False  
print(type(var))  
print(var)
```

```
C:\Users\39346\Desktop\AppData Python\Script Python>python variabili.py  
<class 'bool'>  
False
```



Possiamo convertire un tipo di dato in un altro tipo con un'operazione chiamata casting.

```
var = 5  
print(type(var))  
var = str(var)  
print(type(var))
```

```
var = "10.5"  
print(type(var))  
var = float(var)  
print(type(var))
```

```
<class 'int'>  
<class 'str'>  
<class 'str'>  
<class 'float'>  
[Finished in 69ms]
```

# GESTIRE LE ECCEZIONI

Proviamo a vedere cosa succede se proviamo a convertire in un numero, una stringa che non contiene un numero.

```
var = "gatto"  
var = int(var)
```

```
Traceback (most recent call last):  
  File "C:\Users\39346\Desktop\AppData\Python\Script Python\eccezioni.py", line 2, in <module>  
    var = int(var)  
          ^^^^^^^  
ValueError: invalid literal for int() with base 10: 'gatto'  
[Finished in 72ms]
```

Ci è uscito fuori un errore.

In Python gli errori vengono anche chiamati eccezioni e imparare a leggerle è importante per riuscire a capire dove si sono presentati gli errori e il perché, quindi per capire cosa possiamo fare per risolverli.

Le informazioni importanti che un'eccezione contiene sono il tipo (nell'esempio ValueError) e la sua descrizione (nell'esempio ci sta informando che la funzione int() non ha potuto eseguire il casting perché la stringa non contiene un numero).

La prima cosa da fare quando si presenta un'eccezione è ovviamente andare a trovare l'errore (o bug) e risolverlo (fixarlo).

Un modo per essere sicuri che chi utilizza il nostro programma non possa imbattersi in un errore del genere, è gestire preventivamente eventuali eccezioni utilizzando dei blocchi try except.

```
num = input("Inserisci un numero: ")  
try:  
    print(type(num))  
    num = int(num)  
    print("Il numero che hai inserito è", num)  
except ValueError:  
    print("Il dato che hai inserito non è un numero")
```

```
● Inserisci un numero: gatto  
  <class 'str'>  
  Il dato che hai inserito non è un numero
```

# FORMATTAZIONE DELLE STRINGHE

Abbiamo già visto come possiamo unire più stringhe, utilizzando l'operatore somma.

```
cat = "Elon"

print("Il mio gatto si chiama "+cat+"")
```

```
Il mio gatto si chiama Elon
PS C:\Users\39346>
```

Proviamo a vedere cosa succede se volessimo unire delle stringhe con dei numeri.

```
cat = "Elon"
age = 2

print("Il mio gatto si chiama "+cat+" ed ha "+age+" anni")
```

```
Traceback (most recent call last):
  File "c:\Users\39346\Desktop\Appunti Python\Script Python\formattazione_stringhe.py", line 4, in <module>
    print("Il mio gatto si chiama "+cat+" ed ha "+age+" anni")
    ~~~~~^~~~~~
TypeError: can only concatenate str (not "int") to str
```

Come possiamo vedere, abbiamo ottenuto un'eccezione. Il fatto è che non possiamo semplicemente unire stringhe e numeri utilizzando l'operatore somma, dato che per i numeri tale operatore viene utilizzato per eseguire un'operazione aritmetica.

Quello che invece possiamo fare è utilizzare il casting per convertire un numero in una stringa e quindi unirlo ad un'altra stringa.

```
cat = "Elon"
age = 2

print("Il mio gatto si chiama "+cat+" ed ha "+str(age)+" anni")
```

```
Il mio gatto si chiama Elon ed ha 2 anni
PS C:\Users\39346>
```

In questo caso il tutto funzionerebbe senza problemi, però esiste una tecnica migliore che ci permette di fare questa stessa cosa in maniera più efficiente, ovvero tramite la formattazione.

```
cat = "Elon"
age = 2

#Formattazione
print("Il mio gatto si chiama %s ed ha %d anni" % (cat, age))
```

```
Il mio gatto si chiama Elon ed ha 2 anni
PS C:\Users\39346>
```

Come possiamo vedere il risultato è lo stesso. Quello che abbiamo fatto è inserire dei placeholder, cioè dei caratteri speciali, che indicano che in quelle posizioni andranno inserite rispettivamente una stringa %s e un intero %d. Nella seconda parte abbiamo inserito le variabili (è importante rispettare l'ordine in cui le inseriamo) con cui rimpiazzeremo i placeholder, fatte precedere da un simbolo di %.

Ovviamente avremmo potuto fare la stessa cosa con un numero indeterminato di variabili, inserendo i corrispettivi placeholder. Abbiamo un placeholder diverso per quasi ogni tipo di dato che Python ci mette a disposizione.

Se volessimo utilizzare anche un numero con la virgola, il placeholder da utilizzare sarebbe %f.

```
cat = "Elon"  
age = 2  
weight = 5.678
```

```
#Formattazione  
print("Il mio gatto si chiama %s, ha %d anni e pesa %f kg" % (cat, age, weight))
```

```
Il mio gatto si chiama Elon, ha 2 anni e pesa 5.678000 kg  
PS C:\Users\39346>
```

Come possiamo vedere, di default il numero decimale viene arrotondato a 6 cifre dopo la virgola. Possiamo comunque decidere noi a quante cifre dopo la virgola arrotondare il numero decimale.

```
cat = "Elon"  
age = 2  
weight = 5.678
```

```
#Formattazione  
print("Il mio gatto si chiama %s, ha %d anni e pesa %.2f kg" % (cat, age, weight))
```

```
Il mio gatto si chiama Elon, ha 2 anni e pesa 5.68 kg  
PS C:\Users\39346>
```

Come possiamo vedere, abbiamo arrotondato il nostro numero decimale a soltanto due cifre dopo la virgola. Questo è un modo per formattare stringhe old school, cioè utilizzato anche da altri linguaggi di programmazione.

In Python abbiamo anche altri modi più moderni per formattare le stringhe:

```
cat = "Elon"  
age = 2  
weight = 5.678
```

```
#Formattazione (old school)  
print("Il mio gatto si chiama %s, ha %d anni e pesa %.2f kg" % (cat, age, weight))
```

```
#Formattazione (new school)  
print("Il mio gatto si chiama {cat}, ha {age} anni e pesa {weight} kg".format(cat=cat, age=age, weight=weight))
```

```
#Formattazione Python 3.6  
print(f"Il mio gatto si chiama {cat}, ha {age} anni e pesa {weight} kg")
```

```
Il mio gatto si chiama Elon, ha 2 anni e pesa 5.68 kg  
Il mio gatto si chiama Elon, ha 2 anni e pesa 5.678 kg  
Il mio gatto si chiama Elon, ha 2 anni e pesa 5.678 kg  
PS C:\Users\39346>
```

# LISTE E TUPLE

Con Python possiamo creare delle sequenze di dati con 3 costrutti che il linguaggio ci mette a disposizione, cioè le liste, le tuple e i set.

Possiamo creare una lista con Python, racchiudendone gli elementi tra parentesi quadre.

```
#Liste
my_list = [10, 5, 8, 3, 11, 2]
print(type(my_list))
lunghezza_lista = len(my_list)
print("La lista è lunga",str(lunghezza_lista))
```

```
#indexing
print(my_list[0])
print(my_list[1])
print(my_list[-1])
```

```
<class 'list'>
La lista è lunga 6
10
5
2
PS C:\Users\39346>
```

Possiamo estrarre anche solo una parte della lista con un'operazione chiamata slicing, in cui dovremo specificare due indici, ovvero quello di inizio e quello di fine.

```
my_list = [10, 5, 8, 3, 11, 2]
```

```
#slicing
print(my_list[0:3])
print(my_list[:5])
print(my_list[2:])
print(my_list[:])
```

```
[10, 5, 8]
[10, 5, 8, 3, 11]
[8, 3, 11, 2]
[10, 5, 8, 3, 11, 2]
PS C:\Users\39346>
```

Possiamo anche specificare un terzo valore, che ci permette di indicare il senso nella quale effettuare la selezione. Di default è definito a 1, quindi la lista viene selezionata da sinistra a destra. Se invece lo definiamo a -1, la selezione verrà effettuata da destra a sinistra, invertendo di fatto gli elementi della lista.

```
my_list = [10, 5, 8, 3, 11, 2]
```

```
print(my_list[::-1])
```

```
[2, 11, 3, 8, 5, 10]
PS C:\Users\39346>
```

Per modificare un valore all'interno della lista semplicemente effettuiamo la selezione della lista (sia che si tratti di un valore singolo o di un sottoinsieme) ed effettuiamo l'assegnazione con il nuovo valore.

```
my_list = [10, 5, 8, 3, 11, 2]
```

```
#modifica del primo valore della lista  
my_list[0] = 0
```

```
#modifica degli ultimi due valori della lista  
my_list[-2:] = [7,1]  
print(my_list[:])
```

```
[0, 5, 8, 3, 7, 1]  
PS C:\Users\39346>
```

Per verificare se un valore è presente all'interno di una lista, si utilizza la keyword "in"

```
#ricerca elemento nella lista  
animals = ["cane", "gatto", "topo"]  
print("uomo" in animals)  
print("gatto" in animals)
```

```
False  
True  
PS C:\Users\39346>
```

Per eliminare degli elementi dalla lista, possiamo eseguire la rimozione o per valore, o per indice.

```
animals = ["cane", "gatto", "topo"]
```

```
#rimozione per valore  
animals.remove("gatto")  
print(animals)
```

```
#rimozione per indice  
animal = animals.pop(1)  
print(animals)  
print(animal)
```

```
['cane', 'topo']  
['cane']  
topo  
PS C:\Users\39346>
```

Possiamo aggiungere degli elementi ad una lista già creata utilizzando il metodo append(), che aggiungerà l'elemento al termine della lista.

```
animals = ["cane"]
```

```
#aggiunta elemento nella lista  
animals.append("bestia demoniaca")  
print(animals)
```

```
['cane', 'bestia demoniaca']  
PS C:\Users\39346>
```

Se invece volessimo aggiungere gli elementi in una posizione stabilita da noi, dobbiamo utilizzare il metodo `insert()`, dove andiamo a specificare la posizione dell'elemento che andremo a definire.

```
animals = ["cane", "bestia demoniaca"]

#aggiunta elemento nella lista in una posizione predefinita
animals.insert(1, "topo")
print(animals)

['cane', 'topo', 'bestia demoniaca']
PS C:\Users\39346>
```

Vediamo cosa sono le tuple ora.

Una tupla è un tipo molto simile alla lista. La differenza principale consiste nel fatto che le tuple non sono modificabili, quindi una volta definite, non possiamo più modificare il loro contenuto.

Possiamo definire una tupla inserendo i suoi elementi tra parentesi tonde.

```
#Tuple
my_tuple = (10, 5, 8, 3, 9)
print(type(my_tuple))
print(my_tuple)
print(my_tuple[1])
print(my_tuple[3])
```

```
<class 'tuple'>
(10, 5, 8, 3, 9)
5
(10, 5, 8)
PS C:\Users\39346>
```

Se proviamo a modificare un elemento della tupla, otteniamo un'eccezione che ci informa che è impossibile assegnare nuovi valori ad una tupla che abbiamo già definito.

```
my_tuple[0] = 0

Traceback (most recent call last):
  File "c:\Users\39346\Desktop\Appunti Python\Script Python\liste_e_tuple.py", line 56, in <module>
    my_tuple[0] = 0
    ~~~~~^~~~~
TypeError: 'tuple' object does not support item assignment
PS C:\Users\39346>
```

Sia le tuple che le liste possono contenere tipi diversi di dati al loro interno, però per convenzione vengono utilizzate solo le tuple in questi casi. Vediamo due funzioni utili che possiamo utilizzare sia per le tuple, che per le liste:

```
my_tuple = (10, 5, 8, "ciao", 3, 9, "ciao")

#ottenere l'indice di un elemento
indice = my_tuple.index("ciao")
print(str(indice))

#ottenere il numero di volte in cui un elemento è presente nella lista/tupla
n_occorrenze = my_tuple.count("ciao")
print(str(n_occorrenze))
```

```
3
2
PS C:\Users\39346>
```

# SET E FROZENSET

Un set è un insieme di elementi unici e non ordinati. Questo vuol dire che non può comparire due volte lo stesso elemento all'interno di un set e che al suo interno l'ordine degli elementi non ha alcuna importanza.

Possiamo creare un set inserendo i suoi elementi tra parentesi graffe.

```
#Set
names = {"Giuseppe", "Federico", "Antonino", "Matteo", "Federico"}
print(names)
```

```
• {'Federico', 'Antonino', 'Matteo', 'Giuseppe'}
○ PS C:\Users\39346>
```

Come possiamo vedere, gli elementi sono stati mescolati e quello duplicato è stato rimosso.

Possiamo aggiungere un nuovo elemento al nostro set utilizzando il metodo add()

```
#aggiunta di un elemento
names.add("Lorenzo")
print(names)
```

```
{'Giuseppe', 'Lorenzo', 'Federico', 'Antonino', 'Matteo'}
PS C:\Users\39346>
```

Come possiamo vedere, l'elemento è stato aggiunto in una posizione totalmente casuale.

Possiamo rimuovere un elemento del set utilizzando il metodo remove(), che abbiamo già visto per le liste.

```
#rimuovere un elemento
names.remove("Antonino")
print(names)
```

```
{'Federico', 'Matteo', 'Lorenzo', 'Giuseppe'}
PS C:\Users\39346>
```

Qualora tentassimo di rimuovere un elemento che non è presente all'interno del set, otterremmo un errore. Per aggirare questo problema, dobbiamo utilizzare il metodo discard(), piuttosto che remove(). In altre parole, se non siamo sicuri che l'elemento è presente all'interno del nostro set, utilizziamo discard()

```
names.discard("Paolo")
print(names)
```

```
{'Federico', 'Matteo', 'Lorenzo', 'Giuseppe'}
PS C:\Users\39346>
```

Possiamo anche estrarre un elemento da un set utilizzando il metodo pop(), che abbiamo già visto. Se non indichiamo al metodo pop() che elemento estrarre, ne sceglierà uno in maniera casuale.

```
#estrazione di un elemento
name = names.pop()
print(name)
print(names)
```

```
Matteo
{'Federico', 'Giuseppe', 'Lorenzo'}
PS C:\Users\39346>
```



Qualora volessimo svuotare un set, possiamo utilizzare il metodo `clear()`

```
#svuotare il set
names.clear()
print(names)
```

```
set()
PS C:\Users\39346> 
```

Possiamo anche convertire una lista in un set, e viceversa, tramite il casting

```
#conversione da lista a set
names_list = ["Giuseppe", "Federico", "Antonino", "Matteo", "Federico"]
print(names_list)
```

```
names_set = set(names_list)
print(names_set)
```

```
#conversione da set a lista
names_list = list(names_set)
print(names_list)
```

```
['Giuseppe', 'Federico', 'Antonino', 'Matteo', 'Federico']
{'Giuseppe', 'Antonino', 'Federico', 'Matteo'}
['Giuseppe', 'Antonino', 'Federico', 'Matteo']
PS C:\Users\39346> 
```

Come per liste e tuple, anche per i set abbiamo la controparte immutabile, denominata `frozenset`.

Possiamo creare un `frozenset` utilizzando la funzione `frozenset()` e passando il set al suo interno.

```
#frozenset
names = frozenset({"Giuseppe", "Federico", "Antonino", "Matteo", "Federico"})
```

In questo caso non possiamo più modificare gli elementi.

# DIZIONARI

I dizionari ci permettono di immagazzinare dati in un formato chiave-valore.

Un dizionario si crea racchiudendo i suoi elementi tra parentesi graffe, come abbiamo già visto per i set, solo che questa volta gli elementi saranno in formato chiave-valore, dove il primo elemento sarà la chiave e il secondo sarà il valore, separati da i due punti.

```
items = {'latte':3, 'riso':2, 'tofu':5}
print(items)

{'latte': 3, 'riso': 2, 'tofu': 5}
PS C:\Users\39346>
```

Possiamo accedere al singolo elemento, utilizzando la sua chiave.

```
item = items["latte"]
print(item)

3
PS C:\Users\39346>
```

Allo stesso modo possiamo modificare un valore del dizionario, sempre utilizzando la sua chiave.

```
items["latte"] = 2
item = items["latte"]
print(item)

2
PS C:\Users\39346>
```

In ogni momento possiamo creare un nuovo elemento, semplicemente effettuando un'assegnazione.

```
items["cereali"] = 1
print(items)

{'latte': 2, 'riso': 2, 'tofu': 5, 'cereali': 1}
PS C:\Users\39346>
```

Possiamo anche inserire all'interno di un elemento del nostro dizionario, un altro dizionario.

```
items['yogurt'] = {'fragola':8, 'bianco':3}
print(items)
item = items['yogurt']['fragola']
print(item)

{'latte': 2, 'riso': 2, 'tofu': 5, 'cereali': 1, 'yogurt': {'fragola': 8, 'bianco': 3}}
8
PS C:\Users\39346>
```

Se volessimo ottenere soltanto le chiavi del nostro dizionario, possiamo utilizzare il metodo `keys()`. Questo metodo ci restituisce un oggetto di tipo `dict_keys`

```
print(items.keys())

dict_keys(['latte', 'riso', 'tofu', 'cereali', 'yogurt'])
PS C:\Users\39346>
```

Se invece volessimo ottenere soltanto i valori, possiamo utilizzare il metodo values()

```
print(items.values())
```

```
dict_values([2, 2, 5, 1, {'fragola': 8, 'bianco': 3}])  
PS C:\Users\39346>
```

# CICLO FOR

I cicli ci permettono di eseguire una serie di istruzioni in maniera iterativa.

Un esempio classico di ciclo è il ciclo for, utilizzato da moltissimi linguaggi di programmazione.

```
n = int(input("Fino a che numero vuoi stampare ? "))
```

```
for i in range(0,n):  
    print(i)
```

```
Fino a che numero vuoi stampare ? 10  
0  
1  
2  
3  
4  
5  
6  
7  
8  
9  
PS C:\Users\39346>
```

Nell'esempio precedente, a ogni iterazione del ciclo il valore di i viene incrementato di un valore fino a quando non arriverà al valore di n.

Possiamo utilizzare il ciclo for anche per le strutture dati, per esempio per stampare ogni singolo elemento di una lista.

```
#iterazione sulle liste  
shopping_list = ["tofu", "latte di soia", "riso basmati", "yogurt greco"]
```

```
for index, element in enumerate(shopping_list):  
    print("%d) %s" % (index+1, element))
```

```
1) tofu  
2) latte di soia  
3) riso basmati  
4) yogurt greco  
PS C:\Users\39346>
```

# CICLO WHILE

Per definire un ciclo while, dobbiamo conoscere le espressioni booleane.

Le espressioni booleane sono delle operazioni che ci permettono di eseguire un confronto tra i dati.

```
#uguaglianza
is_equal = 5 == 5
print(is_equal)
```

```
#disuguaglianza
is_different = "casa" != "gatto"
print(is_different)
```

```
#maggioranza
is_greater = 6 > 9
print(is_greater)
```

```
is_greater_or_equal = 6 >= 6
print(is_greater_or_equal)
```

```
#minoranza
is_less = 6 > 9
print(is_less)
```

```
is_less_or_equal = 6 <= 6
print(is_less_or_equal)
```

```
True
True
False
True
False
True
PS C:\Users\39346>
```

Queste sono le principali espressioni booleane che abbiamo con Python.

Adesso possiamo definire il nostro ciclo while per contare dei numeri fino a un valore definito da noi.

```
#ciclo while
n = int(input("Fino a che numero vuoi contare? "))
```

```
i = 0
```

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

```
Fino a che numero vuoi contare? 10
0
1
2
3
4
5
6
7
8
9
PS C:\Users\39346>
```

Possiamo saltare delle esecuzioni del ciclo while tramite il continue, oppure possiamo anche uscire dal ciclo tramite il break.

```
#ciclo while  
n = int(input("Fino a che numero vuoi contare? "))
```

```
i = 0
```

```
while i < n:  
    if(i % 3 == 0): #multipli di 3  
        i += 2  
        continue  
    print(i)  
    i += 2
```

```
Fino a che numero vuoi contare? 30  
2  
4  
8  
10  
14  
16  
20  
22  
26  
28  
PS C:\Users\39346>
```

# ISTRUZIONI CONDIZIONALI E OPERATORI LOGICI

Le istruzioni condizionali ci permettono di eseguire delle condizioni solo se una o più condizioni sono soddisfatte.

```
n = int(input("Inserisci un numero positivo: "))
```

```
if(n < 0):  
    print("%d non è un numero positivo" % n)  
elif(n % 2 == 0):  
    print("%d è un numero pari" % n)  
else:  
    print("%d è un numero dispari" % n)
```

```
Inserisci un numero positivo: 10  
10 è un numero pari  
PS C:\Users\39346> 
```

```
#operatori logici  
1 == 4 and 2 == 2  
1 == 2 or 2 == 1  
not 2 == 1
```

# FUNZIONI

Le funzioni ci permettono di riutilizzare blocchi di codice, prendendo eventualmente dei dati in ingresso, chiamati argomenti o parametri, e restituendo (sempre eventualmente) un output.

```
#Funzioni
def calcolo_area_triangolo(b,h):
    area = b * h / 2
    return area
```

```
b = 5
h = 3
```

```
area = calcolo_area_triangolo(b,h)
print(area)
```

```
7.5
PS C:\Users\39346\Desktop\AppData Python\Script Python>
```



# CLASSE E BASI DI PROGRAMMAZIONE

Python supporta molti paradigmi di programmazione che caratterizzano lo stile del codice. Il paradigma basato sulle funzioni è conosciuto come programmazione procedurale.

Un altro paradigma molto utilizzato è la programmazione orientata agli oggetti. Gli oggetti ci permettono di racchiudere funzioni e variabili all'interno di un'unica entità e rendono il codice maggiormente riutilizzabile e molto più semplice da mantenere.

Per creare un oggetto, dobbiamo innanzitutto creare la classe che lo rappresenterà.

Le funzioni di una classe vengono chiamate metodi. Il primo parametro di un metodo è sempre self, che è una variabile che ci permette di identificare i metodi e i parametri della classe stessa.

```
#Programmazione ad oggetti e Classi
class Triangle:
```

```
    #attributi della Classe
    def __init__(self, a, b, c, h):
        self.a, self.b, self.c, self.h = a, b, c, h
```

```
    #metodi della Classe
    def area(self):
        return self.b * self.h / 2

    def perimeter(self):
        return self.a + self.b + self.c
```

```
#istanzio un nuovo oggetto
triangle = Triangle(4, 3, 8, 5)
```

```
area = triangle.area()
perimeter = triangle.perimeter()
```

```
print(area)
print(perimeter)
```

```
7.5
15
PS C:\Users\39346\Desktop\AppData Python\Script Python>
```

# MODULI

I moduli ci permettono di organizzare il codice dei nostri programmi in più file, in modo di separarne le parti e garantire una riutilizzabilità ottimale del codice.

Questa cosa potrebbe sembrare insensata per quello che abbiamo fatto finora, ma in realtà, quando andiamo a sviluppare un software che ha almeno decine di migliaia di righe di codice, non possiamo fare tutto all'interno di un unico file.

Un modulo non è altro che un file Python, che si deve trovare nella stessa directory del file all'interno della quale lo vogliamo utilizzare, oppure in questa directory standard di Python `/lib/site-packages`

Creiamo un semplice file Python di nome `script.py`, con al suo interno un'unica funzione che stampa una stringa.

`script.py`

```
def hello_world():  
    hello = "Hello world!"  
    print(hello)
```

Per importare un modulo all'interno di un altro file Python, dobbiamo utilizzare la keyword `import`, seguita dal nome del file (senza scrivere l'estensione)

`modulo.py`

```
import script
```

```
script.hello_world()
```

```
Hello world!  
PS C:\Users\39346\Desktop\AppData Python>
```

Il principio è lo stesso anche quando si importano moduli che contengono delle classi.

`geometry.py`

```
class Triangolo:  
    """Una semplice classe che calcola le misure di un triangolo"""
```

```
    def __init__(self, a, b, c, h):  
        self.a, self.b, self.c, self.h = a, b, c, h
```

```
    def area(self):  
        """Calcolo l'area del triangolo"""
```

```
        return float(self.b) * float(self.h) / 2  
    def perimetro(self):  
        """Calcolo il perimetro del triangolo"""
```

```
return self.a + self.b + self.c
```

```
class Quadrato:
```

```
    """Una semplice classe che calcola le misure di un quadrato"""
```

```
    def __init__(self, l):
```

```
        self.l = l
```

```
    def area(self):
```

```
        """Calcolo l'area del quadrato"""
```

```
        return self.l ** 2
```

```
    def perimetro(self):
```

```
        """Calcolo il perimetro del quadrato"""
```

```
        return self.l * 4
```

```
class Rettangolo:
```

```
    """Una semplice classe che calcola le misure di un rettangolo"""
```

```
    def __init__(self, b, h):
```

```
        self.b, self.h = b, h
```

```
    def area(self):
```

```
        """Calcolo l'area del rettangolo"""
```

```
        return self.b * self.h
```

```
    def perimetro(self):
```

```
        """Calcolo il perimetro del rettangolo"""
```

```
        return 2 * (self.b + self.h)
```

modulo.py

```
import geometria
```

```
quadrato = geometria.Quadrato(2)  
print(quadrato.area())
```

```
4  
PS C:\Users\39346\Desktop\AppData Python>
```

È possibile anche utilizzare una o più funzioni (o classi) del modulo importato, senza dover ogni volta specificare il nome di tale modulo, tramite la keyword from sull'import.

modulo.py

```
from geometria import Triangolo, Quadrato, Rettangolo  
triangolo = Triangolo(3,4,5,7)  
print(triangolo.area())  
print(triangolo.perimetro())
```

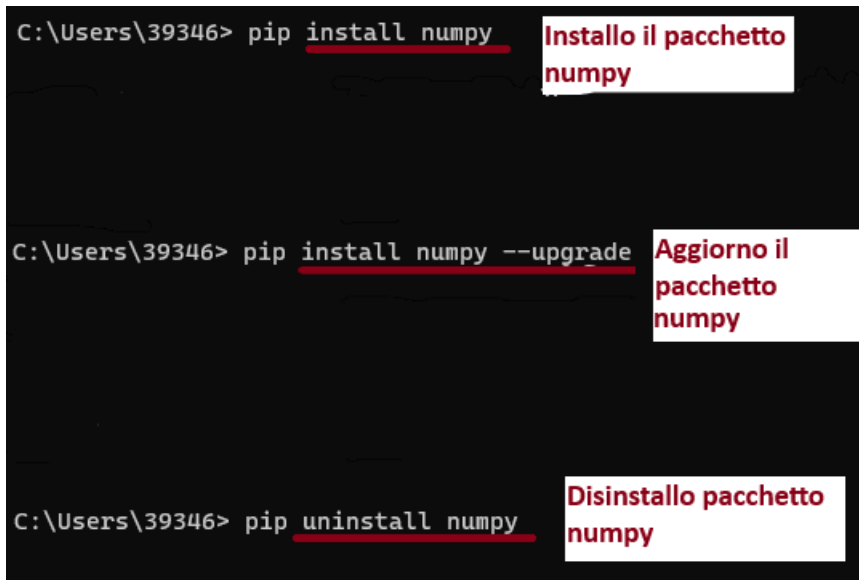
```
14.0  
12  
PS C:\Users\39346\Desktop\AppData Python>
```

Tramite l'import è possibile importare anche moduli delle Standard Library di Python, che sono elencate facendo una ricerca in rete.

# COMANDO PIP

Python ci permette anche di utilizzare moduli creati da altri sviluppatori e sono contenuti all'interno del Python Package Index.

Per utilizzare questi moduli, dobbiamo prima installarli utilizzando pip, il gestore di pacchetti di Python per eccellenza, e dobbiamo farlo da riga di comando.



```
C:\Users\39346> pip install numpy
```

Installo il pacchetto numpy

```
C:\Users\39346> pip install numpy --upgrade
```

Aggiorno il pacchetto numpy

```
C:\Users\39346> pip uninstall numpy
```

Disinstallo pacchetto numpy

Una volta installato il modulo (creato da altri sviluppatori) che ci interessa, possiamo importarlo con il solito procedimento.

```
import numpy
```