

Appunti di scuola

Python

Diomede Mazzone

2020, ver. 0.46



Sommario

Sommario	1
Linguaggi di programmazione	4
Linguaggi interpretati o compilati	4
Installazione, Interfaccia ed ambiente di sviluppo	5
Personalizzare L'IDLE	8
Eeguire istruzioni	10
Variabili, strutture dati e istruzioni	14
Sintassi di base	14
Variabili	15
Casting	17
Operatori	18
Controllo di flusso	19
Selezione	19
Iterazioni	20
Range	21
Input ed output	22
Print	22
Input e gestione degli errori	23
Variabili strutturate	26
Stringhe	26
Tuple	31
Liste	34
Dictionary	36
Funzioni	39
Programmazione ad oggetti (OOP)	39
Caratteristiche della programmazione ad oggetti	41
Definizione di classe	41
Incapsulamento e Information hiding	42
Il metodo Costruttore	43
Attributi di classe e d istanza	44
Ereditarietà	48
Polimorfismo	51

Moduli	52
definire funzioni	52
Utilizzo dei moduli	55
Gestione e salvataggio dei moduli	56
Spazio dei nomi	57
Visibilità delle variabili	57
Variabili dei moduli	59
Moduli personali	59
I moduli standard	59
Modulo Math	59
Modulo Random	60
Modulo Time	61
Modulo Exception	62
Modulo OS	65
Turtle	66
Figure geometriche	69
GUI - interfaccia utente	72
Guizero	72
Menu	76
ListBox	78
Acquisire parametri dall'utente	80
Elaborazione dei dati e grafici	81
NumPy	81
NumPy Array Object	82
Matplotlib	83
Struttura dei grafici	84
Figure	87
Axes	90
Personalizzare i grafici	92
Linea	93
Marker, i punti della curva	96
Assi, proprietà e personalizzazioni	98
Legenda	101
Griglia e sfondo	103
Formattazione del testo ed annotazioni	108
Pandas	109

Accesso ai file	110
Introduzione all'importazione di dati	110
CSV	110
JSON	115

Linguaggi di programmazione

Un algoritmo per essere eseguito da una macchina, restituendo in output sempre lo stesso risultato, ha bisogno di essere descritto in modo non ambiguo, attraverso una sintassi che permetta all'esecutore di interpretare in modo corretto e preciso le istruzioni. Un linguaggio con una sintassi di questo tipo non può essere in linguaggio naturale, ma deve essere necessariamente più vincolante. A questo scopo si distinguono due tipi di linguaggi, di basso livello e di alto livello. Quando la sintassi è scritta per essere eseguita direttamente da un microprocessore, viene detto un **linguaggio di basso livello** o **linguaggio macchina**. Viceversa, se si scrive un algoritmo attraverso una sintassi abbastanza comprensibile per un essere umano, siamo in presenza di un linguaggio di alto livello.

Linguaggi interpretati o compilati

I linguaggi di programmazione di alto livello possono essere distinti, dal punto di vista tecnologico e della loro esecuzione, in due macro-categorie: i **linguaggi interpretati** ed i **linguaggi compilati**, anche se come vedremo esistono linguaggi che fondono le caratteristiche di entrambi le categorie.

Un linguaggio di programmazione viene detto interpretato quando il software che lo deve decodificare, per poi farlo eseguire alla macchina sul quale è installato, si comporta come un interprete durante una diretta televisiva. In questo caso, infatti, l'ambiente di sviluppo interpreta riga per riga le istruzioni e si ferma, restituendo errore, solo nel caso in cui l'istruzione che sta leggendo non è comprensibile. Il software che decodifica le istruzioni viene anche detto **interprete**.

Un linguaggio di programmazione viene detto compilativo quando si comporta come un traduttore di un testo scritto, cerca prima di comprendere l'intero algoritmo alla ricerca di eventuali errori di sintassi e poi lo esegue. Risulta chiaro che questo approccio non permette di eseguire nessuna istruzione nel caso fosse presente un errore anche solo alla fine dell'algoritmo. I software che si occupano di decodificare un algoritmo in questo modo si chiamano **compilatori**.

Bisogna comunque sottolineare che i due approcci non sono paritetici ed entrambi hanno aspetti vantaggiosi e svantaggiosi. Un software realizzato attraverso un linguaggio interpretato risulta **portabile** su più sistemi operativi, in quanto basterà avere un interprete di quel linguaggio installato sulla macchina e quindi il codice non è dipendente dal Sistema sul quale l'interprete viene eseguito. Si potrà notare, però, che l'algoritmo verrà interpretato ad ogni esecuzione, rendendo non troppo performante l'esecuzione stessa. Un software realizzato attraverso un

linguaggio di programmazione compilativo, invece, dipenderà dal singolo sistema operativo. Alla fine della traduzione, infatti, il compilatore genera un file eseguibile solo sul tipo di sistema operativo nel quale è installato. Risulta ovvio che un software così prodotto non garantisce portabilità, ma risulta efficace durante l'esecuzione, perché il file eseguibile prodotto dal compilatore è già in linguaggio macchina, quindi non da interpretare ma solo da eseguire.

Esiste, inoltre, un terzo modello, che prevede una precompilazione attraverso cui il codice viene tradotto in un linguaggio intermedio, detto **byte-code**, che nasce per essere eseguito non dal computer reale, ma da una macchina virtuale installata nel sistema operativo. In questo modo il bytecode risulta sicuramente più veloce di un linguaggio puramente interpretato, garantendo così la portabilità ma ottimizzando l'esecuzione. Python è un linguaggio di questo tipo, garantendo così sia la portabilità che una discreta efficienza nell'esecuzione.

Installazione, Interfaccia ed ambiente di sviluppo

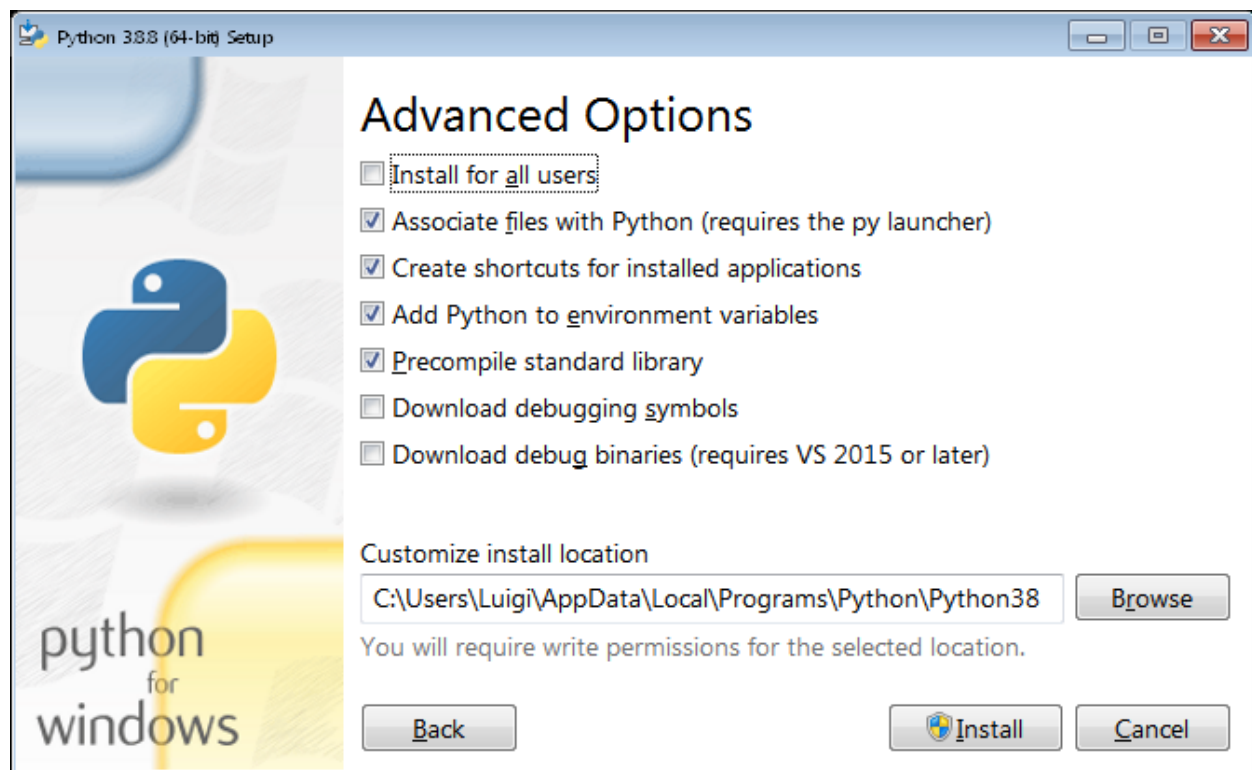
Dal sito di Python (<https://www.python.org/>) è possibile scaricare il pacchetto di installazione per ogni sistema operativo, sarà possibile scegliere tra due versioni ben definite, la versione 2 e la versione 3. Le differenze tra i due rami sono dal punto di vista relativo ad alcuni aspetti della sintassi, ma influiscono soprattutto nell'utilizzo di alcuni moduli che non supportano la nuova versione 3 del linguaggio. Tuttavia per un utilizzo generico e per la generazione di nuovo software è sempre consigliabile l'utilizzo della versione 3. Per entrambe le versioni, comunque, esistono diverse sotto versioni, tutte compatibili all'interno dello stesso ramo (ramo 2.x o ramo 3.x) ma talvolta compatibili solo in alcuni di queste versioni con moduli esterni o determinati sistemi operativi. Queste versioni, comunque, non sono incompatibili all'interno di un sistema, sarà infatti sempre possibile installarne più di una ed utilizzarle nel modo più opportuno.

Si faccia attenzione alle diverse versioni perché, alcune di queste non sono compatibili con tutti i sistemi operativi. Bisogna notare infatti che la versione 3.9 (e superiori) non è compatibile con Windows 7, ma solo con Windows 10. Questo problema può essere aggirato installando una versione precedente, andando a ritroso con le versioni si consiglia di installare la prima compatibile con il proprio sistema operativo.

Una volta lanciato l'eseguibile della versione scelta si consiglia di selezionare il checkbox relativo all'aggiunta di Python al PATH, che consente l'utilizzo dell'applicativo attraverso la linea di comando.

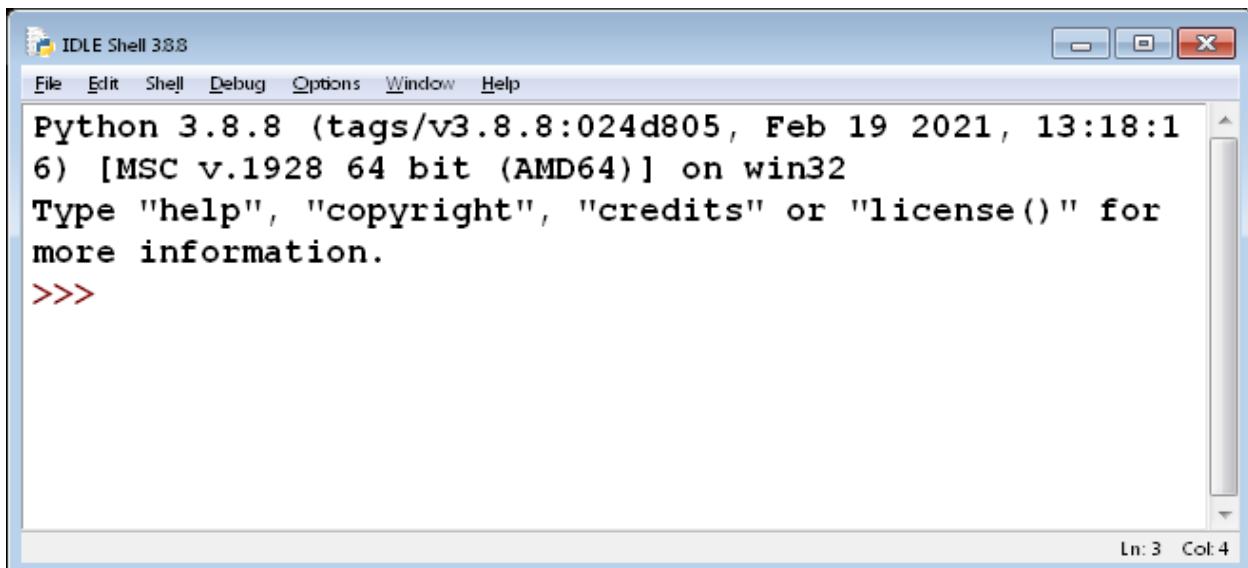
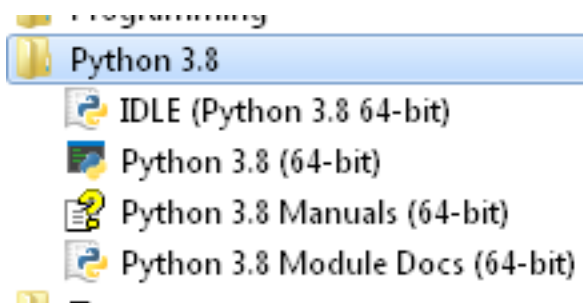


Altra opzione suggerita è l'installazione per tutti gli utenti della macchina, attraverso il checkbox “installa per tutti gli utenti”.



Arrivati a questo punto si può Procedere con l'installazione che verrà eseguita creando una cartella in **C:\Program Files\PythonX** (questo vale per Windows, ogni sistema operativo avrà il proprio percorso).

Una volta installato il pacchetto relativo al proprio sistema operativo, sarà possibile accedere alla prima interfaccia di Python: l'**IDLE**, un ambiente integrato che permette l'utilizzo e l'interprete per la scrittura di codice. È opportuno sottolineare che per la scrittura di più righe di codice, quindi di script di esercitazione o interi software, risulta molto più comodo utilizzare ambienti di sviluppo diversi che includono editor di testo oltre ad agganciarsi all'interprete per l'esecuzione dei programmi realizzati. Si consiglia quindi di valutare l'installazione di ambienti come <https://www.jetbrains.com/pycharm/> o <https://code.visualstudio.com/> che si possono anche interfacciare con **git** e **Github**. Si noti che questi ambienti non sono interpreti Python, ma si agganciano alle versioni di Python installate sul PC senza le quali non sarà mai possibile eseguire codice.



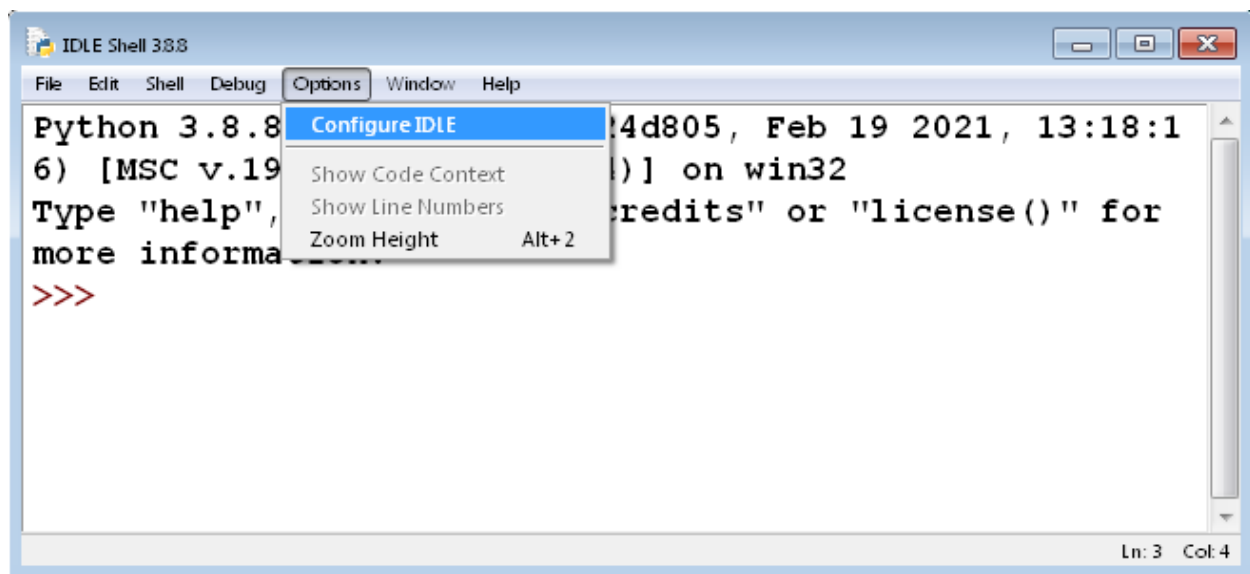
Un programma potrebbe essere scritto con qualunque editor di testo, anche WordPad o blocco note, risulta però sconsigliato perché tali programmi non riconosceranno la sintassi di Python.

e non permetterebbero di riconoscere errori di sintassi in modo semplificato. Gli ambienti di sviluppo sopracitati, invece, permettono non solo di riconoscere il codice, ma di suggerire elementi della sintassi, anche attraverso il completamento automatico delle parole.

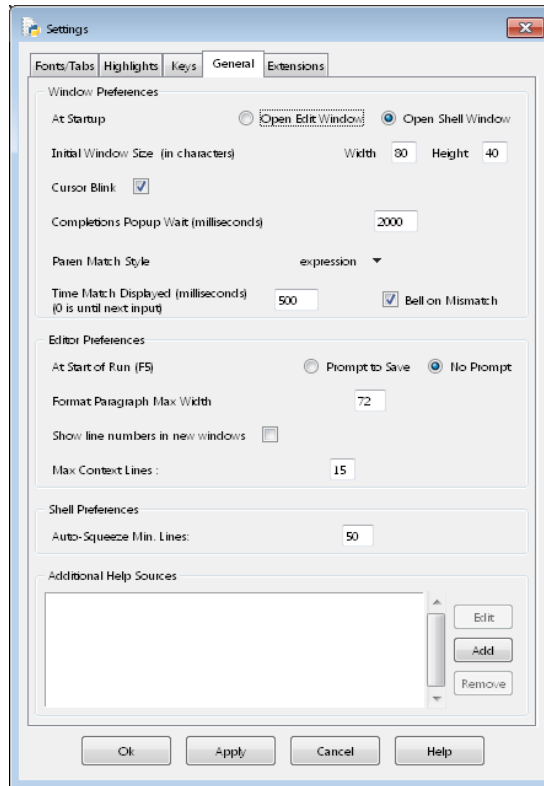
In sintesi per poter eseguire codice Python scritto all'interno di file bisogna:

1. Scaricare una versione di python compatibile con il proprio sistema operativo.
2. Installare un ambiente di sviluppo utile alla scrittura di codice.
3. Scrivere un programma e salvarlo in un file con estensione .py
4. Richiamare l'interprete python o attraverso l'ambiente di sviluppo, l'IDLE o attraverso il prompt dei comandi (o shell per sistemi linux/macosx).

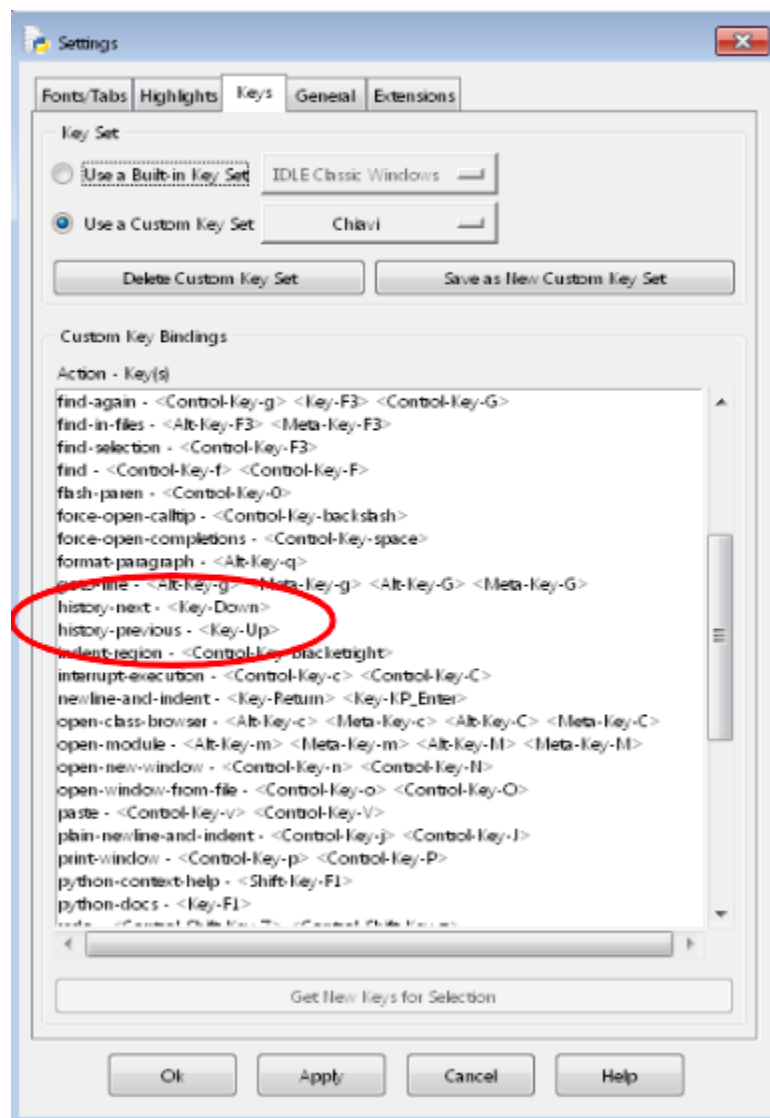
Personalizzare L'IDLE



Le tre parentesi angolari rappresentano il prompt dei comandi, per adeguare l'interfaccia alle proprie esigenze. Si deve scegliere la voce **configura IDLE** nel menu **Options**. In particolare si consiglia di fare in modo di non richiedere sempre la conferma del salvataggio da effettuare ogni volta che un programma viene eseguito. Per fare ciò bisogna selezionare la voce **No Prompt**, nella scheda **General**.

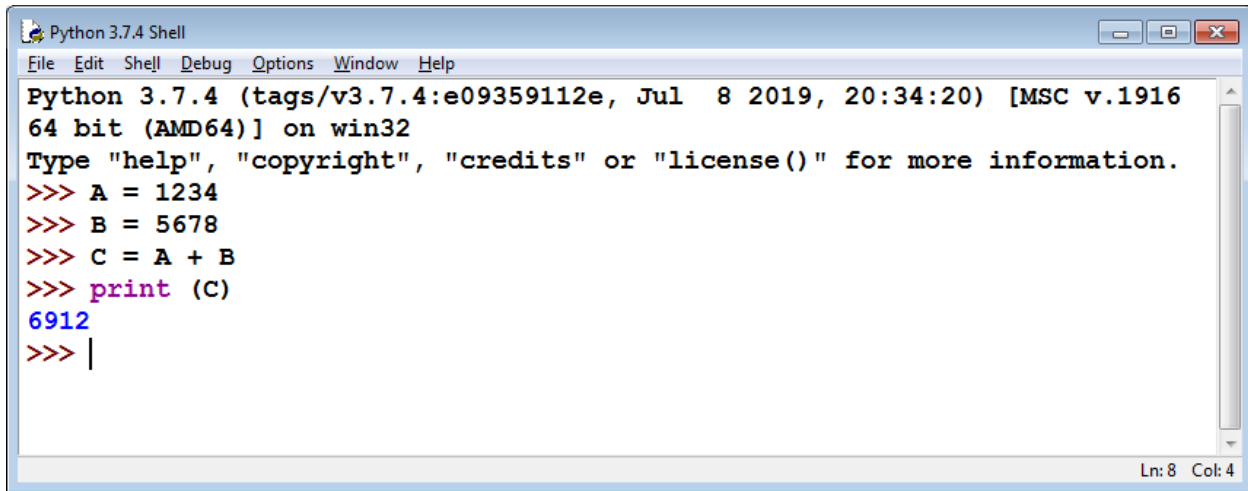


Nella finestra **Keys**, invece, è possibile indicare le chiavi attraverso cui gestire operazioni frequenti con l'uso di un singolo tasto. Generalmente nel *prompt dei comandi* risulta molto comodo navigare attraverso la cronologia delle istruzioni eseguite con l'uso delle frecce: freccia in su per tornare indietro nella cronologia delle istruzioni eseguite e freccia in giù per avvicinarsi all'ultima istruzione lanciata. Per impostare questi utili *shortcut* bisogna associare **Up Arrow** e **Down Arrow** alla **history-previous** e **history-next** nella scheda **Keys**. Dopo aver dato l'OK, salvare con un nuovo nome in **New Custom Key Set**.



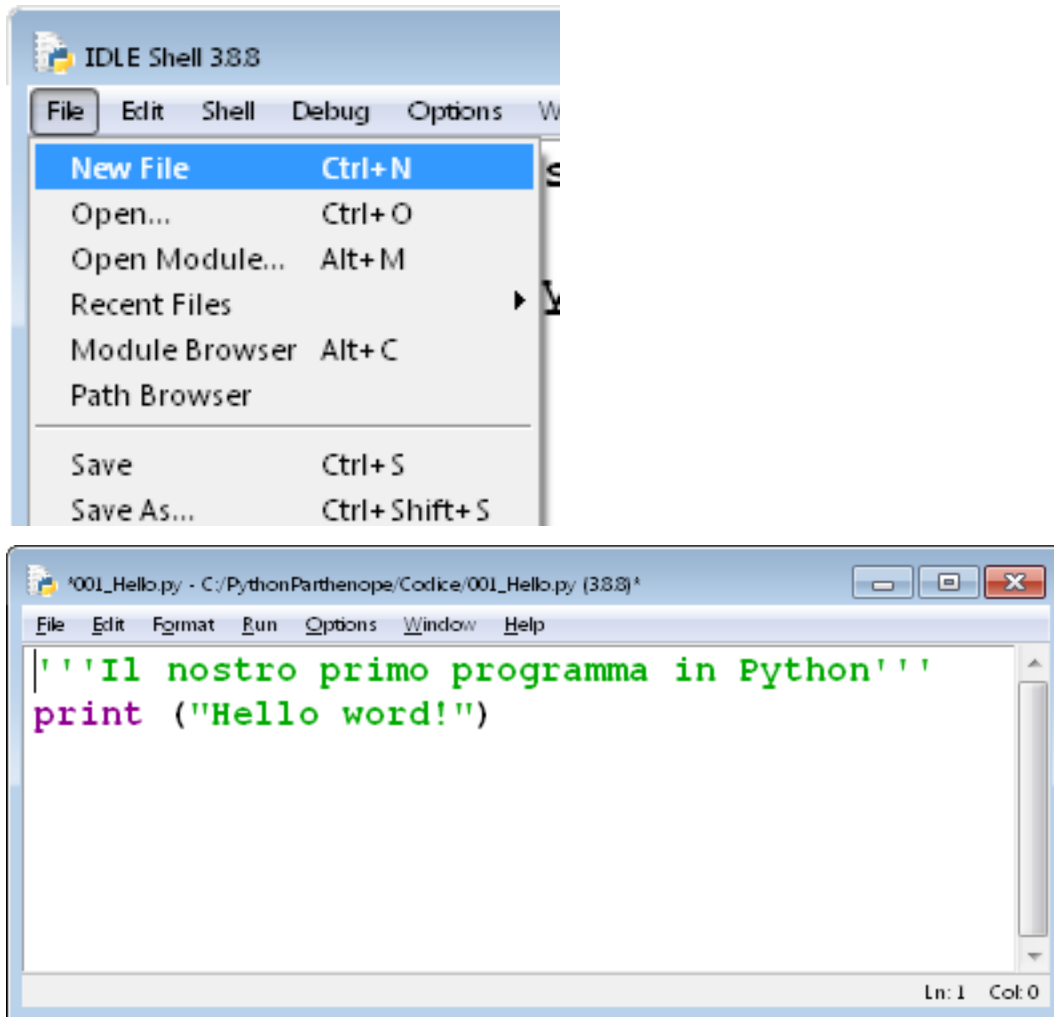
Eseguire istruzioni

A questo punto è possibile iniziare a programmare attraverso l'IDLE, scrivendo un'istruzione per volta.

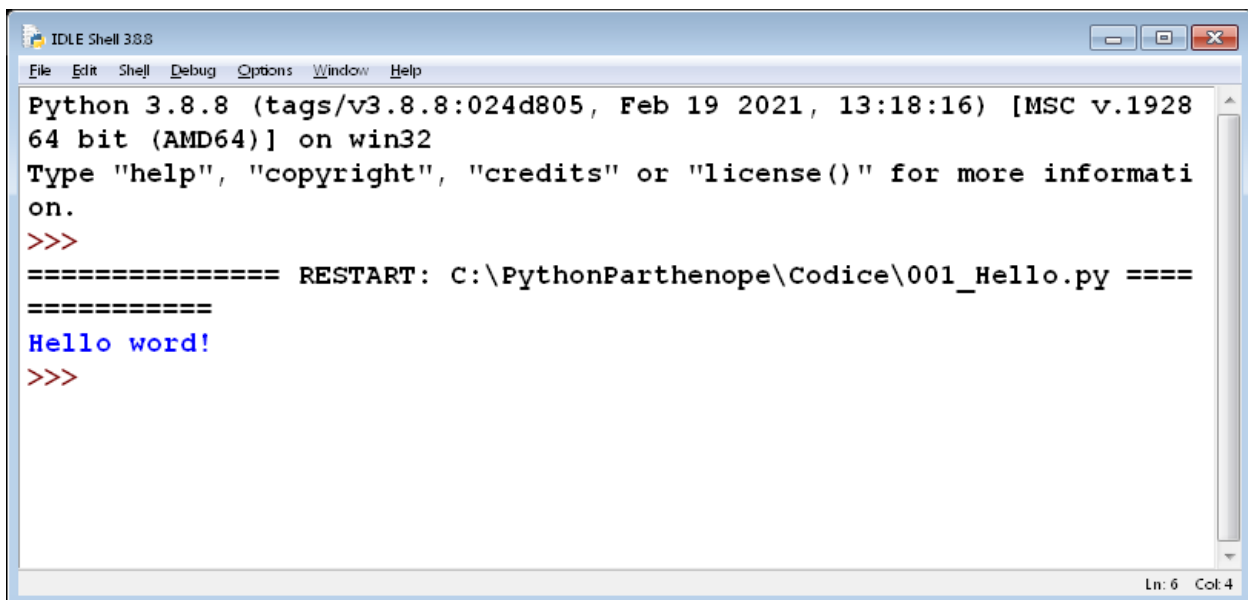
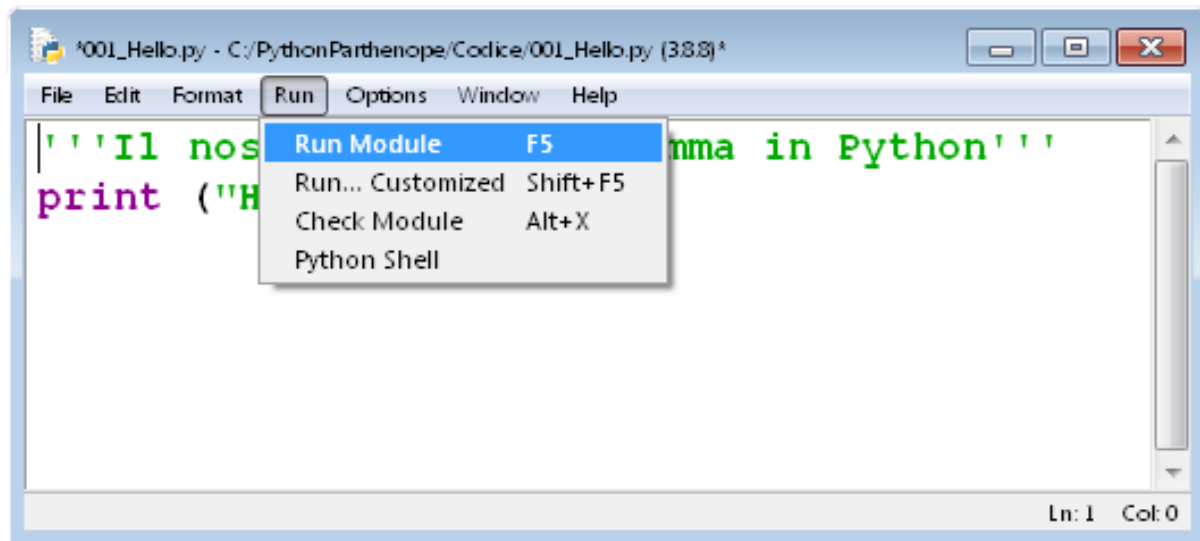


```
Python 3.7.4 Shell
File Edit Shell Debug Options Window Help
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul  8 2019, 20:34:20) [MSC v.1916
64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> A = 1234
>>> B = 5678
>>> C = A + B
>>> print (C)
6912
>>> |
```

Molto spesso invece risulta utile scrivere uno script fatto di più istruzioni in sequenza, un programma in genere è formato anche da più file, collegati tra loro in modo da rendere il codice più chiaro ed organizzato al meglio. PER fare questo è possibile creare un nuovo file attraverso il relativo menu, come indicato in figura. Il file può essere generato attraverso qualunque strumento, anche un editor di testo semplice come il blocco note, ma per essere eseguito deve avere estensione **“.py”**.



Per eseguire lo script bisognerà lanciarlo attraverso il menu **Run - > Run Module**.



Come è stato accennato nei paragrafi precedenti, è possibile usare anche altri editor, ad esempio Notepad++, blocco note o qualunque strumento per la scrittura di testo senza formattazione (si sconsiglia vivamente Word, Writer ecc).

Variabili, strutture dati e istruzioni

Tutti i linguaggi programmazione prevedono una sintassi ben definita che permette una comunicazione chiara e non ambigua al software che deve interpretare il codice. Nel caso di Python, le regole sintattiche sono particolarmente semplici e permettono una scrittura di codice facilmente comprensibile anche da un utente che non abbia particolare dimestichezza con la programmazione. In questo testo sarà data per assodata la conoscenza minima della codifica di un algoritmo, relativo all'uso delle variabili ed alla loro tipologia, oltre ai concetti base relativi alle strutture di controllo come selezione e strutture iterative. Per conoscenza minima si intende il concetto di variabile come contenitore all'interno del quale va inserito un valore, dove il valore può essere di varia natura: una sequenza di caratteri, un numero intero, un numero reale, un booleano e così via. A differenza di molti linguaggi programmazione bisogna sottolineare che in Python è parte della sintassi l'indentazione del codice, questo vuol dire che gli spazi ad inizio rigo permettono all'interprete di comprendere la divisione in blocchi del codice. Non rispettare l'indentazione corretta restituisce un errore di compilazione che non permette l'esecuzione di tutte le righe di codice successive all'errore commesso. Negli esempi che seguono si darà per scontato l'indentazione del codice come concetto acquisito.

Sintassi di base

Python è un linguaggio case-sensitive, questo vuol dire che scrivere il nome di una variabile, una "Casa" quindi è diversa da "casa".

È possibile commentare i programmi in modo da rendere più comprensibile a distanza di tempo o nella condivisione con altri sviluppatori. L'interprete salterà tutto ciò che è indicato come "commento" ed è possibile farlo in due modi:

- I commenti su una sola linea iniziano con `#` *questo è un commento*
- I commenti su più linee sono delimitati con il triplo apice `''' commento su più linee '''` o con il triplo doppio apice `""" commento su più linee """`
- Python racchiude le frasi sia con l'apice singolo che con il doppio apice.

Python 3, come tutti i linguaggi di programmazione, ha parole chiave che non possono essere utilizzate per definire variabili, funzioni e altro. Le parole chiavi sono le seguenti:

and, as, assert, break, class, continue, def, del, elif, else, except, False, finally, for, from, global, if, import, in, is, lambda, None, nonlocal, not, or, pass, raise, return, True, try, while, with, yield.

Parte della sintassi in Python è l'indentazione che permette di raggruppare le istruzioni in blocchi. Questo approccio impone la scrittura di codice molto più leggibile, perchè costringe il programmatore a indentare in modo chiaro e lineare il codice. Ulteriore vantaggio è quello di alleggerire la scrittura dall'uso delle parentesi graffe e dei punti e virgola di fine istruzione, così come invece avviene in molti altri linguaggi di programmazione. Bisogna però rendersi conto che l'interprete potrà generare facilmente errore se non si tiene conto del numero di spazi ad inizio istruzione per garantire una corretta gestione dei blocchi, il rientro standard è composto da 4 spazi. Un blocco di codice è una sequenza di istruzioni raggruppate insieme in base all'allineamento che vengono trattate come se fossero un'unica istruzione. Ovviamente se si considerano strutture di complesse di codice è possibile realizzare blocchi indentati in modo **nidificato**, così da realizzare blocchi di istruzioni dentro altri blocchi.

Per interrompere in modo forzato un programma si utilizza la combinazione di tasti ctrl+C oppure chiudere semplicemente l'IDLE.

Variabili

In Python, per dichiarare una variabile, non è necessario indicare il tipo di valore che andrà conservato. Il tipo di variabile è un concetto molto importante perché a partire da questo sarà possibile manipolare i dati contenuti nelle variabili in modo opportuno. Ad esempio, se viene acquisito un dato numerico come tipo "carattere alfanumerico" non sarà manipolabile attraverso le quattro operazioni permesse ad un numero acquisito come valore numerico di tipo intero o reale. Nell'esempio che segue vengono inizializzate alcune variabili alle quali vengono assegnate valori di tipo diverso, per poi restituirle in output attraverso la console.

```
tipi.py

#python3
#python3
#tipi di variabili
'''
e commenti molto lunghi
'''

numero = 3
stringa = "casa con virgolette"
stringa2 = 'casa con apici'
```



```
booleano = False  
reale = 3.14 #float  
  
print(numero,type(numero))  
  
print(stringa,type(stringa))  
print(stringa2,type(stringa2))  
print(booleano,type(booleano))  
print(reale,type(reale))  
  
print("multiplico il ",numero,"per il ",reale,"ottengo ",numero*reale )
```

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>OUTPUT<<<<<<<<<<<<<<<<<<<<<<<<<<

```
3 <class 'int'>  
casa con virgolette <class 'str'>  
casa con apici <class 'str'>  
False <class 'bool'>  
3.14 <class 'float'>  
multiplico il   3 per il   3.14 ottengo   9.42
```

Si noti che:

- La prima parte del codice vede due tipologie di commenti, dove per commenti intendiamo testo destinato ad un lettore umano di questo codice non interpretabile da Python. Le tipologie di commenti sono due: con il carattere cancelletto “#” è possibile commentare una singola riga, invece un testo più lungo può essere incluso all'interno di tre apici consecutivi (*''' testo molto lungo '''*).
- Il simbolo di uguaglianza permette l'assegnazione del valore alla variabile, questa assumerà il tipo di valore indicato perché l'interprete dedurrà la tipologia a partire dal dato.
- Le variabili **stringa** e **stringa2** contengono valori di tipo stringa, cioè sequenze di caratteri alfanumerici. Possono essere dichiarati attraverso l'uso degli apici o delle virgolette, la differenza fra le due dichiarazioni verrà evidenziata in seguito.
- La funzione **print()** restituisce in output, nella console, ciò che viene incluso tra le parentesi tonde. Come in tutti i linguaggi di programmazione le funzioni acquisiscono in questo modo i parametri di input, separati dalle virgole. Nel caso in esempio la funzione

`print()` acquisisce come primo parametro la variabile da restituire in output e come secondo parametro il risultato della funzione **`type()`**, attraverso cui Python restituisce il tipo di variabile passato come input a tale funzione. Risulta evidente che per ogni variabile il tipo restituito cambia in funzione del dato assegnatogli.

- Si noti che nella console, l'output viene stampato su righe diverse se la funzione *print* viene ripetuta. Se invece le variabili sono stampate attraverso una sola istruzione *print*, ma tutti gli elementi sono separati dalla “virgola” finiscono sulla stessa riga.
- Nell'ultima chiamata alla funzione *print*, oltre a una serie di elementi separati da virgole, è presente una moltiplicazione tra due variabili, la funzione *print* prima esegue la moltiplicazione del contenuto delle due variabili e poi restituisce in output il valore finale.

Quando si deve individuare il nome di una variabile è bene rispettare alcune regole basilari. È consigliabile infatti che il nome della variabile ricordi il significato del dato che dovrà contenere così da rendere il codice più leggibile. È possibile rispettare alcune convenzioni come ad esempio utilizzare il carattere “_” per separare due parole nel caso ce ne fosse bisogno, oppure di attaccare le due parole scrivendo con lettera maiuscola l'iniziale della seconda parola (ad esempio *secondaVariabile*). Non sarà invece possibile inserire spazi all'interno di un nome.

Casting

Il casting è una operazione che permette di cambiare il tipo di dato conservato all'interno di una variabile. Questo tipo di operazione è lecito per le variabili semplici presentate nel paragrafo precedente ma anche in quelle strutturate che verranno trattate successivamente. Avanzando nello studio del linguaggio e delle varie tipologie di contenitori emergeranno le diverse modalità di Casting.

Un'operazione di casting risulta necessaria, ad esempio, se bisogna convertire il valore dato in input da stringa a intero, per una successiva elaborazione.

```
elementi_base/casting.py
```

```
#python3
#casting

stringa = "5"

print ("stringa è di tipo ", type(stringa) )
```

```

intero = int(stringa)

print("Il tipo del valore contenuto in STRINGA è diventato INTERO: ",
      type(intero))

reale = float(stringa)

print("Il tipo del valore contenuto in STRINGA è diventato REALE: ",
      type(reale))

stringa2 = str(float)

print("e da REALE può tornare STRINGA: ", type(stringa2))

```

Operatori

Python mette a disposizione diversi operatori che permettono la manipolazione di valori e l'utilizzo delle variabili.

Assegnazione	=
Operatori Aritmetici	+ - * / % += -= *= /= //= // <i>(divisione intera)</i> ** <i>(potenza)</i> ^ <i>(exor)</i>
Operatori Relazionali	< <= > >= == !=
Operatori Logici	and or not

Per quanto riguarda l'istruzione di assegnazione è possibile utilizzare una versione sintetica per quanto riguarda le assegnazioni a più variabili, che prende la seguente forma:

```
# assegna ad A, B, C rispettivamente i valori 12, 34 3 57
A, B, C = 12, 34, 57

# è possibile anche assegnare valori da altre variabili
A, B, C = d, e, f
```

Controllo di flusso

Prerequisito di questo documento è avere conoscenza degli elementi basilari di un algoritmo, con questo significa avere cognizione delle basilari costrutti di controllo. Tuttavia si ritiene utile ricapitolare i concetti base e descrivere la relativa sintassi in Python 3.x.

Selezione

Il primo costrutto è la selezione di un blocco di codice in funzione della verifica di una condizione. **Se** (una condizione è verificata) **Fai** (blocco di codice) **Altrimenti Fai** (un altro blocco).

```
'''
if condizione:
    blocco1
elif condizione:
    blocco2
else:
    blocco3

oppure con l'operatore ternario:

variabile = expr1 if condizione
            else expr2

esempio:
'''
Maggiore = A if A>B else B
```

Si noti come esista la forma contratta **elif** in luogo di **Else If** e l'inesistenza di un costrutto Switch, presente in altri linguaggi come il C.

Iterazioni

Un'iterazione permette l'esecuzione di un blocco di codice molte volte, in funzione di una condizione. Esistono due tipi di iterazione, relativi a due condizioni differenti:

- **Ciclo di for.** Utile quando si ha la necessità di ripetere il blocco di istruzioni un numero determinato di volte.
- **Ciclo di While.** Necessario quando bisogna condizionare la ripetizione del blocco alla verifica di un evento o del valore di una variabile, senza conoscere in anticipo il numero di ripetizioni che si realizzeranno.

```
'''  
CICLO WHILE  
  
while condizione:  
    blocco  
  
CICLO FOR  
  
for indice in range(Nvolte):  
    blocco  
'''
```

Si tenga presente che:

- l'istruzione **for** in Python itera su tutti gli elementi di una sequenza, nel paragrafo successivo si affronteranno tali tipi di elementi.
- Il ciclo **while** ripete il blocco di istruzioni finché la condizione interna al controllo risulta vera.

Range

L'istruzione range in Python genera una lista di numeri in progressione aritmetica. Permette di ricevere diversi tipi di input, in funzione dei quali genera un diverso output. L'estremo sinistro è sempre incluso, quello destro sempre escluso. Si provi ad eseguire il seguente codice:

```
elementi_base/range.py
```

```
'''
RANGE
'''

# UN PARAMETRO: sottintende iniziare da 0
sequenza = range(10)
print("\n\n prima sequenza: ")
for i in sequenza:
    print(i, end=' ')

sequenza2 = range(5,10)

# DUE PARAMETRI: inizio e fine
print("\n\n seconda sequenza: ")
for j in sequenza2:
    print(j, end=' ')

# TRE PARAMETRI: inizio, fine e step
print("\n\n terza sequenza: ")
sequenza3 = range(5,20,2)
for k in sequenza3:
    print(k, end=' ')

print("\n")
```

Si noti che sono stati usati nomi diversi per le variabili iteratrici (i, j, k) perchè altrimenti l'indice del ciclo successivo sarebbe stato sequenziale rispetto al ciclo precedente.

Input ed output

Quando si costruisce un algoritmo spesso ci si trova nella necessità di comunicare con l'esecutore, sia nel fornire input che nel ricevere informazioni durante l'elaborazione, ad esempio per avere il risultato finale dell'algoritmo realizzato.

Print

Per restituire in output uno o più valori si utilizza la funzione **print()**. Attraverso questa istruzione è possibile restituire a monitor i valori passati per input, tenendo presente che anche questa funzione riconosce il tipo di input e quindi si regola di conseguenza, modificando il relativo output. Si tenga presente il seguente esempio:

```
elementi_base/print.py
```

```
'''
PRINT
'''
a = 124523

print(a, file=sys.stdout)

# posso descrivere il valore da stampare, separando con la virgola

print("a= ",a)

parola1 = "casa"
parola2 = "albero"
parola3 = "montagna"

print(parola1, parola2, parola3, sep="--")

a = range(1,10)
```

```
for i in a:  
    print(i, end='')  
  
for j in a:  
    print(j)
```

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>OUTPUT<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

```
a= 124523  
casa--albero--montagna  
1234567891  
2  
3  
4  
5  
6  
7  
8  
9
```

Si noti che:

- Per inserire una stringa e visualizzarla in output va scritta tra apici e/o virgolette. Se la stringa include una virgoletta va inserita tra apici, se la stringa contiene un apostrofo si deve usare la doppia virgoletta. Tutto ciò che non è tra virgolette o apici viene considerata una variabile, sostituendo in outuput il suo valore.
- La funzione *print* può ricevere oltre che parametri e stringhe anche la modifica di alcuni attributi. I parametri vanno indicati secondo la sintassi *parolachiave* = 'valore' ed il valore deve essere una stringa. Se si indica come valore **"None"** il parametro assumerà il valore di default. I parametri sono:

sep: indica il carattere separatore degli input separare i valori passati alla funzione.

end: indichiamo alla fine di tutto l'output cosa deve stampare. di Default il valore è “\n” che restituisce “*new line*”, quindi la prossima istruzione di print scriverà al rigo successivo.

file: indica dove deve essere direzionato il flusso in output, “=sys.stdout” è il terminale.

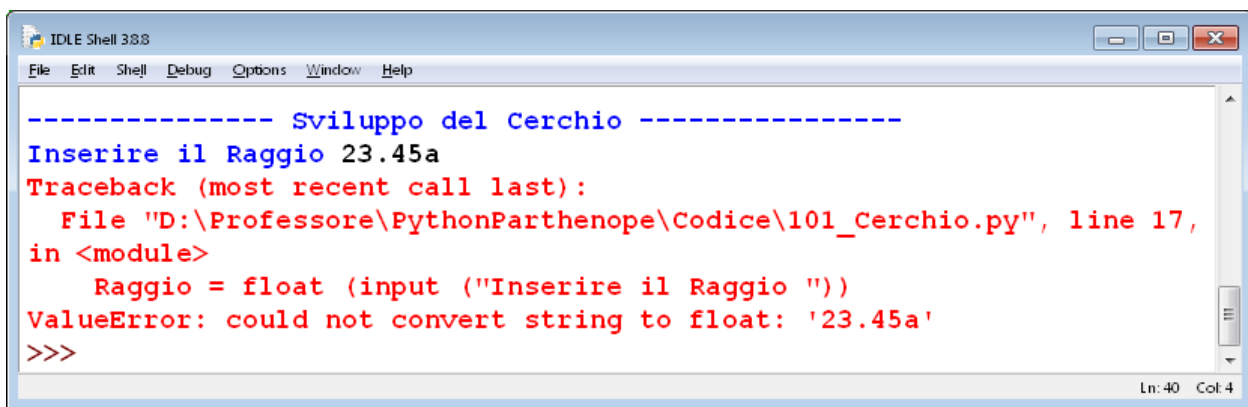
Input e gestione degli errori

Per acquisire dati dall'utente bisogna utilizzare la funzione **input()**, con la consapevolezza però che tutti i dati ricevuti sono stringhe e vanno convertiti in funzione del loro utilizzo. Nell'esempio

che segue è acquisito un valore dall'utente che viene prima convertito in intero e successivamente inserito nella variabile num.

```
num = int(input("inserisci un numero intero:"))
```

Può capitare però che un dato richiesto serva per un calcolo numerico, ma l'utente invece inserisca un carattere non numerico, in questo caso il sistema restituisce errore.



In questi casi l'interprete si blocca, restituisce un errore all'utente che spesso non è molto comprensibile, in python è possibile gestire questi errori che vengono chiamate **eccezioni**.

Le eccezioni sono errori che avvengono soltanto durante l'esecuzione del programma, che dal punto di vista tecnico si chiama fase di "runtime". Tali tipi di errori possono avvenire in più occasioni, ad esempio:

- Eseguendo una divisione per zero si avrà: **except ZeroDivisionError**
- Tentare di convertire in un numero una stringa non convertibile si otterrà: **except ValueError**

Per gestire questi imprevisti Python mette a disposizione un sistema di controllo detto try-except, il cui schema di funzionamento è relativamente semplice. il comando **try** incorpora il blocco di codice che può generare errori, se si solleva un'eccezione l'interprete scorre le varie tipologie di errore elencate nel blocco except. È infatti possibile gestire determinate eccezioni o gestirle in modo generico. Le istruzioni successive al blocco try, quando viene sollevata una eccezione, non vengono prese in considerazione.

Si consideri il seguente esempio:


elementi_base/input-try-except.py

```
'''
TRY-EXCEPT

Struttura
try:
    ... # Blocco di codice che può generare errori
except ValueError:
    print('Valore inserito non valido!')
except ZeroDivisionError:
    print('Divisione per zero')
except:
    print('Altro tipo di errore')
else: # dopo tutte le clausole except
    print('Nessun errore') # se non ci sono errori
finally:
    print('Fine') # Eseguita anche in caso di errore
'''

Raggio = -1.0
while Raggio < 0:
    try:
        Raggio = float(input("Inserire il Raggio "))
    except:
        print ("Inserimento non valido !!!")
        Raggio = -1.0
    else:
        if Raggio < 0:
            print ("Raggio NON puo` essere negativo!")

Area = Raggio * Raggio * 3.14159265
print('Area', Area, sep=' = ')
```

Variabili strutturate

In molte occasioni non è sufficiente conservare un solo valore all'interno di una variabile, ma risulta necessario associare più valori allo stesso contenitore. Questo tipo di strutture non sono semplici, perchè necessitano di associare più dati allo stesso nome. Tutti i linguaggi di programmazione moderni, per risolvere questo inconveniente mettono a disposizione del programmatore diverse strutture dati, più complesse delle variabili semplici presenti negli esempi precedenti. Tali strutture si differenziano tra loro per le caratteristiche relative alla tipologia di dati, alla loro rappresentazione ed all'utilizzo che se ne deve fare. Per tale motivo Python mette a disposizione molte strutture diverse, così da venire in contro alle diverse necessità.

Le principali strutture dati sono quelle indicate di seguito e che verranno trattate nei paragrafi successivi:

- Stringhe
- Liste (list)
- Insiemi (set)
- Dizionari (dictionary)
- Tuple

Stringhe

Una stringa è una sequenza ordinata di caratteri. Spesso questo tipo di dato viene associato alle variabili di tipo semplice, ma in realtà è un tipo di dato strutturato che include diversi tipi di comportamenti facilmente accessibili. Nei paragrafi che seguiranno verranno affrontati i concetti di "comportamento" di variabili complesse. In questo contesto ci si limita a sottolineare che esistono comportamenti comuni a tutte le stringhe.

Nell'esempio che segue vengono inizializzate alcune variabili come stringhe, associando ad ogni variabile una sequenza di caratteri. si noti come sia possibile associare ad una variabile una stringa attraverso le virgolette (" ") oppure gli apici (' '). La differenza consiste nella possibilità di poter inserire nel testo da salvare nella variabili rispettivamente apici o virgolette.

```
estratto da: elementi_base/stringhe.py
```

```
'''  
OPERAZIONI SULLE STRINGHE
```

```
'''
nome = "paolo"
spazio = ' '
cognome = 'esposito'

# concatenazione
frase = nome + spazio + cognome

print(frase)
print()
# possiamo usare apici ' o virgolette ""

# se abbiamo bisogno di usare apici, includiamo
# la stringa tra virgolette e viceversa.

apice = "usiamo l'apostrofo"
virgolette = 'o "virgolettare" un testo'

print(apice)
print(virgolette)
print()
# accesso agli elementi di una stringa

print('la quarta lettera di virgolette è: ',virgolette[4] )
print()
# una stringa è immutabile, non possiamo cambiare
# un singolo carattere, ma l'intera stringa si

# virgolette[4] = 'c' -> restituirebbe errore

virgolette = "cambio la stringa" #posso farlo

print(virgolette)
print()

numeroCarattere = '4'
```

```
numeroIntero = int(numeroCarattere)
numeroIntero += 1

print("il numero incrementato è ", numeroIntero)
print()
```

Nell'esempio è possibile evidenziare:

- L'operazione di concatenazione avviene utilizzando l'operatore " + ", che per le stringhe assume un valore diverso da quello di somma tra numeri.
- È possibile risalire ad una singola lettera attraverso la sua posizione, si noti che la numerazione delle posizioni iniziano da 0.
- Non è possibile sostituire un singolo carattere, ma sovrascrivere una intera stringa.
- Un numero in una stringa non può essere utilizzato in funzioni matematiche, perché risulta come carattere alfanumerico. Per poterlo manipolare come numero in senso stretto bisogna fare un'operazione di casting.
- Poiché la stringa è un tipo di dato immutabile non è consentito modificarla dopo la creazione, ma risulta necessario crearne una nuova che contiene quella da modificare.

Si osservino adesso alcuni comportamenti nell'esempio che segue.

estratto da: elementi_base/stringhe.py

```
# lunghezza di una stringa
n = len(virgolette)
print ("virgolette contiene", n, "caratteri")
print()

# Sezioni di stringa, posso estrarre una parte della stringa
print("estrapoliamo porzioni dalla variabile 'virgolette'")
s = virgolette[3:6]
print(s)
print(virgolette[5:8])
```

```
print('primi 4 caratteri: ',virgolette[:4])
print('dal quarto alla fine: ',virgolette[4:])
print()

#appartenenza ad una stringa verifica
#l'esistenza di uno o più caratteri e restituisce
# TRUE o FALSE

print('frase: ', frase)
print('verifichiamo se "a" è presente: ', 'a' in frase)
print('verifichiamo se "paolo" è presente: ', 'paolo' in frase)
print('verifichiamo se "paola" è presente: ', 'paola' in frase)
print()

#confronto tra stringhe
# == uguaglianze
# < e > precedenza alfabetica

nome1 = 'mario'
nome2 = 'MARIO'

print("verifichiamo se 'nome1' è uguale a 'nome2'", nome1==nome2)
print("verifichiamo se 'nome1' è > di 'nome2'", nome1>nome2)
print()

# ESEMPI CON CICLI
# la funzione len() restituisce il numero di caratteri
# la funzione ord() restituisce il codice numerico associato al
carattere

print (frase)
i=0

print("ciclo while")
while i < len(frase):
```

```
print(frase[i], " ", ord(frase[i]))
i = i + 1

print("ciclo for")
for lettera in frase:
    print(lettera, " ", ord(lettera))
```

Osservando l'esempio sopra descritto si noti che:

- Una delle funzioni più comunemente utilizzate è **len()**, che permette di ricavare la lunghezza di una stringa.
- È possibile ricavare gruppi di caratteri, attraverso l'uso delle parentesi quadre. Si notino tutte le possibili combinazioni, che permettono l'estrapolazione dei primi caratteri, degli ultimi o di quelli centrali, attraverso l'uso degli indici di carattere interni alla stringa.
- Attraverso l'operatore **in** è possibile verificare se un carattere è presente nella stringa data.
- È possibile la comparazione di caratteri e di stringhe, gli operatori verificano la coincidenza dei caratteri ed il loro ordinamento rispettando la codifica ASCII.
- La funzione **ord()** restituisce il codice relativo al carattere.
- Risulta possibile utilizzare i cicli per scorrere tutti i caratteri di una stringa.

Le funzioni relative alla manipolazione delle stringhe sono molteplici, si consiglia di fare riferimento alla documentazione ufficiale offerta dal sito di Python. Alcune funzioni frequenti, infatti, riguardano la manipolazione dei testi estratti da file. Nell'ultimo estratto dello script prima esposto sono presenti alcune di queste funzioni.

```
elementi_base/stringa.py

# dividere una stringa

fraseSpezzata = frase.split(" ")

# si otterrà una lista con i due pezzi della stringa
print(fraseSpezzata)
```

```
# è possibile eliminare una sottostringa
fraseStrippata = frase.lstrip('pa')

# si ottiene
print(fraseStrippata)
```

L'esempio introduce due delle tante funzioni applicabile ad una stringa. **Split()** permette di dividere la stringa separandola in funzione del carattere passato come input, nell'esempio è indicato il carattere "spazio", restituendo una lista contenente i due elementi della stringa iniziale. La funzione **lstrip()**, invece, estrae i caratteri indicati come input dalla testa della stringa e restituisce la restante parte come output. Si tenga presente che esiste Anche la versione **rstrip()** per estrarre la parte finale della stringa e **strip()** che estrae sia in testa che in coda i caratteri indicati In input.

Tuple

Una tupla è una **sequenza ordinata e immutabile** di valori non necessariamente dello stesso tipo. È possibile quindi conservare all'interno di una struttura di questo tipo elementi diversi tra loro, racchiudendoli tra parentesi tonde e separati tra virgole.

Si osservi il codice seguente:

```
elementi_base/tuple.py
```

```
'''
Una tupla è una sequenza ordinata e immutabile di valori non
necessariamente
dello stesso tipo e si racchiudono tra parentesi tonde
'''

tupla = (7, 4, 16, 34)
parole = ( 'foglia', 'casa', 'pompieri', 'albero' )
numero = 100
tupla_mista = (numero, 'foglia', 'casa', 'pompieri', 'albero' )
```



```
print ('tupla: ', tupla, '\n') #\n aggiunge una riga sotto alla parola
mandata in output

print ('parole: ',parole, '\n')

print ('tupla_mista: ',tupla_mista, '\n')

#un elemento di una tupla può essere a sua volta una tupla
elementi = (1, 'ancoraparole', 4,'enumeri', ('tuple', 'di', 'tuple' ))

print ('elementi: ', elementi, '\n')

#è possibile concatenare le tuple
print('concateno due tuple: ',tupla + parole, '\n')

#è possibile accedere agli elementi di una tupla tramite le parentesi
quadre []
print('parole[0]:',parole[0], '\n')

#il tipo di risultato dipende dal tipo di dato inserito nella tupla.
#la funzione type() restituisce il tipo di valore che bisogna analizzare
print('parole[0] è',parole[0],'ed è di tipo:')
print(type(parole[1]), '\n')
print('invece elementi[0] (' , elementi[0],') è di tipo')
print (type(elementi[0]), '\n')

#lunghezza di una tupla
print('la lunghezza della tupla PAROLE è:', len(parole), '\n')

# è possibile estrarre solo una sottosezione della tupla con
NOME[inizio:fine]
print ('elementi[1:4]:',elementi[1:4],'\n')

# è possibile verificare l'appartenenza di una tupla:
print('verifichiamo se 7 è presente nella tupla PAROLE.')
print('la sintassi usa la parola chiave IN, quindi: ')
```

```
print('"7 in parole" restituisce', 7 in parole, '\n')
print("'foglia' in parole, invece restituisce", 'foglia' in parole, '\n'
)

# è possibile anche confrontare due tuple
amici1 = ('mario', 'paolo', 'francesco')
amici2 = ('MARIO', 'paolo', 'francesco')
print('amici1 < amici2: ', amici1 < amici2)
print('invece amici1 > amici2: ', amici1 > amici2)
print("perchè le lettere minuscole seguono quelle maiuscole
nell'ordinamento", '\n')

# è possibile inoltre stampare attraverso un ciclo tutta la tupla
for elemento in elementi:
    print(elemento, " è di tipo", type(elemento))
```

Osservando il codice dell'esempio si noti che:

- Per inserire una stringa nella tupla si utilizzano gli apici. Come è deducibile dall'esempio, senza gli apici l'interprete considera quella parola come una variabile (*numero* è la variabile usata in *tupla_mista*)
- Possono essere inseriti i numeri, verranno considerati dall'interprete per il valore che rappresentano.
- Una tupla ne può contenere un'altra, quindi è possibile costruire variabili strutturate come tuple di tuple, attraverso l'uso di parentesi tonde innestate.
- Attraverso l'operatore "+" è possibile concatenare più tuple.
- Si accede al singolo elemento della Tupla attraverso l'indice di posizione, il primo elemento ha indice 0.
- Attraverso la funzione **len()** è possibile ricavare il numero di elementi appartenenti ad una tupla.
- È possibile estrarre elementi da una tupla utilizzando gli intervalli di indici, esattamente come avviene per le stringhe.
- È possibile verificare l'appartenenza di un elemento ad una tupla, confrontarle e stamparle attraverso un ciclo for.

Liste

Le liste sono strutture molto simili alle tuple e tutte le operazioni applicabili su una tupla può essere applicata ad una lista. La differenza che intercorre tra una lista ed una tupla è che in fase di assegnazione gli elementi che la compongono sono racchiuse tra parentesi quadre, anziché tra parentesi tonde. La differenza sostanziale però è che le liste sono mutabili, cioè è possibile modificare il valore di un singolo elemento.

Possiamo definire una lista in questo modo:

Una lista è una sequenza ordinata e mutabile di valori non necessariamente dello stesso tipo.

Si osservi l'esempio seguente:

```
elementi_base/liste.py
```

```
# DIFFERENZA TRA LISTA E TUPLA
elencoUno = [4,7,2,3]
elencoDue = (4,7,2,3)

print("elencoUno è una ", type(elencoUno))
print("elencoDue è una ", type(elencoDue))

lista = elencoUno

print ("\n!elemento 1 di lista è ", lista[1] )

print("\nLa sua lunghezza è ", len(lista))

lista = lista + ['paolo', 'giovanni', 'mario']

print ("\n!a lista adesso è:")

# stampa tramite ciclo di for
for elemento in lista:
```

```
print(elemento)

# stampa di una lista intera
print(lista)
# oppure stamparne solo una porzione
print("\nalcune porzioni:")
print(lista[4:])
print(lista[:4])

print ("\nLa lista è diventata di", len(lista), "elementi")

'''
è possibile annidare liste, signifca che un suo elemento può
essere una lista. Poichè sono oggetti mutabili è possibile manipolarle
aggiungendo altri elementi (con le tuple non è possibile)
'''

lista[3] = ['pane', 'pasta', 3, 6,1]

print('\n','visualizziamo la lista annidata','\n',lista)

print('\nPosso eliminare elementi in due modi.\n')
lista[0:1] = []
print("lista[0:2], la lista diventa:",lista,'\n')

del lista[2]
print ("del lista[2], la lista diventa:", lista,'\n')

listarange = range(10)
listarange2 = range(2,10,2)

# stampa di liste generate con range

print("listanrange")
for elemento in listarange:
    print(elemento)
```

```
print("listarange2")
for elemento in listarange2:
    print(elemento)

# verifica della presenza di un elemento.

print(7 in listarange)
print(7 in listarange2, "\n")
```

- A differenza delle tuple è possibile aggiungere altri elementi.
- Liste di numeri consecutivi oppure a passo costante possono essere create attraverso la funzione *range()*.
- Un elemento di una *lista* può essere a sua volta una *lista*, generando liste annidate.
- È possibile eliminare un elemento di una lista sostituendolo con un carattere vuoto oppure con l'utilizzo della funzione **del**.
- È possibile verificare l'esistenza di un elemento all'interno di una lista attraverso l'operatore **in**, questo restituirà **true** se trova l'elemento e **false** altrimenti.

Dictionary

Le variabili strutturate possono contenere più valori, come visto per le tuple e per le stringhe. Gli elementi contenuti in questo tipo di variabile però sono indicizzate solo per posizione numerica, è possibile quindi ricavare gli elementi salvati soltanto attraverso un indice numerico.

Si immagini invece di strutturare una rubrica telefonica. Risulterà necessario organizzare gli elementi per chiavi: nome, cognome, numero di telefono ecc. L'ordine degli elementi, inoltre, sarà in ordine alfabetico rispetto ad una di queste chiavi, per nome o cognome. Per tali motivi viene introdotto un altro tipo di struttura dati definito in questo modo:

I dizionari (dictionary) sono **collezioni mutabili e non ordinate di coppie chiave-valore**. Le chiavi devono essere univoche e non mutabili.

Si osservi il seguente esempio:

```
elementi_base/dictionary.py
```

```
# i dizionari sono collezioni mutabili e non ordinate di coppie
chiave-valore
# le chiavi devono essere univoche e non mutabili, quindi non si possono
utilizzare le liste,
# ma stringhe numeri o tuple contenenti oggetti immutabili

# ESEMPI
#
#          CHIAVE          DATO
# Rubrica telefonica      nominativo      Numero di telefono
# elenco insegnanti      materia          nominativo
# cambio valuta           moneta straniera  valore rispetto all'euro

professori = {"matematica": "tripepi", "coding": "mazzone",
"disegno": "cori"}
valuta = { "USD": 1.31212, "GBP": 0.78244, "CAD": 1.45642}

print ("professori: ", professori)
print ("valuta: ", valuta, '\n' )

# accesso agli elementi
prof = professori["matematica"]
print ("accedo all'oggetto relativo alla chiave matematica: ", prof, '\n')

# per aggiungere un elemento bisogna far riferimento ad una nuova chiave

professori["ed. fisica"] = "crisafo"

print ("professori aggiornato: ", professori, '\n')

# anche nei dizionari è possibile usare la funzione len

print ('il numero di elementi è len(professori): ', len(professori), '\n')

# cancellazione di un elemento
```

```
del professori['ed. fisica']

print ("professori aggiornato: ", professori, '\n')

# riferimento ad un dizionario, non si può effettuare la copia

professori2 = professori

professori2['ed. fisica'] = "nuovo prof"

#stampando professori si visualizza la modifica avvenuta in professori2
print("riferimento aggiornato", professori, "\n")

# matrici

mat = {(0,0): 'A', (0,1):'B', (0,2):'C', (0,3):'D', (1,0):'E',
(1,1):'F', (1,2):'G', (1,3):'H'}

print ("elemento della matrice mat[1,3] è:",mat[1,3], '\n')
```

dall'esempio è possibile notare che:

- Ogni elemento è identificato da un'etichetta, ed attraverso questa è possibile modificare il valore del dato associato.
- Si noti che copiando *professori* in *professori2* non si è ottenuta una variabile copia, cioè che occupa un altro spazio di memoria con gli stessi elementi, ma si è ottenuta una variabile che fa riferimento alla stessa area di memoria. In tal modo si ottiene l'effetto che modificando un valore in *professori2*, la modifica sarà effettiva anche per la variabile *professori*.
- Si noti come è possibile creare una matrice i quelle menti sono le chiavi attraverso cui ricavarsi i dati.

Funzioni

Quando un algoritmo diventa particolarmente complesso spesso si ricorre all'utilizzo di sotto algoritmi che permettono l'esecuzione di sezioni autonome di codice. Tuttavia, racchiudere procedure autonome che concorrono all'esecuzione di un software, è una pratica consigliata e prende il nome di programmazione modulare. Un tale approccio permette la scomposizione di un algoritmo complesso in sotto-algoritmi, semplificando la scrittura del codice, il suo riutilizzo e contemporaneamente semplifica la leggibilità e la manutenzione dei singoli moduli. Senza entrare nello specifico delle differenze che intercorrono tra una procedura ed una funzione si potrebbe assumere questa definizione:

Una funzione è un algoritmo che per essere eseguito deve essere invocato all'interno di un programma. Può ricevere in input dei valori, esegue operazioni ed eventualmente restituisce in output un risultato, permettendo così al programma "chiamante" di proseguire l'esecuzione.

Negli esempi precedenti sono state utilizzate delle funzioni, dando per scontato il loro funzionamento. Per restituire all'utente un valore è stata utilizzata infatti la funzione `Print()`, attraverso la quale sono stati inviati valori verso la periferica di output senza preoccuparsi dell'algoritmo necessario affinché potesse avvenire tale flusso di comunicazione verso l'IDLE. lo stesso è avvenuto utilizzando la funzione `len()` per ottenere il numero degli elementi appartenenti ad una lista. È stato possibile utilizzarle perché Python mette a disposizione un numero considerevole di funzioni incorporate (dette anche built in), per un elenco completo si consulti il seguente link: <https://docs.python.org/3/library/functions.html>.

Quando verranno introdotti i moduli verranno esplicitate altre caratteristiche delle funzioni.

Programmazione ad oggetti (OOP)

Sono stati introdotti i concetti di variabili semplici e strutturate, trattandoli come contenitori che possono contenere uno o più valori. Questo tipo di approccio è molto semplice ed è molto chiaro se si programma in modo procedurale, significa programmare con una sequenza di istruzioni e funzioni che lavorano su contenitori capaci solo di conservare dati, sempre accessibili in tutte le parti del programma. Il programmatore, quindi, costruisce di volta in volta funzioni appositamente pensate per manipolare le varie tipologie di dati. In molti casi reali, quindi, tali funzioni risultano molto simili tra loro, si immagini ad esempio di dover realizzare tutte le operazioni che riguardano la gestione di una variabile strutturata destinata a contenere una graduatoria. Inserimento, cancellazione e ordinamento sono funzioni che dovrebbero essere implementate ogni volta che

si gestisce un contenitore di quel tipo. Con lo stesso approccio, ogni volta che si ha a che fare con le stringhe, bisognerebbe implementare le funzioni utili alla manipolazione: sostituisci carattere, cerca carattere, concatena due stringhe ecc.

Secondo quanto detto è possibile introdurre il concetto di **Classe**, come descrizione astratta di un tipo di dato, descrive quindi una famiglia di variabili (chiamate **Oggetti**) con caratteristiche e comportamenti comuni.

Riprendendo gli esempi introdotti in precedenza possiamo considerare la classe *String*, che descrive la struttura dati di tipo *Stringa* (Sequenza di caratteri alfanumerici e relativa posizione) e i comportamenti comuni a tutti gli oggetti appartenenti alla classe. I comportamenti vengono definiti **Metodi** e sono tutte quelle funzioni che si possono applicare ad una stringa: lunghezza, ricerca, aggiunta di un carattere ecc.

In sintesi sono stati introdotti i seguenti concetti:

- **Classe**, astrazione che descrive le caratteristiche comuni ad un insieme di strutture dati.
- **Oggetto**, un determinato elemento appartenente ad una classe. L'oggetto viene anche definito come **istanza** della classe, perché a differenza della classe che è una descrizione astratta, l'oggetto rappresenta la descrizione di un elemento in particolare.
- **Attributo**, uno degli elementi destinati a contenere dati relativi al singolo oggetto. Se si considera la classe RETTANGOLO, la Base e l'Altezza possono essere considerati attributi che al variare dei dati contenuti differenziano i singoli rettangoli, tutti istanze della classe RETTANGOLO.
- **Metodi**, sono i comportamenti applicabili ai singoli oggetti. Sono funzioni definite nella classe da applicare a tutti gli oggetti appartenenti a quella determinata classe. *area()* o *perimetro()*, possono essere metodi della classe RETTANGOLO. Metodi che utilizzeranno i dati conservati negli attributi *Base* ed *Altezza*.

Si potrebbe definire in questo modo: **una classe definisce proprietà e metodi, mentre un oggetto li valorizza e li usa.**

Volendo fornire altri esempi, potremmo definire la Classe Auto, come l'insieme delle autovetture a quattro ruote, con un serbatoio di benzina, un livello di pressione delle gomme, un colore, un determinato numero di portiere, marca e modello. Tutti quelli elencati possono essere considerati **Attributi**, cioè le caratteristiche della singola auto. Possono essere modificabili (livello benzina, pressione gomme) e non modificabili (marca, modello, numero di ruote ecc). Accendere il motore, accelerare o frenare, invece, sono definiti come **Metodi** e sono identificabili come i comportamenti dell'auto, non a caso si utilizzano verbi per definirli.

A questo punto è intuibile che Python permette l'uso di Classi ed oggetti, sia per strutture native del linguaggio che per la costruzione di classi personalizzate.

Caratteristiche della programmazione ad oggetti

L'uso delle classi e degli oggetti è parte di un paradigma di programmazione che prende il nome di **Programmazione ad Oggetti** (in inglese object-oriented programming, in acronimo OOP).

Attraverso questo paradigma di programmazione è possibile costruire software più complessi ed articolati, attraverso un profondo riutilizzo di codice. Senza dilungarsi negli aspetti più tecnici della OOP, verranno affrontate le caratteristiche principali che riguardano la programmazione ad oggetti:

1. Incapsulamento
2. Information Hiding
3. Ereditarietà
4. Polimorfismo

Definizione di classe

Per definire una classe si usa la sintassi riportata di seguito:

```
# definizione di una classe

class MiaClasse:
    nomeAttributo1 = ...
    nomeAttributo2 = ...
    nomeAttributo3 = ...

    def mioMetodoA(parametro1, parametro2, ...)
        ....
        ....
        return x

    def mioMetodoB(parametro1, parametro2, ...)
        ....
        ....
```

```
        return y

# PROGRAMMA CHIAMANTE
# Creazione di una Istanza della classe

oggettoX = MiaClasse()
oggettoY = MiaClasse()
```

Si noti come gli oggetti siano indipendenti tra loro. Per accedere agli attributi di un oggetto (o un metodo) si usa la sintassi **oggetto.attributo**. Nell'esempio che segue si assegna il valore "10" a **nomeAttributo1** dell' **oggettoX**.

```
# richiamare un attributo
oggettoX.nomeAttributo1 = 10

# richiamare un metodo
oggettoX.mioMetodoB(1, 4, ..)
```

I metodi dichiarati negli esempi precedenti, utilizzano parametri inviati dallo script che utilizzerà l'istanza, ma non accede agli attributi interni, caratterizzanti l'istanza stessa.

Il primo parametro che compare nella definizione di un metodo, proprio per poter richiamare tutti i parametri dell'oggetto stesso, è detto **parametro riflessivo** e si indica per convenzione con **self** nella definizione del metodo, mentre è sottinteso quando si richiama il metodo, questo aspetto emergerà negli esempi che seguiranno.

Sarà possibile sempre definire la classe in un file separato importarlo in un altro programma, attraverso il parametro *import*, come per tutti i moduli *built in*.

Incapsulamento e Information hiding

Osservando la dichiarazione di classe introdotta è possibile notare che tutti gli elementi sono inclusi nella dichiarazione relativa alla parola chiave "class". Ancora più evidente se si nota che la classe è tutta "indentata", vuol dire che le caratteristiche di classe ed i metodi implementati sono interni alla classe realizzata e quindi incluse nelle istanze costruite a partire dalla classe. Questo aspetto realizza l'**incapsulamento** perchè permette di vedere un *Oggetto* come un elemento all'interno del quale è incluso tutto quello che lo riguarda, caratteristiche e comportamenti.

La struttura complessa all'interno della quale sono racchiusi attributi e metodi è possibile vederla come una black box, cioè una scatola chiusa che nasconde ciò che è presente al suo interno. Secondo questa visione, quindi, è bene che gli attributi non siano accessibili direttamente da chi utilizza gli oggetti istanziati, ma solo attraverso i metodi progettati ed implementati dall'autore della classe. In tal modo è possibile vedere una classe come una scatola al cui interno sono presenti in modo nascosto le caratteristiche e i metodi sono le interfacce (finestre) attraverso cui leggere o modificare le caratteristiche protette al suo interno. Questa caratteristica prende il nome di **information hiding**. Il modo di rendere private le caratteristiche di un oggetto sarà spiegato nei paragrafi che seguiranno.

Il metodo Costruttore

Quando definiamo un oggetto, bisogna assegnarlo ad una variabile attraverso l'operatore di assegnazione "=", come avviene per qualunque tipo di variabile. Nel caso di un oggetto, questo può avvenire perché tutte le classi hanno un particolare metodo detto "costruttore", che inizializza tutti i parametri necessari alla creazione dell'oggetto da costruire. All'atto della creazione di un oggetto, quindi, viene richiamato il metodo di inizializzazione dichiarato come "`__init__(self,)`".

Il metodo costruttore permette di accedere agli attributi di istanza per inizializzarli e quindi particularizzare una determinata istanza. Nell'esempio che segue è possibile vedere il metodo costruttore e l'utilizzo del parametro riflessivo per accedere ai parametri dell'istanza ed inizializzarli.

```
class auto:

    #Metodo costruttore
    def __init__(self, proprietario, marca, modello, cilindrata, cavalli,
colore):

        self.proprietario = proprietario
        self.marca = marca
        self.modello = modello
        self.cilindrata = cilindrata
        self.cavalli = cavalli
```

Attributi di classe e d istanza

È stato introdotto il concetto di attributo come variabile appartenente alla classe di oggetti descritta. Gli attributi però nel dettaglio possono appartenere sia alla classe che alla singola istanza della classe. La differenza sostanziale è che:

- **Un attributo di classe** caratterizza tutti gli oggetti della classe, sempre modificabili, ma all'atto della creazione dell'istanza questo attributo non viene inizializzato, non caratterizza quindi la singola istanza. Un attributo di classe, in sostanza, appartiene principalmente alla classe e non al singolo oggetto creato e viene dichiarato prima di qualunque metodo e attributo d'istanza.
- **Un attributo dell'istanza** è, al contrario, inizializzato all'atto della creazione dell'oggetto ed è parte della sua peculiarità rispetto agli altri oggetti della classe.

Si osservi il seguente esempio.

oop/auto.py

```
'''
Python3
Programmazione ad oggetti
'''
class auto:

    # Attributi di Classe
    garanzia = 1
    assicurazione = True
    parcoAuto = 0

    #Metodo costruttore
    def __init__(self, proprietario, marca, modello, cilindrata, cavalli,
colore):

        # Attributi di Istanza
        self.proprietario = proprietario
        self.marca = marca
        self.modello = modello
```

```
        self.cilindrata = cilindrata
        self.cavalli = cavalli
        self.colore = colore

    auto.parcoAuto +=1

#Metodo di tipo Get
    def scheda(self):
        return f'\nScheda "{self.proprietario}"\n Marca: {self.marca}\n
Modello: {self.modello}\n Cilindrata: {self.cilindrata}\n Cavalli:
{self.cavalli}\n colore: {self.colore}\n assicurazione:
{self.assicurazione}'

# inizia il programma chiamante

if __name__ == "__main__":

    giovanni = auto("giovanni","ford","fiesta",1500, 160, "rosso")

    marco = auto("marco","fiat","Bravo",2500, 200, "verde")
    pippo = auto("marco","fiat","Bravo",2500, 200, "verde")

    print("Il tipo di variabile costruita è:")
    print(giovanni)
    print(marco)

    print("\nLa singola scheda è:")
    print (giovanni.scheda())
    print (marco.scheda())
    print("\nauto totali: ",auto.parcoAuto)

    print("\n\naltro metodo per visualizzare le informazioni
(__dict__):")
```


- Il metodo **costruttore** è necessario in tutte le classi ed è la funzione che permette la generazione delle istanze a partire dalla classe descritta. Il primo parametro deve essere *self*, seguiranno tutti i *gli attributi di istanza* che caratterizzano l'oggetto da creare.
- Dopo aver dichiarato il metodo costruttore vanno dichiarati *gli attributi di istanza* ai quali vanno associati i parametri di input del metodo costruttore. Si noti come il parametro **self** è usato proprio per richiamare gli attributi da dichiarare.
- Si noti che per richiamare un metodo o un attributo si utilizza il “ . ” che separa il nome dell'istanza (o della classe) dal metodo/attributo richiamato. Si faccia riferimento per esempio a **giovanni.scheda()** che richiama il metodo *scheda*, dell'oggetto *giovanni*. la differenza sostanziale tra un metodo ed un attributo sono le parentesi tonde, nel metodo vengono usate anche se vuote, per gli attributi no.
- Il metodo **scheda()** è un metodo di tipo *get*, perchè permette l'accesso in lettura ad uno o più attributi. Se avesse avuto accesso in modifica di uno o più attributi sarebbe stato un metodo di tipo *set*.
- **giovanni.__dict__** permette la lettura di tutti gli attributi dell'istanza per cui viene richiamato, formattando tutto sotto forma di dictionary. In tal modo è possibile ottenere la singola istanza con un formato standard.
- Si noti che la funzione chiamante è sottoposto alla verifica che lo script risulti “__main__”, così da lanciarlo solo ed esclusivamente se il file viene eseguito dall'interprete e non come file incluso e/o richiamato da un altro script, perchè nel caso in cui venisse incluso in un altro file quella parte di codice non verrebbe eseguita.

Una classe così costruita però non applica il principio di information hiding, perchè gli attributi sono accessibili dall'esterno della classe. Una tale implementazione infatti non protegge gli attributi da eventuali tentativi di modifica, sarà sempre possibile alterare ad esempio l'attributo nome con una semplice assegnazione di valore:

```
giovanni.proprietario = "paolo"
```

Con l'istruzione precedente è possibile mutare un attributo dell'oggetto “giovanni” rendendolo potenzialmente inconsistente. Per proteggere gli attributi da eventuali modifiche bisogna dichiararli privati e per farlo il nome deve avere due underscore (“__”) come primi caratteri. Il costruttore dovrà dichiarare la variabile in questo modo:

```
self.__proprietario = proprietario
```


Qualunque tentativo di accedere all'attributo *proprietario* restituirà un errore. Si noti che lo stesso approccio si può applicare anche ai metodi, rendendoli nascosti allo script che gestisce gli oggetti istanziati.

Ereditarietà

Il principio alla base della programmazione modulare è il riutilizzo di codice, aspetto che diventa ancora più evidente nel paradigma di programmazione ad oggetti. Attraverso il riutilizzo di una classe è infatti possibile ottimizzare il codice e poterne beneficiare in altre parti dello stesso software o in progetti differenti.

Per rendere questo aspetto ancora più efficace è possibile costruire classi a partire da altre classi, così da ereditare attributi e metodi implementati senza dover ripetere la stesura e la progettazione di algoritmi potenzialmente complessi. Il meccanismo così descritto prende il nome di **Ereditarietà**, che permette di avere una classe **padre** (detta **superclasse**) ed una classe **figlio** che utilizza tutte le parti di codice implementate a favore della classe padre.

Si osservi il seguente esempio:

oop/camper_simple.py

```
'''
importo la classe auto, così da rendere la sua implementazione
disponibile all'interprete
'''
from auto import auto

class camper(auto):
    pass

# INIZIO DEL PROGRAMMA CHIAMANTE

if __name__ == "__main__":

    pippo = camper("pippo","Ford","Transit",2500, 1600, "giallo")

    marco = auto("marco","fiat","Bravo",2500, 200, "verde")
```

```
print(pippo.scheda())
print(marco.scheda())
```

[illegible]

Scheda "pippo"

```
Marca: Ford
Modello: Transit
Cilindrata: 2500
Cavalli: 1600
colore: giallo
assicurazione: True
```

Scheda "marco"

```
Marca: fiat
Modello: Bravo
Cilindrata: 2500
Cavalli: 200
colore: verde
assicurazione: True
```

Si noti che:

- Nella prima linea di codice si importa il file relativo alla classe **auto**. La comprensione delle modalità di inclusione verrà affrontato nei paragrafi che seguono.
- La classe **camper** passa come valore (tra le parentesi tonde) la classe da cui eredita attributi e metodi. Il parametro **pass** serve per non generare un errore perchè avverte l'interprete che può andare avanti nell'analisi del codice, anche se non ci sono implementazioni di attributi e metodi.
- Nel programma chiamante vengono istanziati due oggetti, uno di tipo *auto* ed uno di tipo *camper*, per entrambi l'interprete cerca il metodo costruttore, se non lo trova (è il caso della classe *camper*) cerca il costruttore della classe “padre”, quindi applica il primo costruttore che trova, risalendo la catena ereditaria.
- Si noti come in output viene richiamato per entrambi gli oggetti il metodo “scheda()” anche se nella classe *camper* non è esplicitamente implementato.

Nel seguente esempio è esplicitato l'utilizzo del costruttore della **superclasse** (classe padre) e l'aggiunta di attributi specifici per la classe *camper*.

oop/camper.py

```
'''
importo la classe auto, così da rendere la sua implementazione
disponibile all'interprete
'''

from auto import auto

class camper(auto):

    def __init__(self, proprietario, marca, modello, cilindrata, cavalli,
colore, allestimento):

        # rimanda al costruttore della classe padre che si occupa degli
        attributi ereditati
        super().__init__(proprietario, marca, modello, cilindrata,
cavalli, colore)
        self.__allestimento = allestimento

    def scheda(self):
        scheda = f'''\n allestimento:{self.__allestimento}'''
        # utilizza il metodo "scheda()" ereditato
        return super().scheda() + scheda

# INIZIO DEL PROGRAMMA CHIAMANTE

if __name__ == "__main__":

    pippo = camper("pippo","Ford","Transit",2500, 1600, "giallo","laika")
```

```
marco = auto("marco","fiat","Bravo",2500, 200, "verde")

print(pippo.scheda())
print(marco.scheda())
```

<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<[OUTPUT]>>>>>>>>>>>>>>>>>>>>>>>>>

```
Scheda "pippo"
Marca: Ford
Modello: Transit
Cilindrata: 2500
Cavalli: 1600
colore: giallo
assicurazione: True
allestimento:laika
```

```
Scheda "marco"
Marca: fiat
Modello: Bravo
Cilindrata: 2500
Cavalli: 200
colore: verde
assicurazione: True
```

Si noti che:

- Il costruttore della classe `camper` utilizza il costruttore della classe `auto`, richiamandolo con la parola chiave **`super().__init__(...)`**, in tal modo l'interprete inizierà gli attributi di istanza attraverso il costruttore della classe padre, e poi aggiungerà gli attributi dell'oggetto `camper`, così come indicato nelle istruzioni successive.
- Il metodo **`scheda()`**, esattamente come il metodo costruttore, utilizza l'implementazione della classe padre per generare una parte dell'output, aggiungendo la parte di codice relativa al solo attributo aggiunto. Il risultato è ben visibile nella parte di output.

Polimorfismo

L'ultimo aspetto fondante la programmazione ad oggetti è la capacità dell'interprete (o del compilatore per altri linguaggi di programmazione) di scegliere il metodo corretto a secondo dell'oggetto che lo invoca. Un esempio di tale applicazione è il metodo **scheda()** che esegue uno script diverso in funzione dell'oggetto sul quale viene applicato.

Moduli

definire funzioni

Per affrontare i moduli è necessario ricordare il concetto di procedura e di funzione. Una procedura è uno script a cui è assegnato un nome, definito in modo da essere richiamato ogni volta dovesse servire.

Una funzione è una procedura che evolve a partire da parametri forniti come input ed eventualmente restituisce parametri in output.

Tra i vari benefici dell'utilizzo di funzioni possiamo notare:

- Lo sviluppo di funzioni semplifica la leggibilità di un programma, perché riduce gruppi di istruzioni a singole istruzioni da richiamare attraverso un nome ben identificabile.
- Raggruppare un insieme di istruzioni all'interno di una procedura/funzione alleggerisce la scrittura di codice, perché evita la ripetizione dello stesso gruppo di istruzioni più volte nel programma chiamante.
- L'eventuale correzione di una o più istruzioni interne ad una procedura andrebbe fatta una sola volta, non in tutti i punti del programma chiamante in cui quel blocco sarebbe stato presente se non fosse stato racchiuso in una funzione.

Un programma può includere più funzioni attinenti alle stesse attività e le definizioni possono essere riportate all'interno del medesimo file.

Per utilizzare una stessa funzione in più programmi, invece, è possibile copiarne la definizione e incollarla anche in altri file; questa tecnica è però sconsigliata perché rende laborioso l'aggiornamento dei programmi nei casi in cui occorra una modifica alla funzione stessa: sarebbe necessario andare ad apportare la medesima variazione in tutti quei file nei quali la definizione della funzione è stata incollata. Risulta quindi un buon approccio racchiudere tutte le funzioni utili in un unico file, che sarà chiamato modulo. Questo approccio alla programmazione gode di alcuni vantaggi e prende il nome di **approccio modulare alla programmazione**.

I vantaggi da evidenziare sono:

- **Modularità**: un programma può essere suddiviso in moduli indipendenti, più semplici da gestire;
- **Riutilizzabilità**: lo stesso modulo può essere utilizzato in programmi diversi;

Per realizzare una funzione python si osservi il seguente codice.

```
# Sintassi generale

# nella definizione bisogna indicare con "def" il nome, e tra parentesi
# gli eventuali parametri di input.
# Se non ci sono parametri di input si devono comunque mettere le
parentesi ().
def nome_funzione (parametro1, parametro2, ...):
    istruzione_1
    istruzione_2
    istruzione_3
    ...
    return risultato # se non c'è alcun valore di output si omette
l'istruzione return

# Esempio di funzioni

# valore di input: c
# valore di output il risultato di (c * 9 / 5) + 32

def temp(c):
    c=int(input("Enter a temperature in Celsius: "))
    return (c * 9 / 5) + 32

# programma chiamante
print(temp(c))
```

Si osservi che:

- il nome deve essere preceduto dal comando **def**.
- La funzione viene richiamata da un programma detto chiamante e per questo motivo deve essere dichiarata prima del suo utilizzo.
- Tutte istruzioni devono essere interne ad uno stesso blocco, in python questo si traduce nel rispettare l'indentazione di tutte le istruzioni presenti nella funzione da costruire.

Bisogna comunque sottolineare che una funzione non può modificare i parametri passati come input, ogni volta che gli si passa una variabile questa viene copiata in un'altra variabile ed utilizzata, si osservi il seguente codice:

[illegible]

Con l'esecuzione di "Modifica" quindi, non è possibile intervenire sul valore salvato in B. Utilizzando, invece, un array come parametro da fornire alla funzione, sarà possibile modificare la variabile originale.

Per quanto riguarda i parametri di output, questi vengono restituiti attraverso l'istruzione **return**, eventualmente possono anche essere restituiti più parametri, come nell'esempio che segue.

[illegible]

quoziente 6 resto 3

Utilizzo dei moduli

In python (come in molti linguaggi di programmazione) i file contenenti funzioni e altri oggetti utilizzabili da differenti applicazioni sono denominati **moduli**. Per richiamare una funzione definita in un modulo esterno si usa il comando **import**. Ci sono tre diverse modalità di utilizzo del comando:

```
# PRIMO APPROCCIO: import nome_modulo

import math

print(math.pi)    # stampa il valore pi grego
```

In questo modo verranno inclusi nel programma tutti gli oggetti del modulo importato, quindi si ha accesso a tutte le funzioni ed a tutte le variabili definiti nel modulo. L'accesso a un oggetto si ottiene specificando il modulo di appartenenza come prefisso, secondo la sintassi nome **modulo.objecto**, nell'esempio math.pi.

E' possibile includere anche determinati elementi scelti all'interno di un modulo, attraverso la seguente sintassi:

```
# SECONDO APPROCCIO: from nome_modulo import oggetto1, oggetto2

from math import pi

print(pi)
```

In questa modalità si ottiene l'accesso esclusivamente agli oggetti indicati nel comando; per riferirsi a essi all'interno del modulo non c'è bisogno di utilizzare il nome del modulo di origine come prefisso.

Se è necessario includere tutti gli oggetti di un modulo è possibile usare il carattere speciale asterisco "*".


```
# TERZO APPROCCIO: from nome_modulo import *  
  
from math import *  
print(e)
```

In questa modalità si ottiene l'accesso completo a tutti gli oggetti del modulo e non ci sarà bisogno di utilizzare il nome del modulo come prefisso.

Per riferirsi a un oggetto importato con la prima modalità è necessario utilizzare come prefisso il nome del modulo, negli altri due casi l'importazione fa in modo che la variabile e le funzioni importate siano conosciute direttamente nell'ambito del modulo in uso. Bisogna però stare attenti all'eventuale esistenza di elementi con lo stesso nome di quelli importati, perchè potrebbe generare conflitto e quindi mal funzionamenti.

Gestione e salvataggio dei moduli

Il modulo utilizzato tramite il comando import è ricercato dall'interprete all'interno di una lista predefinita di cartelle, dipendente dalla versione di python e dal sistema operativo. È possibile visualizzare la lista delle cartelle attraverso `sys.path`, una lista di stringhe relative ai path delle cartelle nelle quali sono presenti i moduli.

```
# importare il modulo standard sys ci da accesso al sys.path  
  
import sys  
print(sys.path)  
  
# in output avremo  
  
['', '/home/user/Documenti/git/google-chrome', '/usr/bin', '/usr/lib/python38.zip',  
'/usr/lib/python3.8', '/usr/lib/python3.8/lib-dynload',  
'/home/user/.local/lib/python3.8/site-packages', '/usr/lib/python3.8/site-packages']
```

È possibile anche creare dei moduli personali, vanno nominati e salvati in un file .py, questo file andrà salvato in una delle cartelle sopra indicate. Così facendo sarà possibile richiamare questo modulo anche da altre funzioni in esecuzione.

Spazio dei nomi

Per spazio dei nomi intendiamo l'elenco dei nomi e dei corrispondenti valori degli oggetti definiti in un modulo o in una funzione. Definiamo **namespace locale** quello relativo ad un modulo o ad una funzione, il namespace del modulo nel quale la funzione viene chiamata è detto namespace di livello superiore.

In questo modo si viene a creare una gerarchia di namespace uno dentro l'altro, come fossero scatole cinesi. Il namespace più esterno ad ogni applicazione python è quello relativo alle funzioni predefinite, dei moduli e delle parole riservate, ad esempio import o def (parole riservate alla gestione dei moduli, appunto). Questo namespace è chiamato **built-in**.

Le modalità di importazione di un modulo che abbiamo visto, gestiscono diversamente i namespace dei moduli:

- Il primo metodo lascia invariato il namespace locale, per questo è necessario richiamare il nome del modulo prima dell'oggetto.
- Il secondo ed il terzo modo di importazione del modulo fonde i due namespace, questo rende più facile la programmazione perché non bisogna richiamare il modulo. Controindicazione di questo approccio, come è stato notato in precedenza, è che aumenta il rischio di collisione tra i nomi, se gli oggetti del modulo si chiamano allo stesso modo degli oggetti del programma che si sta scrivendo.

Quando si utilizza il primo approccio è possibile utilizzare un alias invece di utilizzare il nome standard del modulo:

```
import math as m
print(m.pi)
```

Visibilità delle variabili

Quando si utilizza una variabile, l'interprete cerca quella variabile prima di tutto nel namespace locale della funzione, all'interno della quale la variabile è stata dichiarata, se non la trova cerca nel namespace che contiene quella funzione e così via fino al namespace built-in.

Una variabile dichiarata in una funzione risulta sconosciuta all'esterno di quella funzione. Bisogna quindi considerare le funzioni come singoli blocchi che permettono una visibilità delle variabili solo all'interno di se stessi, non all'esterno. Questa logica vale per tutti i contenitori, dei moduli fino ai blocchi più piccoli, come le istruzioni racchiuse in un ciclo.

Di conseguenza:

- Se dichiaro una variabile in una funzione, questa non sarà utilizzabile al di fuori di essa.
- Se dichiaro una variabile all'esterno di una funzione, nel programma chiamante, sarà visibile all'interno della funzione. Una variabile dichiarata in questo modo prende il nome di variabile globale.

```
# x variabile globale, dichiarata all'esterno di una funzione
# e visibile anche all'interno della funzione

x = 10
def funzione_esempio():
    y = x
    y += 1
    return y

print(funzione_esempio())

>> 11
```

```
# temp è una variabile locale, dichiarata all'interno di una funzione
# e non visibile all'esterno di essa

def mia_funzione():
    temp = 12
    print(spam)

frutta = temp + 6
>>> NameError: name 'temp' is not defined
```

Variabili dei moduli

assegnare nuovamente un valore. All'interno di un modulo, oltre a dichiarare funzioni, è possibile dichiarare delle variabili con assegnazioni iniziali, così da separare la dichiarazione di queste variabili dal codice.

Un esempio di variabili contenute in un modulo è quello relativo al modulo standard **string**.

La variabile **string.digits** può essere utilizzata per verificare (attraverso un confronto) che tutti i caratteri immessi da tastiera siano di tipo numerico prima di procedere a un'operazione, così da prevenire inserimento di lettere da parte dell'utente.

Analogamente è possibile utilizzare le variabili incluse nel modulo string per verificare se un determinato valore è una lettera minuscola, maiuscola o una rappresentazione esadecimale.

Moduli personali

Per inserire funzioni, variabili o costanti personalizzate, sarà sufficiente inserirle in un file con estensione .py ed importarlo nel programma che dovrà utilizzarle.

In alternativa è possibile salvare il file in una cartella ed includerla nella lista delle cartelle standard da cui i programmi caricano i moduli. La lista delle cartelle vista in precedenza, sys.path, è una normale lista, quindi sarà possibile inserire un elemento come in una lista qualsiasi:

```
sys.path = sys.path + ['c:\python39\moduli_personali']
```

I moduli standard

Python include in modo nativo diverse decine di moduli, per visualizzarli bisogna eseguire il comando **help("modules")**.

Saranno descritti i principali moduli standard di uso comune.

Modulo Math

Il modulo math prevede molte funzioni spesso utilizzate per elaborazioni matematiche.

math.pow(x,y)	Restituisce la potenza con base x disponente y
math.sqrt(x)	restituisce la radice quadrata di x

<code>math.ceil(x)</code>	restituisce il più piccolo intero maggiore o uguale a x
<code>math.floor(x)</code>	restituisce il più grande intero minore o uguale a x
<code>math.factorial(x)</code>	Restituisce il fattoriale di x (con x intero positivo)

L'elenco completo di tutte le funzioni incluse nel modulo Math è possibile visualizzarlo attraverso `help(math)` nell'IDLE di Python.

Modulo Random

Spesso durante la programmazione di alcune funzioni può essere utile generare numeri casuali. In realtà i numeri non sono mai puramente casuali ma sempre il frutto di calcoli, seppur complessi e difficilmente prevedibili, per questo motivo vengono detti numeri **pseudo-casuali**. in Python questi numeri sono prelevati da una sequenza predeterminata di $2^{19937} - 1$ numeri, questo garantisce con una discreta affidabilità la non ripetizione dei numeri forniti.

La funzione **`random()`**, del modulo Random, restituisce quindi un numero pseudo casuale appartenenti all'intervallo che inizia da zero ed arriva a 1 escluso: `[0,1)`.

```
import random
x = random.random()
print(x)

## in output vedremo ##

0.56940303958863
```

Ogni volta che avvieremo lo script nell'esempio precedente genererà un valore diverso. Nella tabella seguente, invece, sono descritte alcune delle funzioni maggiormente utilizzate.

<code>random.randint(a,b)</code>	Restituisce un numero intero appartenente all'intervallo con estremi inclusi, <code>[a,b]</code> .
<code>random.choice(oggetto)</code>	Consente di prelevare un elemento a caso tra quelli che compongono l'oggetto indicizzabile passato come parametro.
<code>random.shuffle(lista)</code>	Applicato ad una lista restituisce la stessa lista, ma con gli elementi mescolati a caso.
<code>random.sample(oggetto,n)</code>	Estrae una lista di n oggetti a caso a partire dall'oggetto di m elementi (con n minore o uguale di m) passato come parametro, sia Esso una tupla una lista o insieme.

Per come è stato presentato la funzione che restituisce numeri casuali, è possibile dedurre che a parità di condizioni restituirà sempre la stessa sequenza di elementi, anche perchè l'esecutore esegue un algoritmo ed in quanto tale ripete delle istruzioni, sempre le stesse. In taluni casi risulta invece utile avere sempre la stessa sequenza di valori casuali presi da una stessa lista, per fare questo si usa la funzione `seed()`. Si indica quindi un valore seme con il quale inizializzare il generatore di numeri casuali. In tale modo sarà possibile, quindi, personalizzare il numero iniziale ed ottenere la stessa sequenza di valori casuali più volte.

Modulo Time

Ogni computer considera il tempo a partire da un determinato istante iniziale, vengono così calcolati i secondi complessivi "di vita" del computer. Vuol dire che tutte le date e tempi vengono considerati come un contatore di secondi a partire da quel momento iniziale. La funzione **`time()`** restituisce il numero di secondi a partire dal quel punto iniziale, che prende il nome di **`epoc`**. La funzione **`clock()`** restituisce invece il numero di secondi, ma in formato float, quindi con una precisione inferiore al secondo (anche se non tutti i sistemi lo permettono).

Secondo questa logica per conoscere una data bisogna convertire quel numero di secondi in giorno, mese e anno, questa conversione è realizzata dalla funzione **`gmtime(n)`** del modulo Time. Questa funzione prende in input un numero intero corrispondente ai secondi trascorsi e restituisce un oggetto di tipo **`struct_time`**, equivalente a una tupla di 9 numeri interi aventi il seguente significato:

Indice	Attributo	Significato	Valori ammessi
0	<code>tm_year</code>	Anno	
1	<code>tm_mon</code>	Mese	[1,12]
2	<code>tm_mday</code>	Giorno del mese	[1,31]
3	<code>tm_hour</code>	Ora	[0,23]
4	<code>tm_min</code>	Minuto	[0,59]
5	<code>tm_sec</code>	Secondi	[0,61]
6	<code>tm_wday</code>	Giorno della settimana	[0,6] 0 Domenica, 1 Lunedì...
7	<code>tm_yday</code>	Giorno dell'anno	[1,366]
8	<code>tm_isdst</code>	ora legale	0 Solare, 1 Legale, -1 sconosciuta

esempio di utilizzo del tipo **`struct_time`**:

```
import time

# Voglio sapere il giorno del mese della data x
time.gmtime(23193723908) [2]

>> 24

# Voglio sapere il giorno del mese della data x
time.gmtime(23193723908) [7]

>> 359
```

Di seguito alcuni esempi di funzioni sul tempo, almeno quelle più comunemente usate:

- **process_time()** restituisce i secondi usati dalla CPU per il processo, senza le attese.
- **process_time_ns()** restituisce i nanosecondi usati dalla CPU per il processo, senza le attese.
- **perf_counter()** restituisce il valore totale dei secondi trascorsi dall'inizio del processo.
- **perf_counter_ns()** restituisce il valore totale dei nanosecondi trascorsi dall'inizio del processo.
- **sleep(secs)** sospende l'esecuzione del processo per il numero di secondi (float) indicato; il tempo di sospensione effettivo può differire da quello richiesto.

Modulo Exception

Prima di eseguire un programma è possibile verificare la presenza di **errori di tipo sintattico** attraverso la voce **check Module** del menu **Run** della IDLE di Python.

Gli errori di tipo **semantico** o **logico**, invece, sono più difficili da prevedere e spesso emergono solo durante l'esecuzione del programma, fornendo risultati inattesi, concludendo l'esecuzione ma in modo inesatto.

Un'altra classe di errori sono quelli detti di **runtime**, molto simili a quelli semantico/logici, ma che restituiscono tentativi di realizzare operazioni non consentite.

Ad esempio non è possibile effettuare la divisione per zero, nel caso dovesse capitare python arresta lo script e restituisce un errore all'utente, probabilmente poco comprensibile.

Nel seguente script si calcola la media dei voti di uno studente:

```
somma = 0
n = 0
```

```
domanda = 'inserisci il voto ( zero per fermare): '  
voto = input(domanda)  
  
while voto != '0':  
    somma = somma + float(voto)  
    n = n+1  
    voto = input (domanda)  
media = somma / n  
print( ' la media è: ', media)
```

Il programma restituirà un errore se non viene inserito alcun valore.

```
inserisci il voto ( zero per fermare): 0  
  
Traceback (most recent call last):  
  File "/home/user/Documenti/git/labPy/tutorial/esempi/media.py",  
line 12, in <module>  
    media = somma / n  
ZeroDivisionError: division by zero
```

Attraverso il modulo *Exception*, presente nel sistema, è possibile intercettarli, per gestirli o anche solo per renderli chiari all'utente.

Per intercettare questo tipo di errori bisogna inserirli in una sequenza che si chiama **try-except** come nell'esempio seguente e affrontato nel capitolo precedente.

```
somma = 0  
n = 0  
domanda = 'inserisci il voto ( zero per fermare): '  
  
try:  
    voto = input(domanda)  
    while voto != '0':  
        somma = somma + float(voto)  
        n = n+1  
        voto = input (domanda)  
media = somma / n
```



```
print( ' la media è: ', media)
except:
    print("inserisci almeno un voto")
```

In questo modo sarà possibile intercettare la divisione per zero, così da segnalarela con chiarezza all'utente dello script. Non verrà però distinto alcun errore, restituirà sempre lo stesso messaggio, anche se l'errore dipenderà dall'inserimento di un carattere letterale in luogo di uno numerico previsto.

Nel seguente script, invece, si differenziano i diversi tipi di errore, così da guidare al meglio l'utente.

```
somma = 0
n = 0
domanda = 'inserisci il voto ( zero per fermare): '

try:
    voto = input(domanda)

    while voto != '0':
        somma = somma + float(voto)
        n = n+1
        voto = input (domanda)

    media = somma / n
    print( ' la media è: ', media)

except ZeroDivisionError:
    print('inserisci almeno un voto')

except ValueError:
    print('non è un numero valido')

except:
    print("errore generico")
```

Si noti che la ricerca di errore generica va sempre messa in coda a tutte le altre.

L'elenco degli errori riconoscibili sono indicate è disponibile nella documentazione on line di python.

In genere, quando si gestiscono gli errori, risulta più utile utilizzare una funzione che restituisce un codice di errore, così da gestirlo internamente al programma, senza utilizzare la funzione print di comunicazione con l'utente, attraverso una stringa di output.

Per rilevare errori logici, quindi con il verificarsi di una condizione, si utilizza la funzione **raise**, come descritto nel seguente esempio.

```
def verificaVoto(voto):  
    try:  
        votoSuf= int(voto)  
        if (votoSuf < 0) or (votoSuf > 10):  
            raise RuntimeError()  
  
        if (votoSuf < 6):  
            return 'bocciato'  
        elif:  
            return 'promosso'  
  
    except RuntimeError:  
        return 0  
    except:  
        return -1
```

La funzione così costruita restituirà un codice di errore che potrà eventualmente essere gestito dalla funzione chiamante.

Modulo OS

Nella descrizione basilare di Python è stato sottolineata la capacità del linguaggio di rendere gli script indipendenti dal sistema operativo, sfruttando le potenzialità di un linguaggio interpretato. Tuttavia alcuni casi può tornare utile sfruttare le caratteristiche del S.O. nel quale si agisce, per fare questo si sfrutta il modulo **OS**. Per fare ciò bisogna prima importare il modulo e successivamente richiamare la funzione **popen()**, alla quale va passata l'istruzione desiderata. Ad esempio in Windows la seguente istruzione modifica il nome sul bordo esterno della finestra di esecuzione. La funzione popen, in sostanza, crea una comunicazione diretta con il comando cmd di windows, per l'esecuzione del comando inserito tra parentesi.

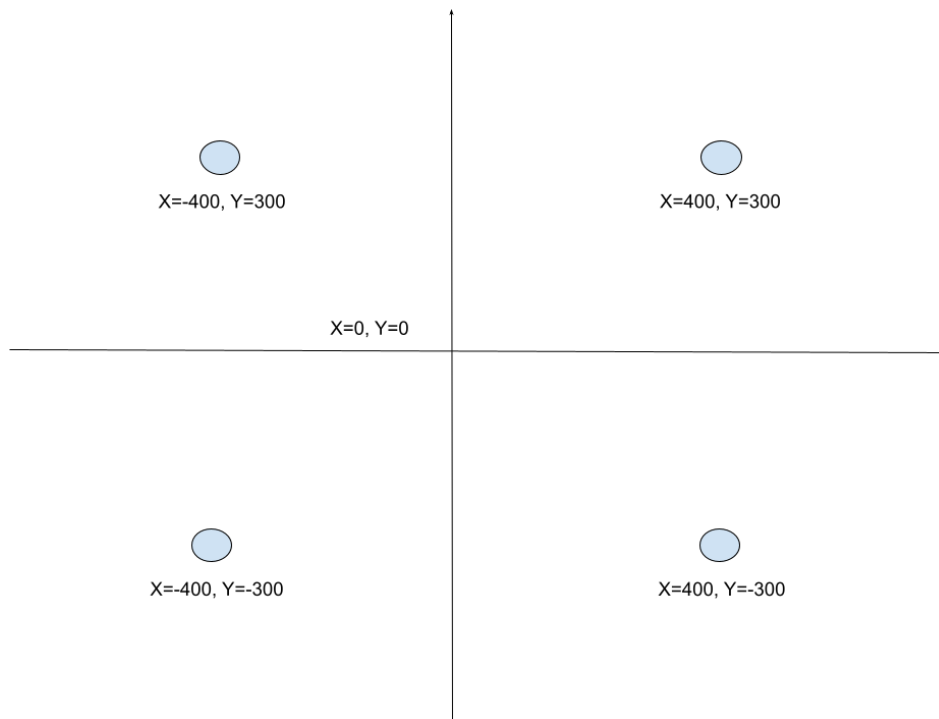
```
import os

os.popen('titolo nuovo')
```

Turtle

Il modulo turtle permette di creare disegni geometrici attraverso il movimento di un'ipotetica tartaruga rappresentata da una freccia.

Prima di iniziare a disegnare dobbiamo inizializzare una tartaruga assegnandole un nome, negli esempi che seguono utilizzeremo *Tarty*. Il movimento della tartaruga avviene attraverso posizioni del piano cartesiano così descritto:



Si consideri il seguente esempio:

```
turtle/turtle-istruzioni-base.py
```

```
'''
```

```
per tutte le istruzioni relative al modulo turtle si faccia riferimento
alla documentazione
della libreria
https://docs.python.org/3/library/turtle.html
'''

from turtle import Turtle, Screen

# inizializzo la tartaruga
tarty = Turtle()

# per visualizzare una tartaruga al posto della freccia
tarty.shape('turtle')

# visualizza uno sfondo e assegna un colore con la codifica RGB
esadecimale
sfondo = Screen()
sfondo.colormode(255)

R = 0
G = 255
B = 0
sfondo.bgcolor((R, G, B))

# per identificare la posizione della tartaruga

print(tarty.xcor(), tarty.ycor())

'''

movimenti della tartaruga
La velocità può essere impostata d un valore compreso tra 1 o 10.
Esistono anche velocità standard
'fastest' : 0
'fast'    : 10
'normal'  : 6
```

```
'slow'      : 3
'slowest'   : 1
'''

tarty.speed('normal')

tarty.pensize(10) # definisce lo spessore della penna
tarty.reset()    # cancella lo schermo
tarty.forward(100) # va avanti di 100 pixel
tarty.right(90)   # gira a destra
tarty.backward(90) # cammina all'indietro indietro di 90 pixel
tarty.left(90)    # gira a sinistra
tarty.color('red') # definisce un colore
tarty.circle(20)  # genera un cerchio con il raggio di 10 pixel
tarty.penup()     # "alza la penna" dal foglio, quindi non scrive
tarty.forward(100)
tarty.pendown()   # abbassa la penna sul foglio per scrivere
tarty.goto(100,-200) # salta alla posizione
#tarty.setpos(100,100) uguale a goto()
#tarty.setposition(100,100) uguale a goto()
tarty.hideturtle() # nasconde la tartaruga
tarty.forward(100)
tarty.showturtle()

aspetta = input("clicca un tasto")
```

il, file ampiamente commentato, permette di comprendere le istruzioni basilari per il movimento della tartaruga. In particolare si noti che:

- Bisogna inizializzare una tartaruga assegnandogli un nome ed è possibile visualizzarla sotto forma di freccia o di tartaruga.
- È possibile assegnare un colore di sfondo al piano di lavoro.
- È possibile definire una velocità di movimento.

Si faccia riferimento, per tutte le funzioni possibili, alla documentazione ufficiale <https://docs.python.org/3/library/turtle.html>

Figure geometriche

È facile intuire che attraverso l'uso dei cicli si rende possibile il disegno geometrico di figure regolari. Attraverso l'uso di di cicli innestati sarà possibile generare arte geometrica.

Si faccia riferimento al seguente esempio per la realizzazione di poligoni regolari:

```
turtle/turtle-geometria.py

from turtle import Turtle, Screen

tarty = Turtle()
sfondo = Screen()

sfondo.colormode(255)

R = 0
G = 255
B = 0
sfondo.bgcolor((R, G, B))

tarty.speed(3)
tarty.turtlesize(10)

# quadrato
tarty.goto(-100,-100)

tarty.begin_fill() # inizia un disegno a colore pieno
tarty.color('red')
for i in range(4):
    tarty.forward(100)
    tarty.left(90)
tarty.end_fill() # finisce il disegno a colore pieno

tarty.penup()
tarty.goto(100,-100)
```

```
tarty.pendown()
tarty.begin_fill() # inizia un disegno a colore pieno
tarty.color('blue')

for i in range(5):
    tarty.forward(100)
    tarty.left(60)

tarty.end_fill() # finisce il disegno a colore pieno

wait = input("PRESS ENTER TO CONTINUE.")
```

L'utilizzo di funzioni e di cicli permette la costruzione di figure molto complesse. Sarebbe possibile, infatti, considerare una singola figura geometrica all'interno di una funzione e ripeterla più volte, così da generare texture articolate come nell'esempio che segue:

turtle/turtle-cicli-innestati.py

```
import turtle

tarty = turtle.Turtle()
tarty.speed('fast')

for n in range(36):
    for i in range(6):
        tarty.forward(100)
        tarty.left(60)
    tarty.right(10)

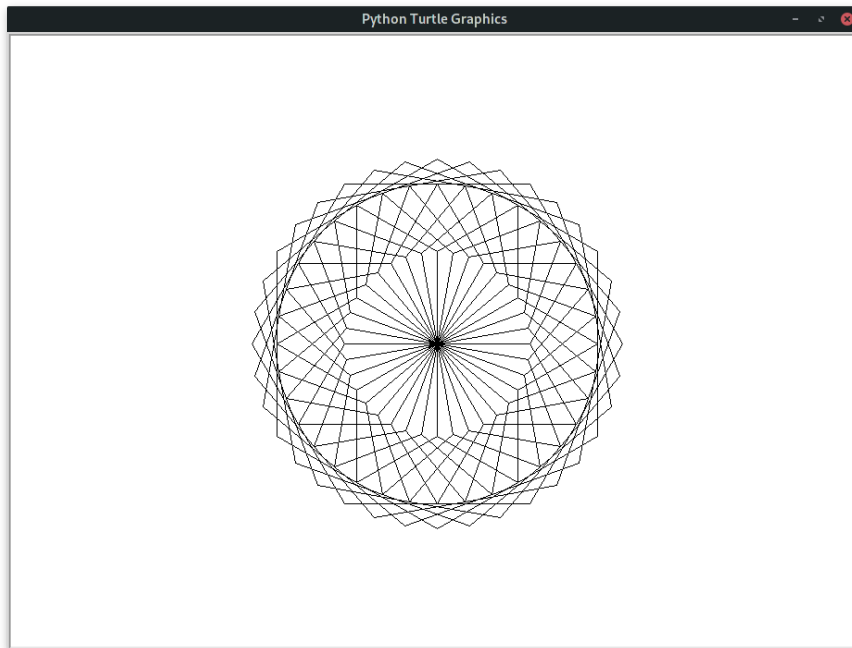
'''
si sarebbe potuto racchiudere l'esagono in una funzione.

def esagono():
    for i in range(6):
        tarty.forward(100)
```

```
tarty.left(60)

for i in range(36):
    esagono()
    tarty.right(10)

'''
wait = input("premi invio per chiudere.")
```



Si noterà come sono stati innestati due cicli, uno dentro l'altro, così da ripetere 36 volte l'esagono che a sua volta si realizza con una ripetizione di 6 volte `tarty.forward(100)` e `tarty.left(60)`. Si noti inoltre che i cicli sono stati realizzati attraverso l'istruzione **range(n)**, che permette di generare una *lista* di n numeri interi, così da poter gestire al meglio il costrutto ciclico For. L'esempio include la funzione `esagono()`, si provi a levare gli apici che commentano, ed eseguire il codice commentato così da realizzare la stessa figura. L'uso delle funzioni è altamente consigliato, come è noto, per migliorare la leggibilità del codice, la manutenzione e gestione degli errori.

GUI - interfaccia utente

Una GUI (Graphical User Interface) È un tipo di interfaccia utente che consente l'interazione senza l'utilizzo del prompt dei comandi. Python mette a disposizione, all'interno dei moduli base, il modulo Tkinter che permette la costruzione di interfacce semplici così da poter gestire le applicazioni in modo agevole. Si noti che questo modulo non è l'unico ed a partire da questo ne sono stati creati altri con caratteristiche differenti o più specifiche per determinati tipi di attività.

Nella documentazione relativa a tkinter, all'interno del sito di Python, sono presenti esempi e specifiche sul suo utilizzo, anche in relazione alle diverse versioni di Python (<https://docs.python.org/3/library/tkinter.html>).

Guizero

Il modulo Guizero (<https://pypi.org/project/guizero/>) è uno di quei moduli derivati da un fork di *tkinter* al quale sono state aggiunte diverse funzionalità e facilitazioni. l'installazione risulta particolarmente semplice anche attraverso il gestore di pacchetti *Pip*, sul sito comunque ed è scritto bene il procedimento di installazione e sulla relativa pagina di *github* sono pubblicate altre istruzioni per il suo utilizzo (<https://lawsie.github.io/guizero/>).

Un primo esempio che possiamo realizzare è la classica finestra *hello World*, riportato nell'esempio che segue. Bisogna sottolineare che, come vedremo, l'oggetto chiamato **App** è in sostanza il contenitore grafico all'interno del quale inseriremo i nostri elementi.

helloworld.py

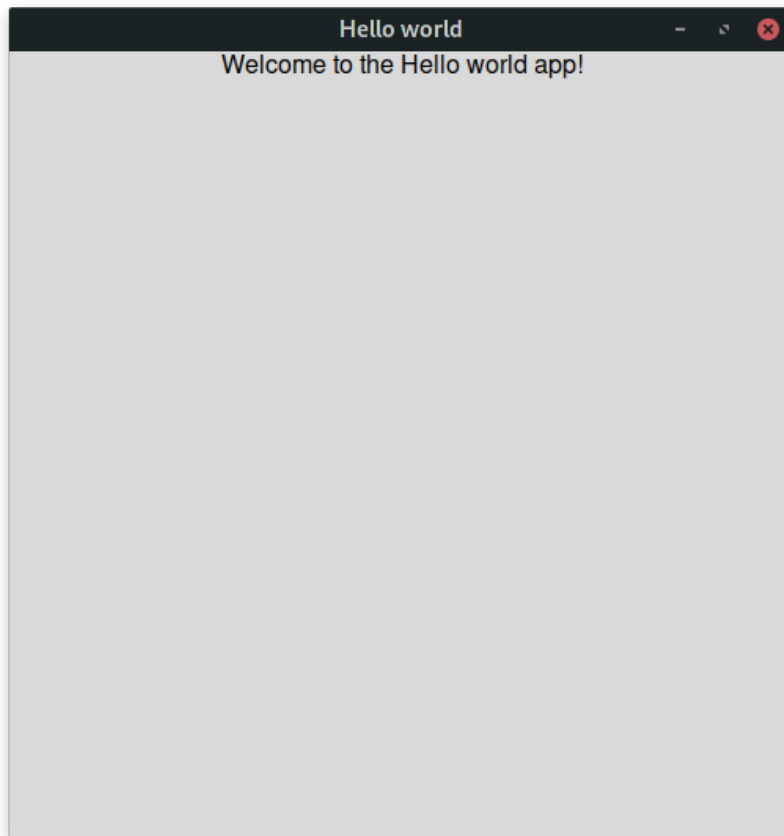
```
# importo dalla libreria le classi di oggetti App e Text
from guizero import App, Text

# creo una istanza di App e la associo all'oggetto app
# importo dalla libreria le classi di oggetti App e Text
from guizero import App, Text

# creo una istanza di App e la associo all'oggetto app
app = App(title="Hello world")
```

```
# creo una istanza dell'oggetto Text e la associo all'oggetto message
message = Text(app, text="Welcome to the Hello world app!")

# restituisco a momitor l'oggetto app
app.display()
```



Se si volesse importare tutto il modulo, ovviamente, si sarebbe dovuto importare l'insieme degli oggetti della libreria:

```
from guizero import *
```

Nell'esempio che segue viene costruito un oggetto finestra all'interno del quale sono presenti tre tasti, associati a tre funzioni differenti. ciascuna funzione restituisce in output, all'interno della finestra un valore una stringa di testo.

tasti.py

```
#GUIZERO

from guizero import *
from random import *

def tastol():
    output.append("abbiamo un testo")
    output.append("fine del primo output")
    output.append("")

def tastol2():
    for i in range(0,5):
        output.append(i)
        i+=1
    output.append("fine del secondo output")
    output.append("")

def tastol3():
    output.append(randint(0,100))
    output.append("fine del terzo output")
    output.append("")

app = App(title="Esempio", width=500, height=300, bg="#f4d742")

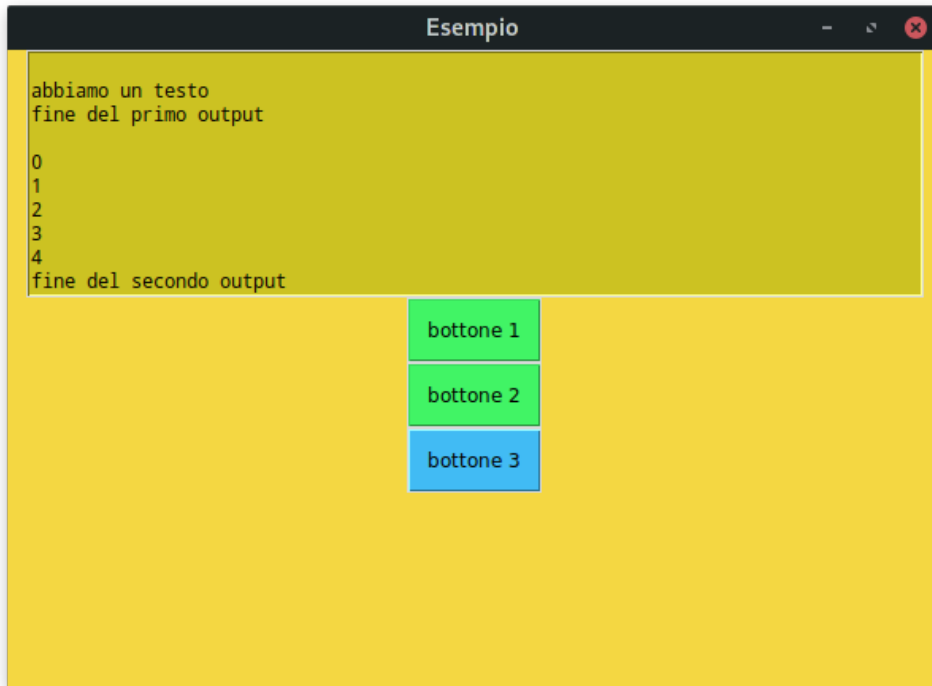
output = TextBox(app, width=80, height=10, multiline=True)

p1 = PushButton(app, text=' bottone 1 ', command=tastol)
p2 = PushButton(app, text=' bottone 2 ', command=tastol2)
p3 = PushButton(app, text=' bottone 3 ', command=tastol3)

p1.bg = "#41f465"
p2.bg = "#41f465"
p3.bg = "#41bbf4"
```

```
output.bg = "#ccc222"
```

```
app.display()
```



Volendo analizzare il codice così da individuare le funzionalità essenziali possiamo evidenziare i seguenti elementi:

- **def tast1(), tast2() e tast3().** Vengono definite tre funzioni che restituiscono rispettivamente una stringa, la sequenza di numeri compresa fra 1 e 5 ed un numero intero casuale compreso tra 0 e 100.
- **output.append().** Richiama il metodo *append* dell'oggetto *output* appartenenti alla libreria *guizero*. questo oggetto permette di aggiungere (append) all'interno della finestra che in genere corrisponde all'oggetto *app*, l'output passato come parametro all'interno delle parentesi tonde.
- **App(title="Esempio", width=500, height=300, bg="#f4d742").** Questo metodo permette di costruire una finestra che verrà assegnata all'oggetto *app*. Si noti come i parametri di input di questo metodo definiscano titolo, altezza, larghezza e colore della finestra creata.
- **TextBox(app, width=80, height=10, multiline=True).** Il metodo *TextBox* costruisce uno spazio all'interno del quale sarà possibile collocare testo e verrà successivamente associata ad un oggetto chiamato *output* destinato a contenere gli output delle funzioni

creato in precedenza. Si noti come i parametri di input di questo metodo sono la finestra principale all'interno della quale collocare il textbox, larghezza, altezza ed un parametro che indica la possibilità di rappresentare testo su più linee.

- **PushButton(app,text='bottone 1',command=tasto1).** Il metodo *PushButton*, permette di creare dei tasti all'interno della finestra, indicata come primo parametro (app), recanti un testo indicato come secondo parametro (text) ed una funzione da richiamare indicato come terzo parametro (command). questi tasti devono essere assegnati a delle variabili. Nel nostro caso P1 P2 e P3.
- **p1.bg.** p1 (e rispettivamente p2 e p3) è un oggetto di tipo PushButton, quindi è possibile richiamare metodi relativi a quella classe di oggetti; **bg** è il metodo che permette di assegnare un colore al singolo tasto personalizzando l'effetto grafico. Come è possibile vedere nel codice il metodo BG è utilizzabile anche sull'oggetto output, così da personalizzare ulteriormente la finestra all'interno della quale vengono restituiti gli output delle tre funzioni, richiamate dei tre pulsanti.
- **app.display().** Il metodo display permette di visualizzare l'elemento grafico corrispondente all'oggetto app, che come abbiamo visto contiene tutti gli elementi precedentemente creati. In sostanza l'oggetto display permette la visualizzazione dell'area di lavoro all'interno della quale abbiamo collocato tutti gli elementi creati.

Risulta scontato immaginare che tutti gli elementi all'interno dell'area di lavoro possono essere allineati, personalizzati ed organizzati come risulta meglio rappresentata l'interfaccia del software ipotizzato.

Menu

L'oggetto **MenuBar**, importabile direttamente dalla libreria, permette la creazione di menù in cima alla finestra creata attraverso un array che rappresenta la lista di finestre e la lista di voci all'interno delle singole finestre punto l'esempio riportato di seguito estratto direttamente dal sito relativo alla documentazione della libreria crea un menù a due voci. Si noti inoltre che ogni voce a due parametri un'etichetta che la contraddistingue ed una funzione da richiamare al click del mouse.

menu.py

```
# MENU

from guizero import App, MenuBar

def file_function():
```

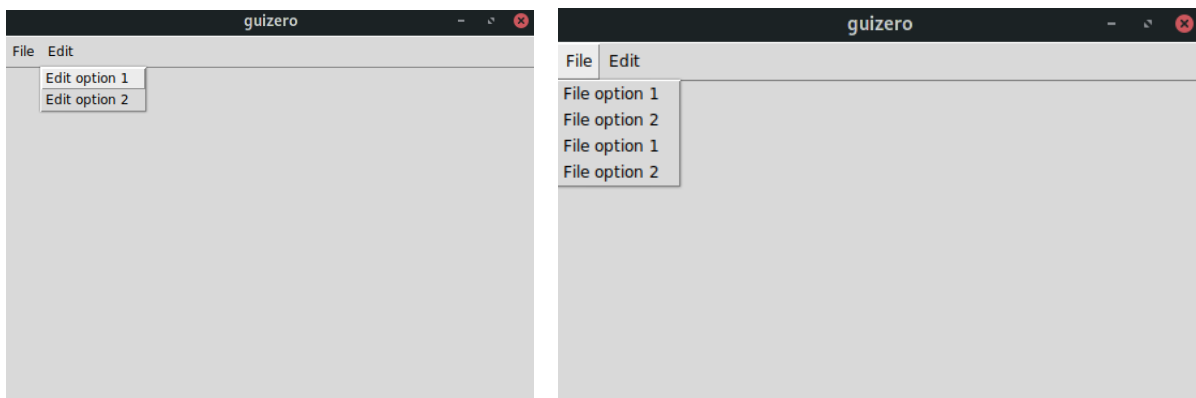
```

print("File option")

def edit_function():
    print("Edit option")

app = App()
menubar = MenuBar(app,
                    toplevel=["File", "Edit"],
                    options=[
                        [ ["File option 1", file_function], ["File option
2", file_function],["File option 1", file_function], ["File option 2",
file_function] ],
                        [ ["Edit option 1", edit_function], ["Edit option
2", edit_function] ]
                    ])
app.display()

```



Si noti che:

- Sono state definite prima le funzioni che verranno richiamate, come nell'esempio dei tasti.
- All'oggetto **menubar** è stato assegnato il metodo che costituisce un oggetto della classe **MenuBar**, passandogli come parametri l'oggetto all'interno del quale collocare il menù, quindi l'area di lavoro identificata dalla variabile *app*; la lista delle etichette relative ai titoli dei vari menù a tendina associate al parametro **toplevel**; la lista, sottoforma di array a più dimensioni, di tutte le voci dei singoli menù attraverso il parametro **options**.

Da notare è che il parametro *options* associa il primo elemento del vettore al primo elemento associato al parametro *toplevel*, così il secondo, il terzo ecc.

il **MenuBar**, inoltre, non può essere rappresentato in una griglia quindi non prevede parametri di allineamento del testo e di celle.

ListBox

Una **listbox** è un oggetto che elenca elementi cliccabili con il mouse punto esistono diversi parametri per la configurazione di questo oggetto grafico, da impostare di volta in volta attraverso i parametri previsti. L'esempio che segue realizza una listbox con un valore predefinito già selezionato e con il vincolo di non poterne selezionare più di uno contemporaneamente. L'output viene visualizzato in un'area di testo evidenziata da un differente colore all'interno dell'area di lavoro.

listbox.py

```
"""
GUIZERO - listbox
"""
from guizero import *

def f1():
    output.append("Questo è il primo tasto")

def f2():
    for i in range(0,5):
        output.append(i)
        i+=1

def f3():
    output.append("FINE")

def selectf():
    output.append(lista.value)

app = App(title="Esempio ", width=500, height=600, bg="#e3adad")
```

```

p1 = PushButton(app,text='bottone 1',command=f1)
p2 = PushButton(app,text='bottone 2',command=f2)
p3 = PushButton(app,text='bottone 3',command=f3)

menubar = MenuBar(app,
                    toplevel=["File", "Edit"],
                    options=[
                        [ ["File option 1", f1], ["File option 2",
f1],["File option 3", f1], ["File option 4", f1] ],
                        [ ["Edit option 1", f1], ["Edit option 2", f1] ]
                    ])

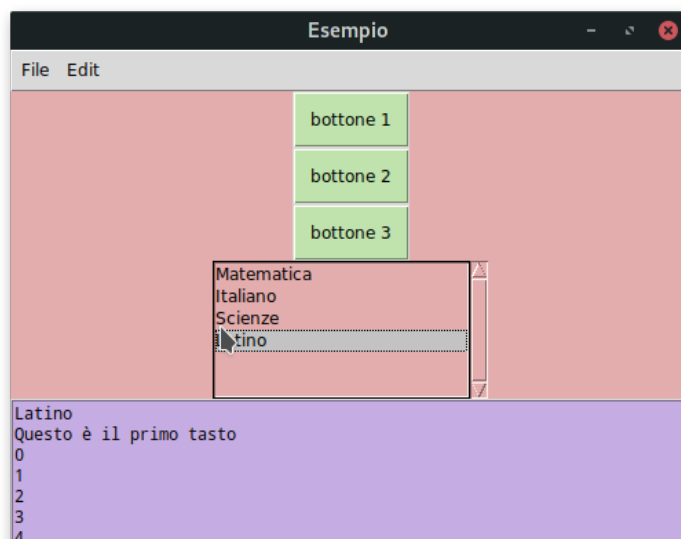
lista = ListBox(app, items=["Matematica", "Italiano", "Scienze",
"Latino"], selected="Matematica", command=selectf, grid=None,
                align="left",
                visible=True,
                enabled=True,
                multiselect=False,
                scrollbar=True,
                width=200,
                height=100)

output = TextBox(app, width=80, height=10, multiline=True)

p1.bg = "#c0e3ad"
p2.bg = "#c0e3ad"
p3.bg = "#c0e3ad"
output.bg = "#c5ade3"

app.display()

```



Nel codice precedente è possibile notare che:

- Tutti i valori restituiti dalle funzioni richiamate attraverso i tasti e la *listbox* vengono inseriti all'interno dell'area di testo libero.
- I parametri di input per la funzione che crea una **Listbox** sono:
 - L'oggetto *app*, all'interno del quale verrà collocata, come primo parametro.
 - Le etichette della lista da poter selezionare in modo mutuamente esclusivo hanno combinato. La lista deve essere associata al parametro **items** come secondo input della funzione.
 - Tutti gli altri parametri sono opzionali, significa che di default il modulo *guizero* assocerà un valore di default.

Acquisire parametri dall'utente

Nel caso si voglia acquisire informazioni dall'utente bisognerà prevedere all'interno dell'interfaccia dei campi di testo, così da acquisire quel valore ed elaborarla all'interno di una funzione. Nell'esempio che segue verranno acquisiti due valori di input di tipo numerico che verranno elaborati e restituiti come output in un'area di testo.

inputUtente.py

```
from guizero import *

def Moltiplica(a, b):
    x = int(a.value)
    y = int(b.value)

    risultato = x * y
    output.value = risultato

app = App(title="Moltiplica",bg="#f5f5f5")

output = TextBox(app, width=80, height=10, multiline=True)

etichettaA = Text(app, text="Inserisci A:")
paramA = TextBox(app)
etichettaB = Text(app, text="Inserisci B:")
```



```
paramB = TextBox(app)

pushB = PushButton(app, text="Moltiplica!", command=Moltiplica,
args=[txtb_base, txtb_esp])

app.display()
```

Si noti come nella riga di codice associata al tasto della moltiplicazione (pushB) i parametri riguardano l'oggetto *app* relativo all'area in cui apparire, la funzione da eseguire e la lista dei parametri da acquisire attraverso i campi di testo *paramA* e *paramB*, questi parametri vengono direttamente passati alla funzione *Moltiplicazione* indicata nel parametro precedente.

Elaborazione dei dati e grafici

Vettori, matrici ed array di grandi dimensioni sono strumenti fondamentali per l'analisi di dati numerici. Spesso quindi si gestiscono i numeri all'interno di strutture dati in modo che, attraverso l'uso di cicli, è possibile elaborarli e manipolarli in modo da ottenere i risultati voluti.

Quando la mole di dati è grande invece questo tipo di operazione non è ottimale, perché l'uso di cicli su strutture dati di lunghezza variabile ed eterogenei tra loro, rende le operazioni onerose in termini di tempo e di spazio.

NumPy

Negli ambienti di sviluppo il calcolo scientifico basato su Python vengono utilizzate e strutture dati più efficienti di quelli forniti dal linguaggio base, si utilizza infatti principalmente la libreria NumPy.

Apparentemente le strutture dati sembrano simili a semplici liste, cioè generici contenitori di oggetti. In realtà, invece, gli array appartenenti a NumPy, sono array di tipo omogeneo (tutti i dati di un array sono dello stesso tipo) e di dimensione fissa, quindi non modificabile dopo la creazione. Queste due caratteristiche rendono molto più efficienti le operazioni applicate su strutture di questo tipo.

NumPy, inoltre, include una vasta collezione di operazioni base e funzioni dedicate alle strutture dati come algoritmi di alto livello relativi all'algebra lineare e a trasformazioni più complesse.

Numpy offre, inoltre, un backend numerico per quasi tutte le librerie scientifiche e tecniche dell'ecosistema Python. Per questi argomenti più specifici è più opportuno consultare il sito ufficiale: <http://www.numpy.org>.

NumPy Array Object

Come abbiamo detto, la libreria *NumPy* riguarda sostanzialmente le strutture di dati per rappresentare array multidimensionali di dati omogenei. La principale struttura di dati presenti in questa libreria si chiama **ndarray**, che oltre ai dati conservati nella struttura contiene importanti metadati relativi alla forma, la dimensione, la tipologia ed altri attributi. Per conoscere tutti gli attributi associati ai dati lo si può fare attraverso il comando **help(np.ndarray)** in python oppure **np.array?** nella console *IPython*. Nella seguente tabella sono rappresentati alcuni esempi di attributi.

Attributo	Descrizione
Shape	Una tupla che contiene il numero di elementi per ogni dimensione (assi) dell'array.
Size	Il numero totale degli elementi dell'array
Ndim	Il numero delle dimensioni (assi)
nbytes	Il numero di Bytes usati per salvare i dati
dtype	Il tipo di dato degli elementi dell'array

L'esempio che segue, invece, permette di comprendere come accedere agli attributi di un Array.

```
data = np.array( [ [1,2] , [3,4] , [ 5,6 ] ] )

type(data)
>> <class 'numpy.ndarray'>

data
>> array ([ [ 1,2] ,
            [ 3,4] ,
            [ 5,6 ] ] )

data.ndim
>> 2
```

```
data.shape
>> (3, 2)

data.size
>> 6

data.dtype
>> dtype('int64')

data.nbytes
>> 48
```

Quando si crea un Array numpy, non è più possibile modificare il dtype, è possibile invece crearne una copia con un operazione di casting.

Ci sono diversi modi per generare un Array, dipende dalla sua struttura iniziale e soprattutto dall'uso che se ne deve fare. Per questo motivo il modulo mette a disposizione una grande varietà di metodi utili a generare Array di questo tipo.

Matplotlib

Nel mondo scientifico, come in tanti altri campi, è sempre utile rappresentare i dati sotto forma di grafici, Per avere rappresentazioni sia in 2D che in 3D di grandi quantità di numeri. Per ottenere queste rappresentazioni, è possibile procedere principalmente attraverso due metodologie: elaborare separatamente i grafici dopo aver acquisito i dati, oppure generare grafici in modo automatico a partire dai dati. In Python è possibile elaborare in modo automatico i grafici e la più popolare e generica libreria di generazione dei grafici è **Matplotlib** (www.matplotlib.org).

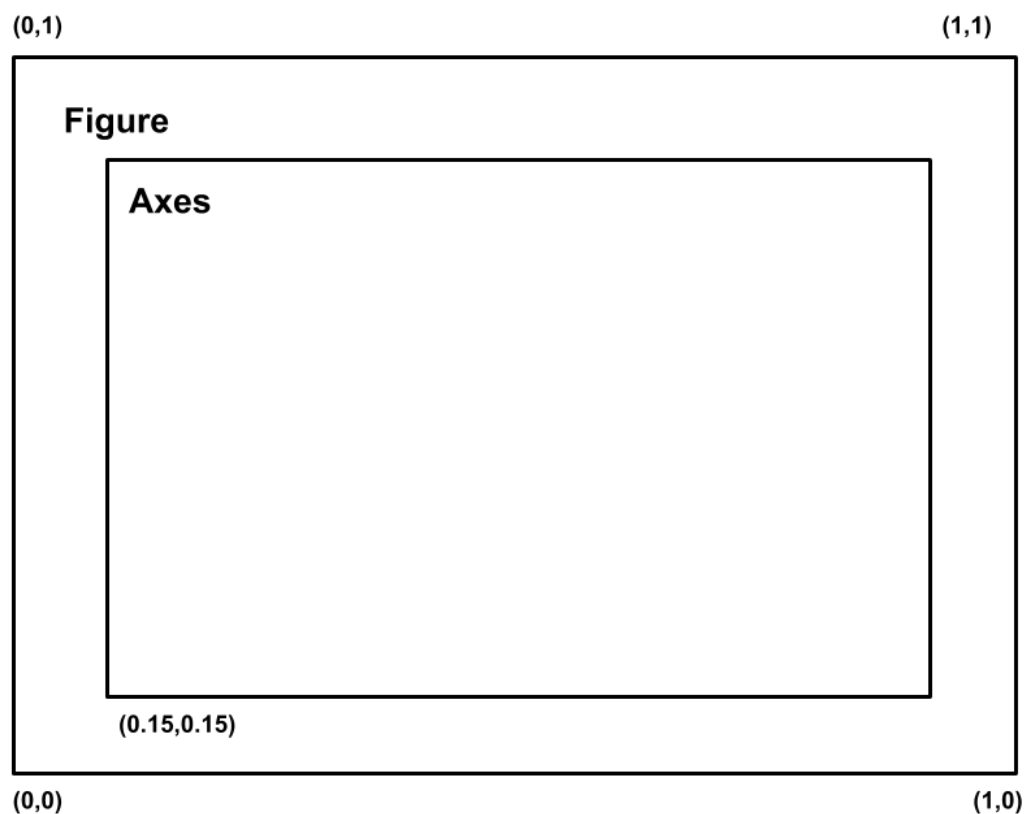
Esistono diversi modi per utilizzare *Matplotlib*, negli esempi che seguiranno verrà importata esattamente come gli altri moduli, così da poterla utilizzare in *script* generici interagendo anche con altri moduli e librerie.

Bisogna Inoltre sottolineare che questa libreria non contiene i soltanto funzioni per generare grafici, ma continui anche supporti per la visualizzazione i grafici in differenti ambienti, quindi la loro esportazione in diversi formati di file come png, pdf, swg.

Quando bisogna richiamare gli strumenti software di backend, per creare ad esempio una finestra in cui rappresentare il grafico, è necessario richiamare la funzione **plt.show()**, così da creare una finestra che alla chiusura terminerà lo script che l'ha generata (https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.show.html).

Struttura dei grafici

Un grafico, in *Matplotlib*, è strutturato come un'istanza chiamata **Figure**, ed uno o più Istanze di **Axes** all'interno della figura. Il concetto di istanza deriva dal fatto che la figura e gli assi sono *oggetti* che racchiudono in sé caratteristiche e metodi, detti anche funzioni, che permettono la loro manipolazione. *Figure* racchiude un'area detta **Canvas** che rappresenta lo spazio di lavoro, le istanze *Axes* forniscono un sistema di coordinate (utili ai grafici) che sono assegnate a regioni fisse dell'intera area di lavoro.

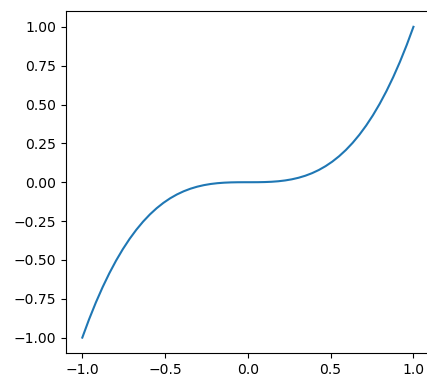


Nella figura precedente sono rappresentate le due istanze, *Figure* determina un'area di lavoro e *Axes* determina lo spazio all'interno del quale verranno collocati gli assi cartesiani. Il sistema di coordinate che prevede la coppia $(0,0)$ in basso a sinistra e la coppia $(1,1)$ in alto a destra permette la collocazione degli *Axes* e quindi dei grafici. Queste coordinate vengono utilizzate soltanto quando si colloca un elemento nell'area di disegno.

Se sono presenti più grafici (quindi più istanze *Axes*) all'interno della stessa area di disegno, possono essere collocati in modo arbitrario all'interno dell'area attraverso le coordinate, oppure si può procedere in modo automatico utilizzando gli strumenti messi a disposizione del modulo.

Si osservi il seguente esempio molto semplice.

```
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(-1.0,1.0,50,endpoint=True)
y = x**3
plt.plot(x,y)
plt.show()
```



Si osservi che:

- La dimensione degli assi combacia.
- I **tick mark** (si chiamano così i punti evidenziati degli assi) degli assi sono tutti distanziati di 0.5 unità.
- Non c'è titolo e non ci sono etichette sugli assi.
- Non c'è una legenda.
- Il colore della linea del grafico è blu.

Queste sono le impostazioni di default, si vedrà in seguito come personalizzarle. Si osservi invece l'esempio che segue, più articolato: due equazioni inserite nello stesso grafico.

```
## questo esempio è presente in http://github.com/doceo/labPy/matplotlib
# grafico-funzioni-1.py

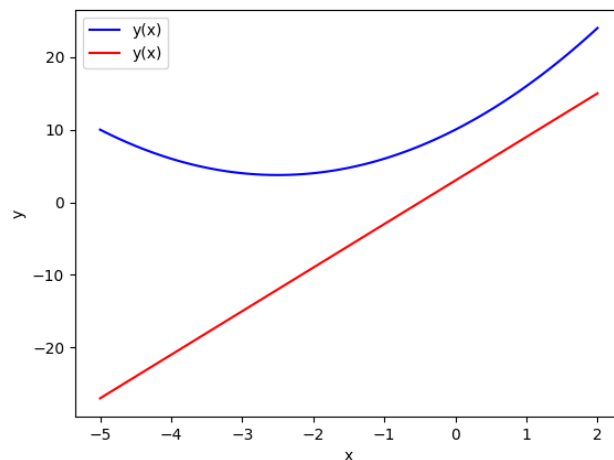
import matplotlib.pyplot as plt
```

```
import numpy as np

# https://numpy.org/doc/stable/reference/generated/numpy.linspace.html
x = np.linspace(-5,2,100)

y1 = x**2 + 5*x +10
y2 = 6*x + 3

fig, ax = plt.subplots()
ax.plot(x,y1,color="blue", label="y(x) ")
ax.plot(x,y2,color="red", label="y(x) ")
ax.set_xlabel("x")
ax.set_ylabel("y")
ax.legend()
plt.show()
```



Le variabili $y1$ e $y2$, si noti, sono espressioni della variabili x , che a sua volta è un oggetto di tipo *numPy*. La variabile x è un array di tipo *np* generato dal metodo **linspace()**, che restituisce numeri equidistanti su un intervallo specificato (nel caso specifico sono 100 punti compresi tra -5 e 2). Successivamente viene usata la funzione **plt.subplots()** per generare l'istanza di Figure e di Axes. Questa funzione quindi risulta conveniente perché attraverso di essa possiamo generare contemporaneamente l'area di lavoro e le istanze dei grafici. A questo punto sarà necessario utilizzare i metodi relativi ad Axes, così da creare i grafici delle equazioni scritte nelle prime righe di codice. Per fare ciò è stato utilizzato il metodo **ax.plot()**, che prende come primo e secondo argomento array di tipo *numPy*, quindi i dati relativi ai valori X e Y dei punti dei grafici, *subplot* unisce tutti questi punti in forma di linea all'interno del grafico. Inoltre è stato usato il parametro **color** come argomento opzionale per specificare il colore della linea e il parametro **label** per indicare i valori nella legenda. Per indicare invece la legenda relativa agli assi, quindi per

distinguere l'asse delle x dall'asse delle y, Sono stati introdotti altri due metodi: **set_xlabel("x")** e **set_ylabel("y")**. Come è possibile comprendere dal codice, sia i parametri all'interno della funzione *plot* che i parametri di input per *set_xlabel("x")* sono di tipo stringa. A conclusione dello script, *plt.show()* genererà la finestra con i grafici richiesti.

È possibile immaginare, quindi, che esista un'ampia gamma di parametri che permettono una completa personalizzazione dei grafici realizzabili.

Figure

Come abbiamo detto l'oggetto *Figure* è necessario per rappresentare un grafico, perchè genera uno spazio di lavoro destinato a contenere gli oggetti di tipo *Axis*. *Figure* possiede molte proprietà e metodi per ottimizzare questo processo, in particolare è possibile fornire le dimensioni precise dello spazio di lavoro attraverso l'attributo **figsize**, assegnandogli tuple intese come coppia (width, height), l'unità di misura è il pollice (non il centimetro). Spesso, inoltre, si determina il colore dello spazio di lavoro attraverso l'attributo **facecolor**.

Una volta creato lo spazio di lavoro bisogna usare il metodo **add_axes** per creare una nuova istanza *Axis* e da collocare in una posizione all'interno dello spazio di lavoro. *add_axes* prende in input come argomenti una lista contenente le coordinate che, partendo dall'angolo in basso a sinistra per arrivare a quello in alto a destra, determinano la posizione dell'oggetto *Axis*.

Ad esempio, se si assegna la lista (0, 0, 1, 1) all'oggetto *Axis*, verrebbe occupata l'intera area di lavoro, senza lasciare spazio alla legenda o ad altri elementi utili. Per tale motivo è buona norma utilizzare una lista diversa, ad esempio (0.1, 0.1, 0.8, 0.8). In questo modo l'oggetto occuperebbe il centro dello spazio di lavoro corrispondente a una copertura del 80%.

Dopo aver creato le istanze *Figure* e *Axis* e collocato *Axis* dentro la *Figure*, attraverso *add_axes*, è possibile rappresentare i dati utilizzando i metodi messi a disposizione dall'oggetto *Axis*. Prima di esplorare le possibilità relative all'oggetto *Axis*, è bene sottolineare le tante possibilità che *Figure* mette a disposizione per la creazione, la manipolazione e la stampa di grafici, una volta personalizzati attraverso i metodi di *Axis*. Ad esempio:

Metodo	Descrizione	Esempio
suptitle()	Inserisce un titolo al centro dell'area di lavoro, Il titolo in sito come stringa nel primo parametro di input	<pre>fig.suptitle('This is the figure title', fontsize=12)</pre>

savefig()	<p>Salva il grafico in un file, il nome del file deve essere passato come primo parametro (<i>fname</i> nell'esempio). Esistono molti parametri da poter passare a questo metodo, anche se non sono obbligatori. L'estensione del file, di default, è determinata dal formato indicato nella stringa passato come primo parametro, è possibile comunque personalizzare l'estensione del file attraverso il parametro <i>format</i>. (png, pdf, eps ed svg sono tutte estensioni accettate. L'argomento dpi, invece, definisce la risoluzione. Nunzia ti viene sotto definisco la comédie su Subito plt Shih Tzu nice to be happy with The quality of life And finally the figures che non serviva l'apparecchio si trasforma in Show ARP table fixed column for the listing Please tecniche del salto in su</p>	<pre>savefig(fname, dpi=None, facecolor='w', edgecolor='w', orientation='portrait', papertype=None, format=None, transparent=False, bbox_inches=None, pad_inches=0.1, frameon=None, metadata=None)</pre>
------------------	---	--

Tutti i metodi messi a disposizione dall'oggetto *Figure*, sono ben descritti nella documentazione di riferimento:

https://matplotlib.org/3.1.1/api/figure_api.html?highlight=figure#module-matplotlib.figure

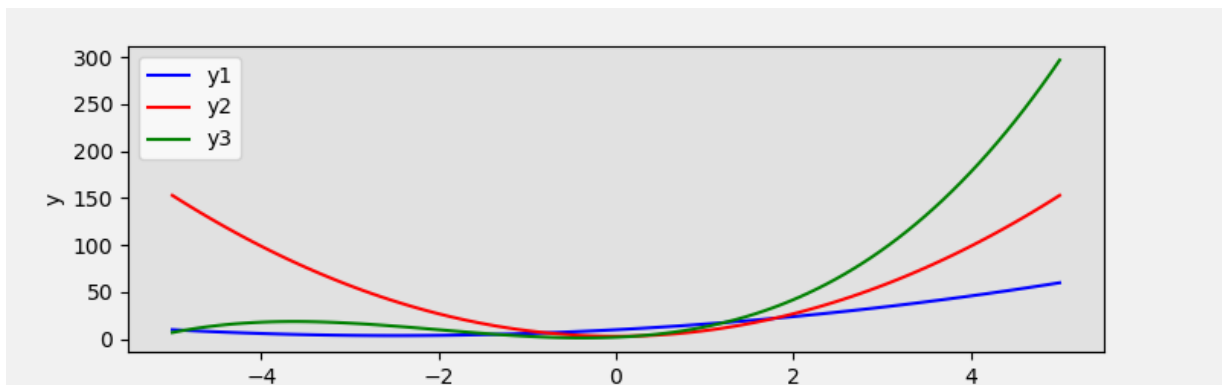
Un esempio di utilizzo di parametri è disponibile attraverso il seguente codice:

figure-esempio.py

```
## esempio riportato sugli appunti condivisi

import matplotlib.pyplot as plt
import numpy as np
```

```
# i colori vanno passati come stringa,  
# il colore è ricavato dalla codifica esadecimale RGB  
fig = plt.figure(figsize=(8,2.5), facecolor="#f5f5f5")  
  
# la posizione di Axes la determiniamo come distanza dai bordi  
left, bottom, width, height = 0.1, 0.1, 0.8, 0.8  
ax = fig.add_axes((left, bottom, width, height), facecolor="#e1e1e1")  
  
x = np.linspace(-5,5,100)  
  
y1 = x**2 + 5*x + 10  
y2 = 6*x**2 + 3  
y3 = x**3 + 6*x**2 + 4*x + 2  
  
ax.plot(x,y1,color="blue", label="y1")  
ax.plot(x,y2,color="red", label="y2")  
ax.plot(x,y3,color="green", label="y3")  
ax.set_xlabel("x")  
ax.set_ylabel("y")  
ax.legend()  
  
# salva il grafico in un file di nome grafico.png  
fig.savefig("grafico.png", dpi=100, facecolor="#f1f1f1")  
plt.show()
```



Axes

Le istanze degli oggetti *Axes*, come è stato introdotto precedentemente, sono collocati all'interno delle istanze di oggetti *Figure*. Gli *Axes*, dunque, sono il cuore del modulo *Matplotlib*, perché attraverso di essi è possibile manipolare i grafici che rappresentano l'elaborazione numerica e, sempre attraverso questi oggetti, è possibile scegliere il tipo di grafico per la rappresentazione da realizzare.

È stato utilizzato più volte, negli esempi precedenti, il metodo `add_axes`, sottolineando come questo metodo fosse particolarmente duttile nel posizionare sia manualmente che automaticamente i grafici all'interno dell'area di lavoro. Questo tipo di necessità emerge soprattutto quando bisogna rappresentare più grafici in una griglia, all'interno di una sola istanza *Figure*. Esistono diverse modalità per gestire rappresentazioni complesse, ma per motivi di semplicità verrà descritta solo quella relativa alla funzione `plt.subplots()`.

Per impostare una griglia di grafici bisogna passare a questa funzione due parametri: **nrows** e **ncols**. Questi due valori permetteranno a *subplot* di generare una griglia di *nrows* righe per *ncols* colonne.

```
fig, axes = plt.subplots(nrows=3, ncols = 3)
```

La funzione `subplot` restituirà una tupla (**fig, axes**), dove *fig* è una istanza *Figure* e *axes* è un array *NumPy* di dimensione (nrows, ncols, che rappresentano righe e colonne), i cui elementi sono istanze di tipo *Axes*, da collocare nell'area di lavoro.

È possibile concludere quindi che il metodo `subplots` permette di gestire più grafici nella stessa area di lavoro restituendo le istanze necessarie sia relative a *Figure* che ad *Axes*. Si noti infatti, nell'esempio che segue, che i grafici vengono inseriti nella matrice chiamata *ax*, così da permettere una gestione più semplice di dati comuni a tutti i grafici. Nell'esempio, infatti, viene applicato un ciclo di `for` per indicare a tutti i grafici le etichette relative agli assi cartesiani.

subplot-griglia.py

```
import matplotlib.pyplot as plt
import numpy as np

# indichiamo una matrice di una riga per 4 colonne
```

```
fig, ax = plt.subplots(1, 4, figsize=(14,3))

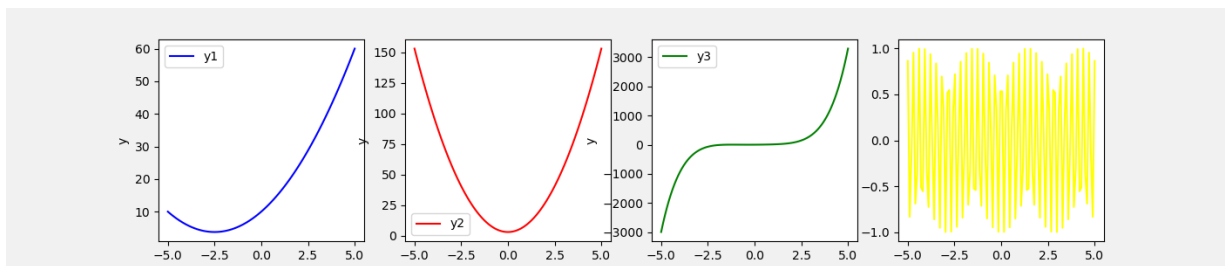
# la posizione di Axes la determiniamo come distanza dai bordi
left, bottom, width, height = 0.1, 0.1, 0.8, 0.8
x = np.linspace(-5,5,100)

y1 = x**2 + 5*x +10
y2 = 6*x**2 + 3
y3 = x**5 + 6*x**2 + 4*x+2
y4 = np.cos(20*x)

ax[0].plot(x,y1,color="blue", label="y1")
ax[1].plot(x,y2,color="red", label="y2")
ax[2].plot(x,y3,color="green", label="y3")
ax[3].plot(x,y4,color="yellow", label="y4")

for i in range(3):
    ax[i].set_xlabel("x")
    ax[i].set_ylabel("y")
    ax[i].legend()

# salva il grafico in un file di nome grafico.png
fig.savefig("grafico.png", dpi=100, facecolor="#f1f1f1")
plt.show()
```



Osservando l'esempio precedente emerge l'utilizzo del metodo **plot()** per assegnare all'oggetto *Axes*, appartenenti alla matrice *ax*, il grafico da rappresentare. La forma del diagramma è in funzione degli input che riceve, se ad esempio diamo un solo parametro ingresso lui costruirà il grafico in cui sull'asse Y ci saranno gli elementi del vettore dato in ingresso e sull'asse X ci

saranno gli indici di tale vettore. Se invece diamo in ingresso a *plot* due vettori, lui inserirà sull'asse X il primo vettore e sull'asse Y il secondo, così da generare i punti del diagramma e di conseguenza la sua linea.

Personalizzare i grafici

Il metodo *plot*, mette a disposizione molti parametri per gestire i grafici e personalizzare la visualizzazione, nella tabella che segue sono elencati solo alcuni di questi.

Argomento	Esempio	Descrizione
color	va determinato attraverso una stringa identificativa del colore: "red", "blue", oppure un colore RGB in formato esadecimale (#f3f3f3)	colore della linea
alpha	indica la trasparenza con un numero reale compreso tra 0.0 (totalmente trasparente) ed 1.0 (totalmente opaco)	trasparenza
linewidth, lw	numero reale	spessore della linea
linestyle, ls	"-" linea continua "--" linea tratteggiata "." linea punteggiata "-." alternanza di tratto e punto	lo stile della linea del grafico
marker	+,o,* = croce, cerchi e stelle s = quadrato . = punto piccolo 1,2,3,4..= forme triangolari	ogni punto può essere rappresentato con un simbolo
markersize	numero reale	la grandezza del punto (marker)
markerfacecolor	valori identificativi di colori	il colore del punto (marker)
markeredgewidth	numero reale	larghezza della linea che definisce il marker
markeredgecolor	valori identificativi dei colori	colore della linea di definizione del marker

Linea

Il grafico di default prevede una semplice linea, è chiaro però che se il numero di funzioni aumenta, al fine di rendere leggibile un grafico, è necessario cambiare non soltanto il colore, ma anche il tipo di linea che può diventare tratteggiata, una sequenza di punti, una linea spezzata oppure cambiare proprio la natura del grafico, rappresentandolo come grafico a barre, a dispersione oppure come aree di superficie colorata.

Attraverso il seguente script è possibile vedere tutte le possibili personalizzazioni grafiche relative alle linee, quindi ad una funzione lineare, indipendentemente da come i punti vengono rappresentati o uniti. Un grafico di questo tipo rappresenta un andamento, ad esempio una progressione dei valori del Y in funzione delle X.

linestyle.py

```
"""
=====
Linestyles
=====

https://matplotlib.org/examples/lines\_bars\_and\_markers/linestyles.html

This examples showcases different linestyles copying those of Tikz/PGF.
"""
import numpy as np
import matplotlib.pyplot as plt
from collections import OrderedDict
from matplotlib.transforms import blended_transform_factory

linestyles = OrderedDict(
    [('solid', (0, ())),
     ('loosely dotted', (0, (1, 10))),
     ('dotted', (0, (1, 5))),
     ('densely dotted', (0, (1, 1))),

     ('loosely dashed', (0, (5, 10)))
```

```

('dashed', (0, (5, 5))),
('densely dashed', (0, (5, 1))),

('loosely dashdotted', (0, (3, 10, 1, 10))),
('dashdotted', (0, (3, 5, 1, 5))),
('densely dashdotted', (0, (3, 1, 1, 1))),

('loosely dashdotdotted', (0, (3, 10, 1, 10, 1, 10))),
('dashdotdotted', (0, (3, 5, 1, 5, 1, 5))),
('densely dashdotdotted', (0, (3, 1, 1, 1, 1, 1)))]

plt.figure(figsize=(10, 6))
ax = plt.subplot(1, 1, 1)

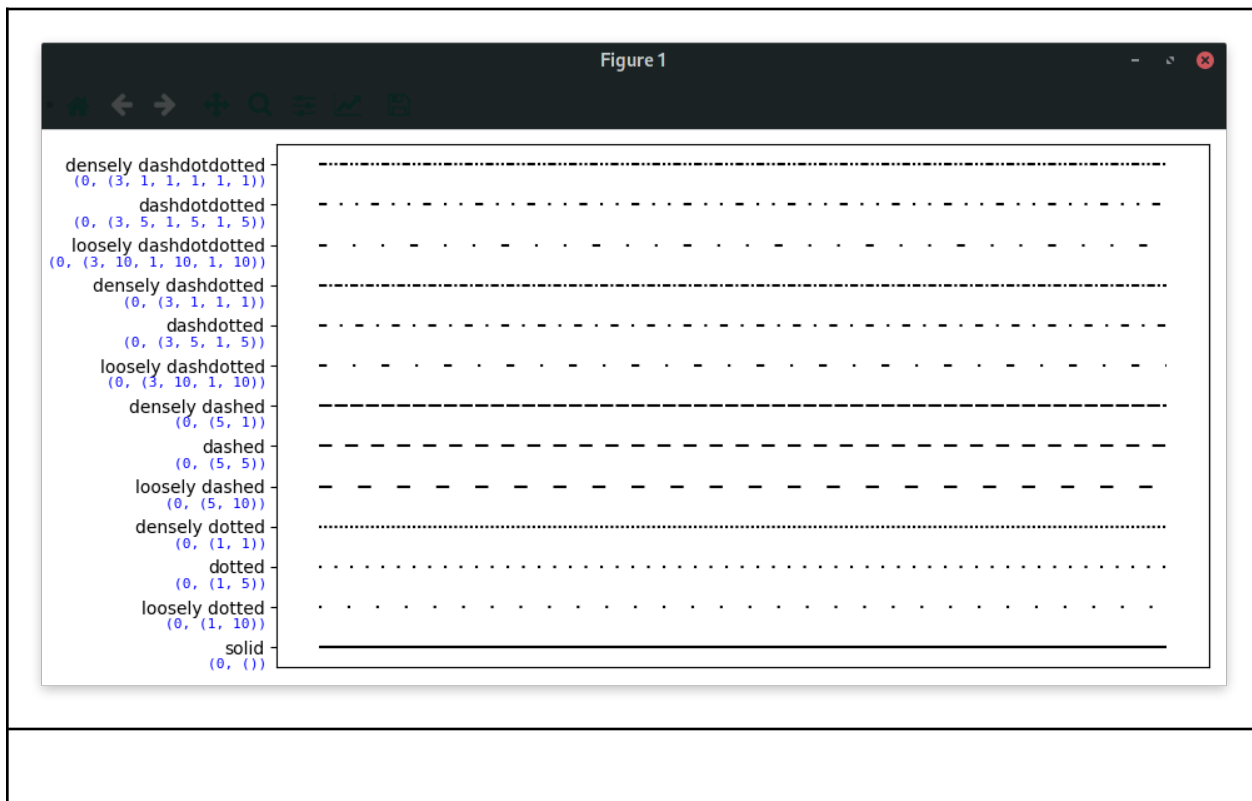
X, Y = np.linspace(0, 100, 10), np.zeros(10)
for i, (name, linestyle) in enumerate(linestyles.items()):
    ax.plot(X, Y+i, linestyle=linestyle, linewidth=1.5, color='black')

ax.set_ylim(-0.5, len(linestyles)-0.5)
plt.yticks(np.arange(len(linestyles)), linestyles.keys())
plt.xticks([])

# For each line style, add a text annotation with a small offset from
# the reference point (0 in Axes coords, y tick value in Data coords).
reference_transform = blended_transform_factory(ax.transAxes,
ax.transData)
for i, (name, linestyle) in enumerate(linestyles.items()):
    ax.annotate(str(linestyle), xy=(0.0, i),
                xycoords=reference_transform,
                xytext=(-6, -12), textcoords='offset points',
                color="blue", fontsize=8, ha="right", family="monospace")

plt.tight_layout()
plt.show()

```



Volendo sintetizzare il ruolo del modulo *pyplot*, potremmo considerarlo una raccolta di funzioni che consentono di utilizzare le funzionalità di *matplotlib*. Ogni funzione di *pyplot* agisce in una singola finestra di *plot*, detta *Figura*, ad esempio:

- crea una figura
- crea un'area di plotting all'interno di una figura
- disegna dei grafici nell'area di plotting
- personalizza il plot con etichette e altri elementi grafici

Bisogna sottolineare, inoltre, che *pyplot* è **stateful**, ovvero tiene traccia dello stato della figura corrente e della relativa area di disegno, le sue funzioni agiscono sulla figura corrente.

Oltre al tipo di linea, ovviamente, è possibile Modificare i colori attraverso il parametro **color**.

I colori ammessi, oltre a tutti quelli rappresentabili sotto forma di codice esadecimale, sono:

- blue
- green
- red
- cyan
- magenta

- bilious yellow
- grey

```
pyplot.plot(x,y, color='red', linestyle='--')
```

Marker, i punti della curva

Quando i dati rappresentati non devono essere uniti da una linea, ma devono essere rappresentati attraverso dei punti definiti, chiamati **Marker**, possiamo rappresentare questi punti in diversi modi, evidenziando così delle differenze.

la funzione *plot*, attraverso i parametri relativi ai *Marker*, indicati nella tabella ad inizio paragrafo, consente di identificare i punti di un andamento, se pur lineare.

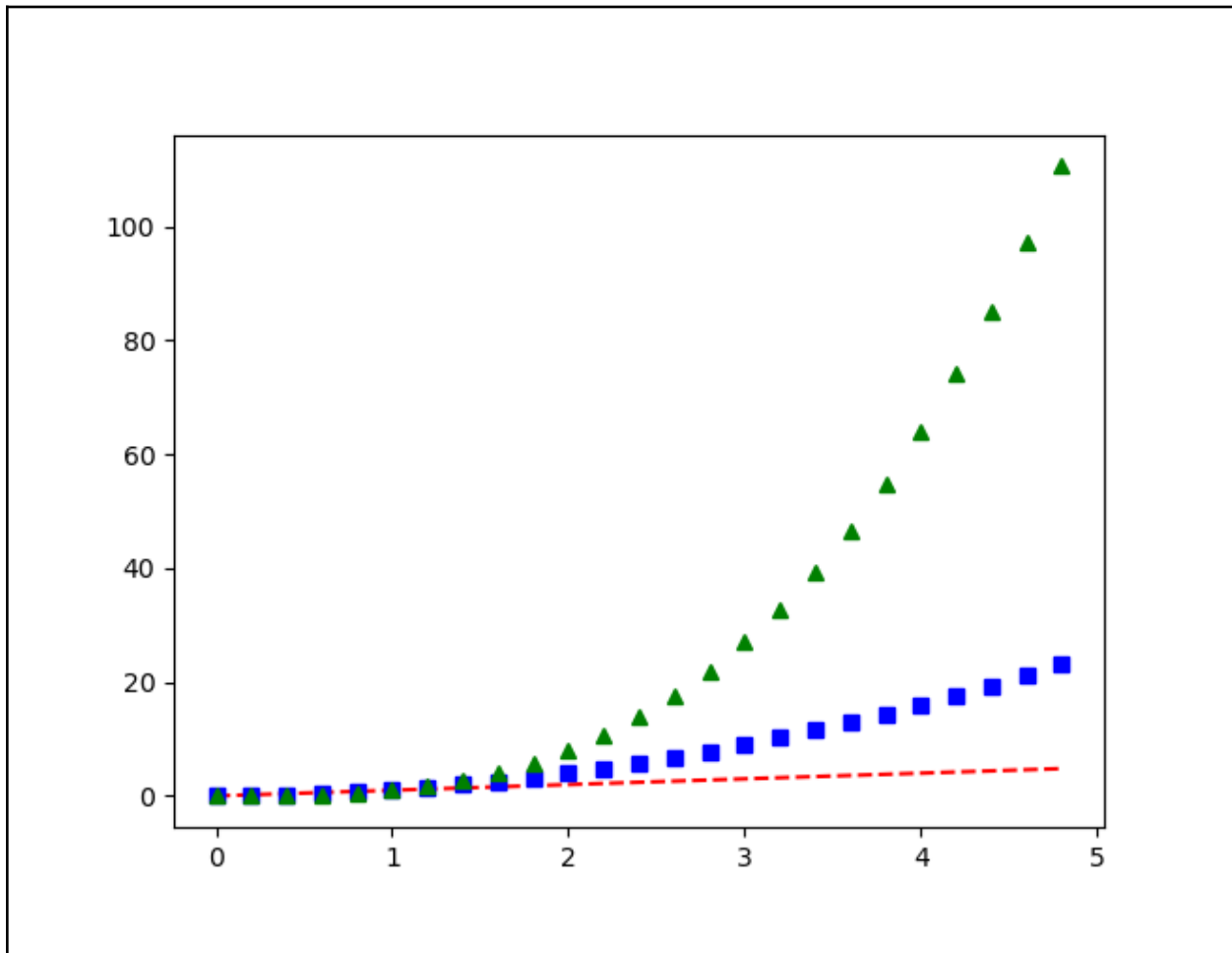
Bisogna sottolineare la differenza, quindi, tra un grafico di tipo *plot*, che rappresenta i valori in modo lineare, dal grafico a dispersione (**scatter**) che non rappresenta i dati da un punto di vista lineare, ma come una nuvola di punti eventualmente da analizzare e manipolare. Il metodo **Scatter** verrà trattato successivamente.

plot-marker.py

```
import numpy as np
import matplotlib.pyplot as plt

# i punti saranno rappresentati a distanza di 0,2 tutti i numeri
# compresi tra 0 e 5
t = np.arange(0., 5., 0.2)

# trattini rossi, blue quadrati e triangoli verdi
plt.plot(t, t, 'r--', t, t**2, 'bs', t, t**3, 'g^')
plt.show()
```



Da notare nell'esempio precedente che i punti sono definiti con un array di tipo *numpy* e che vengono passati alle tre funzioni dichiarate all'interno della funzione `plot`. Infatti:

```
plt.plot(t, t, 'r--', t, t**2, 'bs', t, t**3, 'g^')
```

prende in input 9 parametri:

1. `t`, relativo alle X della prima funzione
2. `t`, le Y della prima funzione, lineare rispetto alle X
3. `'r--'` linea tratteggiata relativa alla prima funzione
4. `t`, relativo alle X della seconda funzione
5. `t**2`, le Y della seconda funzione, quadratica rispetto alle X
6. `'bs'` quadrati per rappresentare i punti relativi alla seconda funzione
7. `t`, relativo alle X della terza funzione
8. `t**3`, le Y della terza funzione, cubica rispetto alle X

9. 'g^' triangoli per rappresentare i punti relativi alla terza funzione

Passando tutte e tre le funzioni nello stesso *plot* avremo rappresentato le tre curve nello stesso oggetto *Axes*, così da rappresentarle insieme. Per le descrizioni di tutte le rappresentazioni possibili bisogna consultare il manuale relativo a *plot*: https://matplotlib.org/3.3.3/api/markers_api.html

Assi, proprietà e personalizzazioni

Gli oggetti *Figure* ed *Axes*, è stato più volte detto, rappresentano rispettivamente il piano di lavoro e il grafico al suo interno (o più grafici rappresentati da più oggetti *Axes*). All'interno del grafico, però, ci sono almeno due assi per grafici bidimensionali e 3 assi per quelli a 3 dimensioni.

Ogni asse potrà quindi essere personalizzato attraverso apposite funzioni, alcune delle quali affrontate negli esempi precedenti, ad esempio *set_xlabel*, *set_ylabel* e per la definizione delle etichette degli assi X e Y. A questi metodi già affrontati è possibile aggiungere quelli relativi al **Range** degli assi cartesiani, come nell'esempio che segue.

assi-range.py

```
import matplotlib as mpl
import numpy as np
import matplotlib.pyplot as plt

# definisco le x della curva
x = np.linspace(0,30,500)

# definisco le y della curva
y = np.sin(x) * np.exp(-x/10)

fig, axes = plt.subplots(1,3,figsize=(9,3),
subplot_kw={'facecolor':'#ebf5ff'})

axes[0].plot(x,y,lw=2)
axes[0].set_xlim(-5,35)
axes[0].set_ylim(-1,1)
axes[0].set_title(" valori definiti")
```

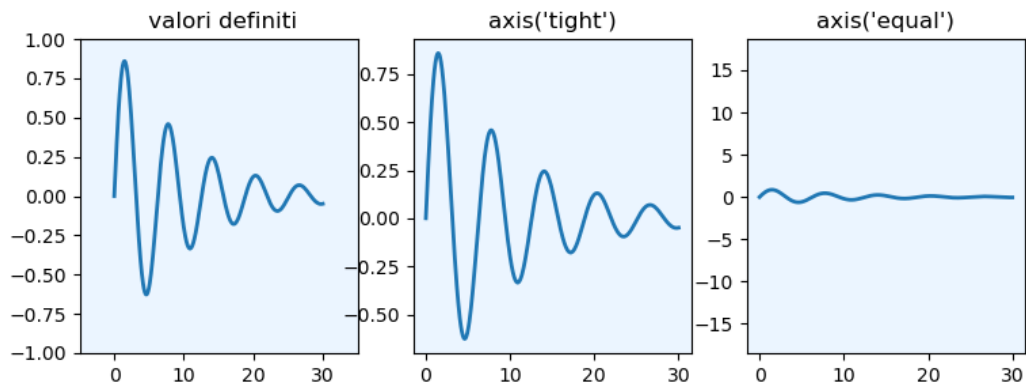
```

axes[1].plot(x,y,lw=2)
axes[1].axis('tight')
axes[1].set_title("axis('tight')")

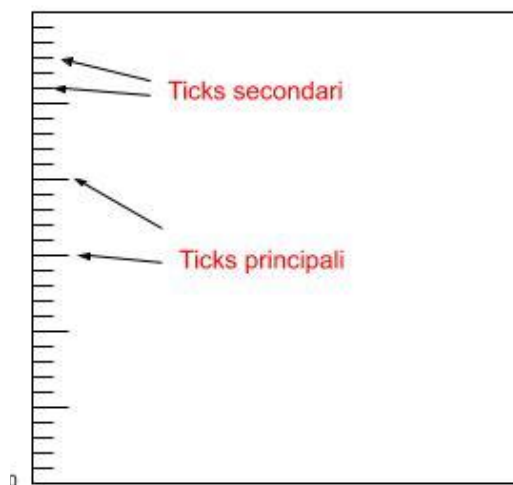
axes[2].plot(x,y,lw=2)
axes[2].axis('equal')
axes[2].set_title("axis('equal')")

plt.show()

```



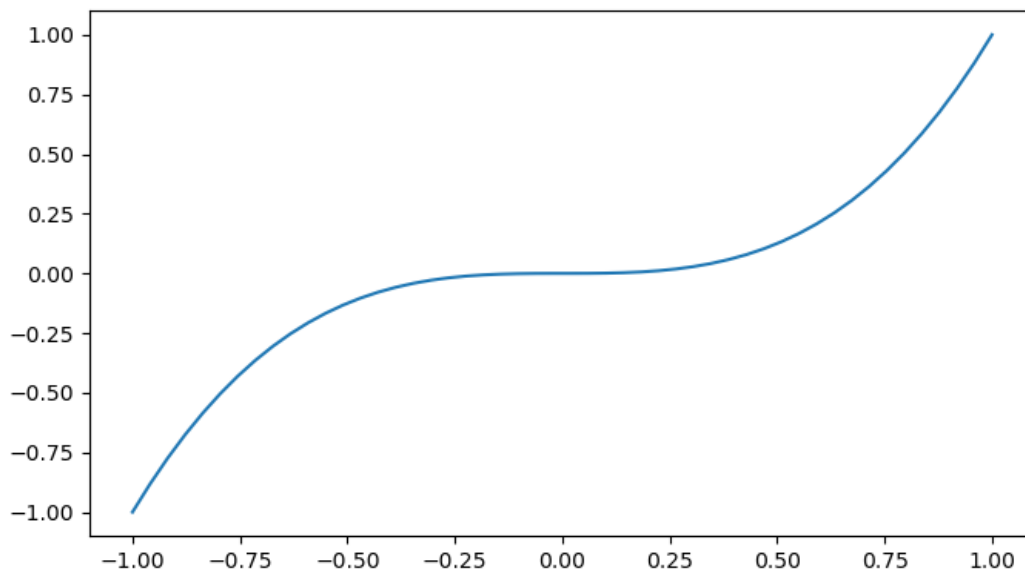
I punti identificati sugli assi sono i **Ticks**, che possiamo distinguere in principali e secondari che sono definibili attraverso le funzioni **set_major_locator** e **set_minor_locator**.



La funzione **pyplot.xticks()** (o **.yticks()**), inoltre, permette di modificare il comportamento di *pyplot* relativo ai *tick mark* dell'asse X e Y. I tick mark sono i punti evidenziati sulle due assi.

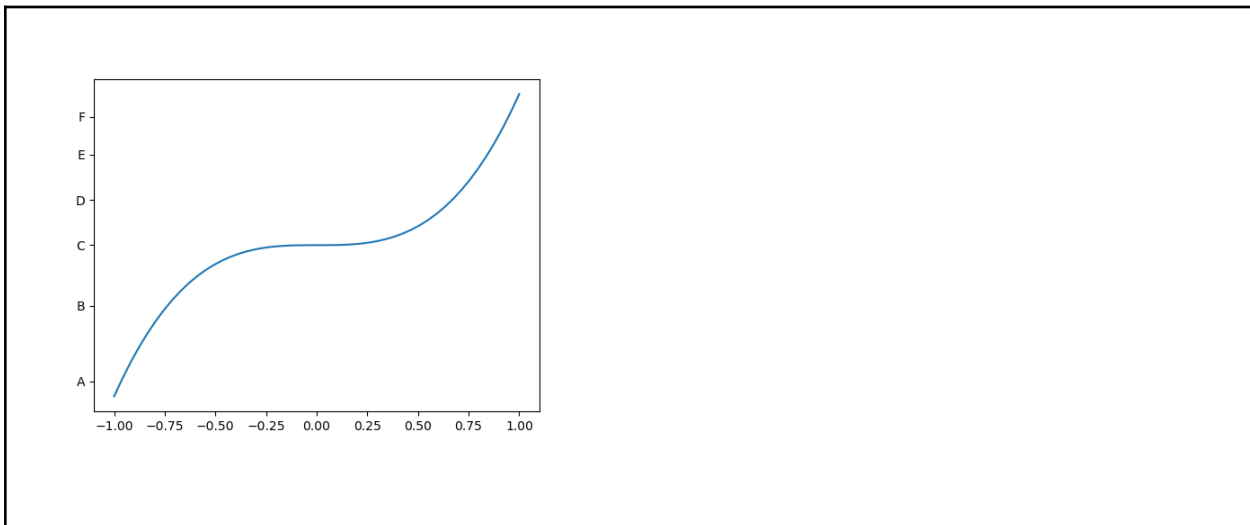
Applicando questi valori di ingresso alla funzione si otterrebbe il grafico in figura, con i *tick mark* distanziati di 0.25.

```
plt.xticks([0.25*k for k in range(-4,5)])  
plt.yticks([0.25*k for k in range(-4,5)])
```



L'argomento da passare a questa funzione, comunque può anche non essere una lista equispaziata, ma una lista di punti scelti in relazione dei dati da rappresentare. Sarà possibile, inoltre, passare anche una etichetta da associare ai singoli tick del relativo asse.

```
import numpy as np  
import matplotlib.pyplot as plt  
x = np.linspace(-1.0, 1.0, 50, endpoint=True)  
y = x**3  
plt.plot(x, y)  
plt.yticks([-0.9, -0.4, 0.0, 0.3, 0.6, 0.85], ['A', 'B', 'C', 'D', 'E', 'F'])  
  
plt.show()
```



Legenda

Un grafico con più linee necessita sicuramente di una leggenda che spieghi con un'etichetta le diverse curve rappresentate. È possibile dotare ogni istanza *Axes* di una leggenda attraverso il metodo **legend**, così da includere la descrizione di ogni curva che viene inclusa a partire dall'argomento *label* passato alla funzione *Axes.plot*. Il metodo *legend* permette di agire su moltissimi parametri, per i dettagli si consiglia di accedere all'help (`help(plt.legend)`).

```
ax.legend(loc="lower right", title="Legend Title", frameon=False)
```

Sarà possibile inoltre condividere un'unica leggenda per tutti i grafici, quindi per tutte le istanze *Axes* del piano di lavoro, come nell'esempio che segue.

legenda-condivisa.py

```
import numpy as np
import matplotlib.pyplot as plt
fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(10,4))
fig.suptitle('Example of a Single Legend Shared Across Multiple
Subplots')

# valori
```

```
x = [1, 2, 3]
y1 = [1, 2, 3]
y2 = [3, 1, 3]
y3 = [1, 3, 1]
y4 = [2, 2, 3]

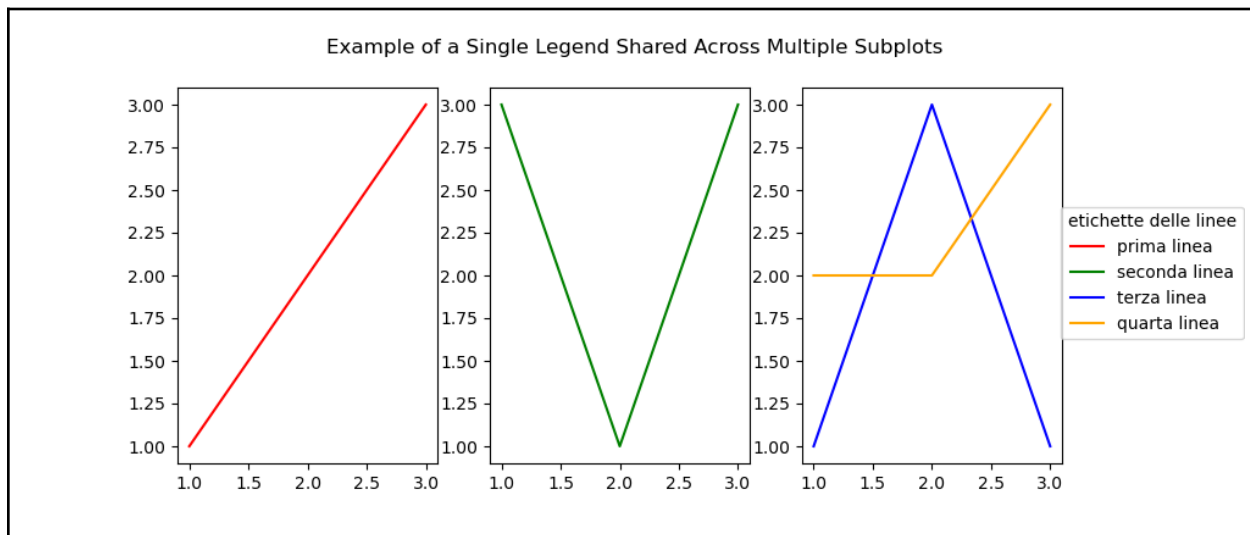
# etichette delle linee
line_labels = ["prima linea", "seconda linea", "terza linea", "quarta
linea"]

# creo le istanze dei grafici
l1 = ax1.plot(x, y1, color="red")[0]
l2 = ax2.plot(x, y2, color="green")[0]
l3 = ax3.plot(x, y3, color="blue")[0]
l4 = ax3.plot(x, y4, color="orange")[0] # A second line in the third
subplot

# Creo la legenda
fig.legend([l1, l2, l3, l4],      # associo i grafici
           labels=line_labels,    # associo le etichette
           loc="center right",    # posiziono la legenda
           borderaxespad=0.1,    # la distanza dai bordi
           title="etichette delle linee" # assegno un titolo
          )

# posiziono la legenda all'interno dell'area di lavoro
plt.subplots_adjust(right=0.85)

plt.show()
```



Di default gli argomenti di una legenda sono disposti in un'unica colonna, ma è possibile personalizzare la descrizione in più colonne. Un'altra personalizzazione molto spesso necessaria è la posizione della leggenda stessa, è possibile, infatti, attraverso il parametro **loc**, come nell'esempio precedente, posizionarla nei quattro angoli; *loc=1* indica l'angolo in alto a destra, *loc = 2* indica l'angolo in alto a sinistra e così via seguendo un ordine antiorario.

Griglia e sfondo

Il colore dell'area di lavoro è stato affrontato nei paragrafi precedenti, ma può essere considerato anche in relazione ad una eventuale griglia come sfondo del grafico. Questa caratteristica è introdotta dal metodo **grid** di Axes.

Nell'esempio che segue vengono prima di tutto definiti i *Ticks* da evidenziare, tra minori e maggiori, successivamente vengono associati al singolo grafico ed in ultimo ogni istanza di Axes viene personalizzata per definire il tipo di linea da realizzare, per i *major* e per i *minor* di ogni asse.

grid.py

```
import matplotlib as mpl
import numpy as np
import matplotlib.pyplot as plt
```



```
# definisco le x della curva
x = np.linspace(0,30,500)

# definisco le y della curva
y = np.sin(x) * np.exp(-x/10)

fig, axes = plt.subplots(1,3,figsize=(9,3))

# definisco quali ticks devo evidenziare come minor e come major e li
salvo nelle relative variabili
x_major_ticker = mpl.ticker.MultipleLocator(4)
x_minor_ticker = mpl.ticker.MultipleLocator(1)
y_major_ticker = mpl.ticker.MultipleLocator(0.5)
y_minor_ticker = mpl.ticker.MultipleLocator(.25)

axes[0].plot(x,y,lw=2)
axes[0].set_xlim(-5,35)
axes[0].set_ylim(-1,1)
axes[0].set_title(" valori definiti")

#associo al grafico i minor e major ticker
axes[0].xaxis.set_major_locator(x_major_ticker)
axes[0].xaxis.set_minor_locator(x_minor_ticker)
axes[0].yaxis.set_major_locator(y_major_ticker)
axes[0].yaxis.set_minor_locator(y_minor_ticker)

# definisco una griglia per il primo grafico, che identifica i major
ticks
axes[0].grid(color="grey", which="major", linestyle=':', linewidth=0.5)

axes[1].plot(x,y,lw=2)
axes[1].axis('tight')
axes[1].set_title("axis('tight')")

#associo al grafico i minor e major ticker
axes[1].xaxis.set_major_locator(x_major_ticker)
```

```
axes[1].xaxis.set_minor_locator(x_minor_ticker)
axes[1].yaxis.set_major_locator(y_major_ticker)
axes[1].yaxis.set_minor_locator(y_minor_ticker)

# definisco una griglia per il secondo grafico, che identifica i minor
ticks
axes[1].grid(color="grey", which="minor", linestyle=':', linewidth=0.25)

axes[2].plot(x,y,lw=2)
axes[2].axis('tight')
axes[2].set_title("axis('tight')")

#associo al grafico i minor e major ticker
axes[2].xaxis.set_major_locator(x_major_ticker)
axes[2].xaxis.set_minor_locator(x_minor_ticker)
axes[2].yaxis.set_major_locator(y_major_ticker)
axes[2].yaxis.set_minor_locator(y_minor_ticker)

# definisco una griglia per il terzo grafico, che identifica i major
ticks sull'asse y
# e i minor ticks sull'asse x
axes[2].grid(color="grey", which="minor", axis='y', linestyle=':',
linewidth=0.5)
axes[2].grid(color="grey", which="major", axis='y', linestyle='-',
linewidth=0.5)
axes[2].grid(color="grey", which="minor", axis='x', linestyle=':',
linewidth=0.5)

plt.show()
import matplotlib as mpl
import numpy as np
import matplotlib.pyplot as plt

# definisco le x della curva
x = np.linspace(0,30,500)
```

```
# definisco le y della curva
y = np.sin(x) * np.exp(-x/10)

fig, axes = plt.subplots(1,3,figsize=(9,3))

# definisco quali ticks devo evidenziare come minor e come major e li
salvo nelle relative variabili
x_major_ticker = mpl.ticker.MultipleLocator(4)
x_minor_ticker = mpl.ticker.MultipleLocator(1)
y_major_ticker = mpl.ticker.MultipleLocator(0.5)
y_minor_ticker = mpl.ticker.MultipleLocator(.25)

axes[0].plot(x,y,lw=2)
axes[0].set_xlim(-5,35)
axes[0].set_ylim(-1,1)
axes[0].set_title(" valori definiti")

#associo al grafico i minor e major ticker
axes[0].xaxis.set_major_locator(x_major_ticker)
axes[0].xaxis.set_minor_locator(x_minor_ticker)
axes[0].yaxis.set_major_locator(y_major_ticker)
axes[0].yaxis.set_minor_locator(y_minor_ticker)

# definisco una griglia per il primo grafico, che identifica i major
ticks
axes[0].grid(color="grey", which="major", linestyle='-', linewidth=0.5)

axes[1].plot(x,y,lw=2)
axes[1].axis('tight')
axes[1].set_title("axis('tight')")

#associo al grafico i minor e major ticker
axes[1].xaxis.set_major_locator(x_major_ticker)
axes[1].xaxis.set_minor_locator(x_minor_ticker)
axes[1].yaxis.set_major_locator(y_major_ticker)
```

```
axes[1].yaxis.set_minor_locator(y_minor_ticker)

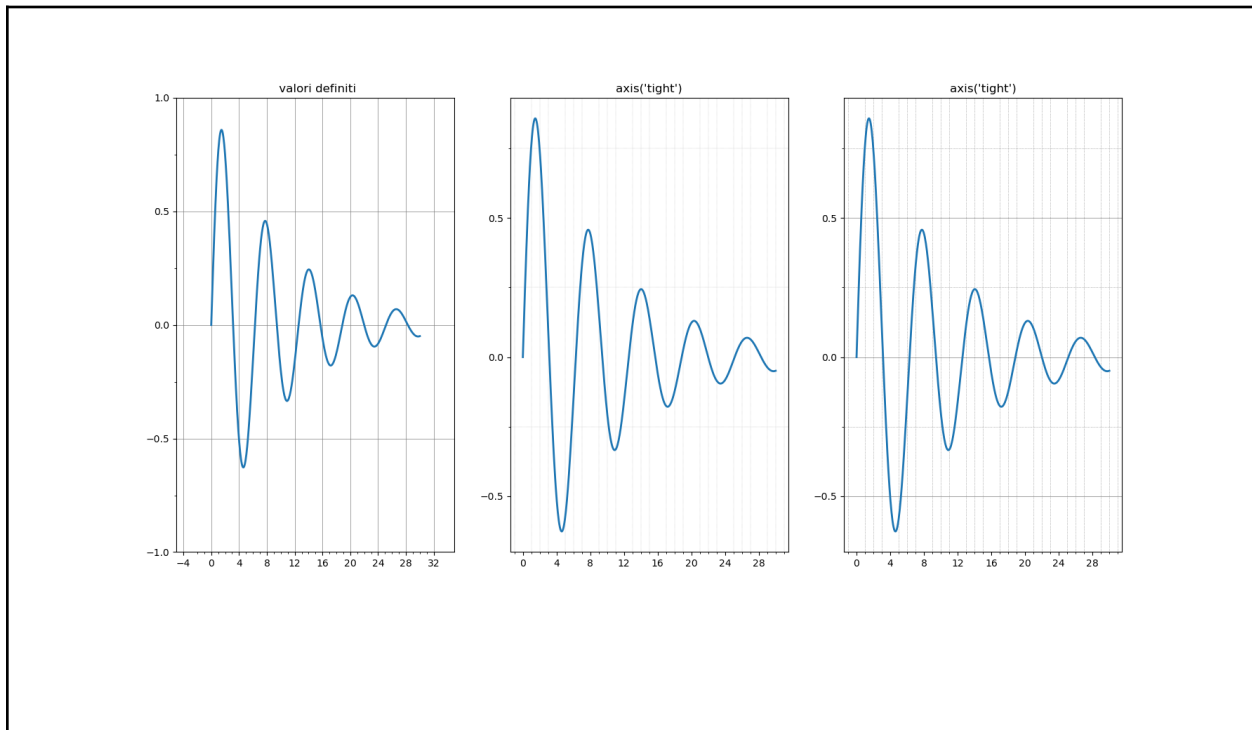
# definisco una griglia per il secondo grafico, che identifica i minor
ticks
axes[1].grid(color="grey", which="minor", linestyle=':', linewidth=0.25)

axes[2].plot(x,y,lw=2)
axes[2].axis('tight')
axes[2].set_title("axis('tight')")

#associo al grafico i minor e major ticker
axes[2].xaxis.set_major_locator(x_major_ticker)
axes[2].xaxis.set_minor_locator(x_minor_ticker)
axes[2].yaxis.set_major_locator(y_major_ticker)
axes[2].yaxis.set_minor_locator(y_minor_ticker)

# definisco una griglia per il terzo grafico, che identifica i major
ticks sull'asse y
# e i minor ticks sull'asse x
axes[2].grid(color="grey", which="minor", axis='y', linestyle=':',
linewidth=0.5)
axes[2].grid(color="grey", which="major", axis='y', linestyle='-',
linewidth=0.5)
axes[2].grid(color="grey", which="minor", axis='x', linestyle=':',
linewidth=0.5)

plt.show()
```



Formattazione del testo ed annotazioni

In molti grafici è utile personalizzare in modo accurato le descrizioni delle curve, i font utilizzati ed eventuali annotazioni. Tutti i possibili parametri sono presenti in un *dictionary* di *matplotlib* che si chiama **`mpl.rcParams`** e per visualizzare tutte le possibili modifiche si può accedere direttamente a questo dizionario nel seguente modo:

```
import matplotlib as mpl

print(mpl.rcParams)
```

Per accedere a questo dizionario e modificare l'oggetto *Axes*, bisognerà utilizzare alcuni specifici metodi, messi a disposizione dell'oggetto stesso, come ad esempio per l'oggetto *ax* si potrà richiamare **`ax.text`** oppure **`ax.annotate`**.

Matplotlib nasce dall'esigenza di rappresentare al meglio ogni tipo di grafico, è quindi immaginabile che le possibilità sono moltissime e applicabili sia al singolo grafico che ad un

intero programma che include molteplici grafici. Per approfondire questi aspetti si consiglia di consultare la seguente pagina <https://matplotlib.org/3.3.3/tutorials/introductory/customizing.html> in questo contesto fare solo un esempio esplicativo.

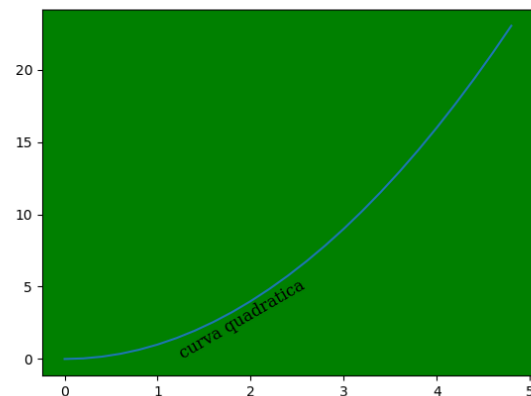
annotate.py

```
import matplotlib as mpl
import numpy as np
import matplotlib.pyplot as plt

t = np.arange(0., 5., 0.2)

mpl.rcParams['axes.facecolor'] = 'green'
plt.annotate("curva quadratica", xy=(1,0), fontsize=14, family="serif")

plt.plot(t, t**2)
plt.show()
```



Pandas

La libreria Pandas permette di elaborare in modo semplice strutture dati complesse, come serie o tabelle rendendo semplici trasformazioni, conversioni e manipolazioni in generale.

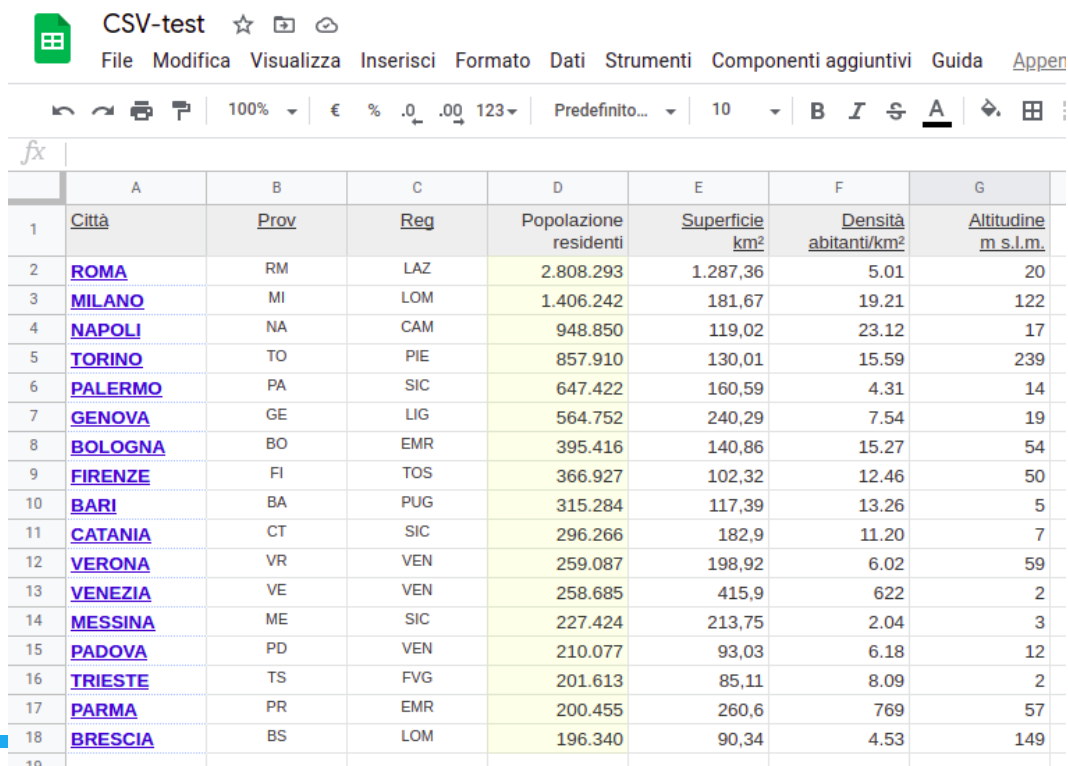
Accesso ai file

Introduzione all'importazione di dati

In molte occasioni bisogna interagire con file presenti nel proprio pc o comunque elaborati con altri strumenti. Spesso, inoltre, questi dati sono generati da altri soggetti, archiviati in banche dati oppure pubblicate on line. Risulta necessario, quindi, utilizzare un formato condiviso di scrittura e quindi di lettura di queste informazioni tramite file, così da renderle comprensibili ad applicazioni di diversa natura. Un formato di file utile a questo scopo è il csv (non è l'unico) attraverso cui è possibile scrivere su un file di testo dati strutturati, i cui elementi sono separati da virgole o da un segno di interpunzione scelto e identificato. Altro approccio alla memorizzazione dei dati è il JSON, un altro tipo di organizzazione delle informazioni, sempre in un file di testo puro, ma non strutturato. Vuol dire che i dati hanno un formato più simile ai dizionari (visti in python) o comunque a strutture organizzate per chiavi associate a strutture dati non necessariamente uguali tra loro.

CSV

Nell'immagine che segue è possibile visualizzare un file relativo ad un foglio elettronico, la prima riga rappresenta il significato del valore indicato nella colonna corrispondente.



	A	B	C	D	E	F	G
1	Città	Prov	Reg	Popolazione residenti	Superficie km²	Densità abitanti/km²	Altitudine m s.l.m.
2	ROMA	RM	LAZ	2.808.293	1.287,36	5,01	20
3	MILANO	MI	LOM	1.406.242	181,67	19,21	122
4	NAPOLI	NA	CAM	948.850	119,02	23,12	17
5	TORINO	TO	PIE	857.910	130,01	15,59	239
6	PALERMO	PA	SIC	647.422	160,59	4,31	14
7	GENOVA	GE	LIG	564.752	240,29	7,54	19
8	BOLOGNA	BO	EMR	395.416	140,86	15,27	54
9	FIRENZE	FI	TOS	366.927	102,32	12,46	50
10	BARI	BA	PUG	315.284	117,39	13,26	5
11	CATANIA	CT	SIC	296.266	182,9	11,20	7
12	VERONA	VR	VEN	259.087	198,92	6,02	59
13	VENEZIA	VE	VEN	258.685	415,9	622	2
14	MESSINA	ME	SIC	227.424	213,75	2,04	3
15	PADOVA	PD	VEN	210.077	93,03	6,18	12
16	TRIESTE	TS	FVG	201.613	85,11	8,09	2
17	PARMA	PR	EMR	200.455	260,6	769	57
18	BRESCIA	BS	LOM	196.340	90,34	4,53	149

In questa tabella è conservato un dato strutturato, i cui elementi sono indicati nella prima riga, colonna per colonna. Si potrebbe così ricostruire e dedurre un dato strutturato di questo tipo:

```
Popolazione_città(città, Prov, Reg, Popolazione residenti,
Superficie km², Densità abitanti/km², Altitudine m s.l.m.
```

Sarà possibile, quindi, dedurre che ogni riga rappresenta i dati relativi ad una città e, colonna per colonna, saranno deducibili i dati estratti nell'ordine indicato.

In sintesi è possibile ricavare dal file riportato che:

- L'intestazione della prima riga indica la struttura dei dati indicati nel file.
- Ogni riga successiva alla prima indica un dato strutturato, un'istanza del tipo di dato rappresentato nell'intestazione (nel caso in esame i dati relativi alla popolazione, città per città).
- Ogni colonna ha un suo tipo di dato semplice di riferimento, *Città* è un tipo di dato **testo**, come *Prov* e *Reg*, *Popolazione residenti* e *Altitudine* hanno un tipo di dato **intero**, invece le successive un tipo di dato numero **decimale**.

Il file in esempio, tuttavia, è un tipo di file elaborato da un determinato software, un foglio elettronico della suite di Google, GSuite. Per rendere questa tabella esportabile in un formato compatibile con la maggior parte dei linguaggi di programmazione è sufficiente scaricarlo in formato csv, così da generare un file di testo con estensione .csv e modificabile da qualunque editor di testo o foglio elettronico. In questo file ogni riga rappresenta un dato strutturato e le colonne separano gli elementi del singolo dato attraverso le virgole. Il file, aperto in un editor di testo, assumerà il seguente aspetto:

```

1 Città,Prov,Reg,"Popolazione residenti","Superficie km²","Densità abitanti/km²","Altitudine m s.l.m."
2 ROMA,RM,LAZ,2.808.293,"1.287,36",5.01,20
3 MILANO,MI,LOM,1.406.242,"181,67",19.21,122
4 NAPOLI,NA,CAM,948.850,"119,02",23.12,17
5 TORINO,TO,PIE,857.910,"130,01",15.59,239
6 PALERMO,PA,SIC,647.422,"160,59",4.31,14
7 GENOVA,GE,LIG,564.752,"240,29",7.54,19
8 BOLOGNA,BO,EMR,395.416,"140,86",15.27,54
9 FIRENZE,FI,TOS,366.927,"102,32",12.46,50
10 BARI,BA,PUG,315.284,"117,39",13.26,5
11 CATANIA,CT,SIC,296.266,"182,9",11.20,7
12 VERONA,VR,VEN,259.087,"198,92",6.02,59
13 VENEZIA,VE,VEN,258.685,"415,9",622,2
14 MESSINA,ME,SIC,227.424,"213,75",2.04,3
15 PADOVA,PD,VEN,210.077,"93,03",6.18,12
16 TRIESTE,TS,FVG,201.613,"85,11",8.09,2
17 PARMA,PR,EMR,200.455,"260,6",769,57
18 BRESCIA,BS,LOM,196.340,"90,34",4.53,149
  
```


Si noti inoltre che i dati hanno perso ogni formattazione del testo indicato nel foglio elettronico, che invece ha molte altre proprietà di gestione delle celle. Il Csv, essendo solo un formato di testo semplice, non permette infatti una formattazione elaborata del testo rappresentativo dei dati.

Per gestire file csv, Python integra nativamente un modulo **CSV**, basta importarlo negli script in cui serve, più utile che leggere il file come testo ed analizzare il contenuto.

Nel codice seguente viene letto il file csv dell'esempio precedente, generando liste relative alla singola riga, contenente i singoli dati estratti. Si noti come sia possibile estrarre i dati dal file ed importarli in un tipo di dato Dictionary.

lettura-csv.py

```
import csv
# apertura file riga per riga organizzando il risultato in liste
with open('testcsv.csv', 'r') as file:
    reader = csv.reader(file)
    for row in reader:
        print(row)

print("")
print("")

# apertura del file riga per riga organizzando l'output come dictionary
with open("testcsv.csv", 'r') as file:
    csv_file = csv.DictReader(file)
    for row in csv_file:
        print(dict(row))
```

```
[ 'Città', 'Prov', 'Reg', 'Popolazione residenti', 'Superficie km²', 'Densità abitanti/km²', 'Altitudine m s.l.m.' ]
[ 'ROMA', 'RM', 'LAZ', '2.808.293', '1.287,36', '5.01', '20' ]
[ 'MILANO', 'MI', 'LOM', '1.406.242', '181,67', '19.21', '122' ]
[ 'NAPOLI', 'NA', 'CAM', '948.850', '119,02', '23.12', '17' ]
[ 'TORINO', 'TO', 'PIE', '857.910', '130,01', '15.59', '239' ]
[ 'PALERMO', 'PA', 'SIC', '647.422', '160,59', '4.31', '14' ]
[ 'GENOVA', 'GE', 'LIG', '564.752', '240,29', '7.54', '19' ]
[ 'BOLOGNA', 'BO', 'EMR', '395.416', '140,86', '15.27', '54' ]
[ 'FIRENZE', 'FI', 'TOS', '366.927', '102,32', '12.46', '50' ]
[ 'BARI', 'BA', 'PUG', '315.284', '117,39', '13.26', '5' ]
[ 'CATANIA', 'CT', 'SIC', '296.266', '182,9', '11.20', '7' ]
[ 'VERONA', 'VR', 'VEN', '259.087', '198,92', '6.02', '59' ]
[ 'VENEZIA', 'VE', 'VEN', '258.685', '415,9', '622', '2' ]
[ 'MESSINA', 'ME', 'SIC', '227.424', '213,75', '2.04', '3' ]
[ 'PADOVA', 'PD', 'VEN', '210.077', '93,03', '6.18', '12' ]
[ 'TRIESTE', 'TS', 'FVG', '201.613', '85,11', '8.09', '2' ]
[ 'PARMA', 'PR', 'EMR', '200.455', '260,6', '769', '57' ]
[ 'BRESCIA', 'BS', 'LOM', '196.340', '90,34', '4.53', '149' ]

{ 'Città': 'ROMA', 'Prov': 'RM', 'Reg': 'LAZ', 'Popolazione residenti': '2.808.293', 'Superficie km²': '1.287,36', 'Densità abitanti/km²': '5.01', 'Altitudine m s.l.m.': '20' }
{ 'Città': 'MILANO', 'Prov': 'MI', 'Reg': 'LOM', 'Popolazione residenti': '1.406.242', 'Superficie km²': '181,67', 'Densità abitanti/km²': '19.21', 'Altitudine m s.l.m.': '122' }
{ 'Città': 'NAPOLI', 'Prov': 'NA', 'Reg': 'CAM', 'Popolazione residenti': '948.850', 'Superficie km²': '119,02', 'Densità abitanti/km²': '23.12', 'Altitudine m s.l.m.': '17' }
{ 'Città': 'TORINO', 'Prov': 'TO', 'Reg': 'PIE', 'Popolazione residenti': '857.910', 'Superficie km²': '130,01', 'Densità abitanti/km²': '15.59', 'Altitudine m s.l.m.': '239' }
{ 'Città': 'PALERMO', 'Prov': 'PA', 'Reg': 'SIC', 'Popolazione residenti': '647.422', 'Superficie km²': '160,59', 'Densità abitanti/km²': '4.31', 'Altitudine m s.l.m.': '14' }
{ 'Città': 'GENOVA', 'Prov': 'GE', 'Reg': 'LIG', 'Popolazione residenti': '564.752', 'Superficie km²': '240,29', 'Densità abitanti/km²': '7.54', 'Altitudine m s.l.m.': '19' }
{ 'Città': 'BOLOGNA', 'Prov': 'BO', 'Reg': 'EMR', 'Popolazione residenti': '395.416', 'Superficie km²': '140,86', 'Densità abitanti/km²': '15.27', 'Altitudine m s.l.m.': '54' }
{ 'Città': 'FIRENZE', 'Prov': 'FI', 'Reg': 'TOS', 'Popolazione residenti': '366.927', 'Superficie km²': '102,32', 'Densità abitanti/km²': '12.46', 'Altitudine m s.l.m.': '50' }
{ 'Città': 'BARI', 'Prov': 'BA', 'Reg': 'PUG', 'Popolazione residenti': '315.284', 'Superficie km²': '117,39', 'Densità abitanti/km²': '13.26', 'Altitudine m s.l.m.': '5' }
```

La funzione *reader* permette di analizzare il file anche se il carattere separatore è diverso. Ad esempio, se il separatore fosse il tabulatore, nello script precedente bisognerebbe sostituire la riga seguente:

```
reader = csv.reader(file, delimiter = '\t')
```

Si noti che la tabella creata attraverso questa funzione contiene la prima riga delle intestazioni che all'interno delle elaborazioni numeriche potrebbe dare fastidio. Esistono diversi modi per saltare la prima riga, il più semplice è cancellarla. Se invece la tabella è totalmente numerica e quindi si immagina di dover elaborare direttamente i valori del nostro file, esiste un metodo del modulo NumPy che legge file csv e trasforma il contenuto direttamente in valori numerici, a differenza del modulo csv che li trasforma in formato stringa. Nell'esempio che segue non verrà approfondito l'approccio attraverso il modulo NumPy, ma verrà rappresentato un modo semplice per convertire i valori estratti da un file csv in valori numerici manipolabili dal punto di vista matematico.

lettera-csv_elaborazione.py

```
import csv
# inizializzo un vettore dentro cui salvare la tabella
popolazione = []

'''
apertura file riga per riga organizzando il risultato in liste
salvate in POPOLAZIONE, il risultato finale sarà una matrice
'''

with open('Foglio1.csv', 'r') as file:
    reader = csv.reader(file)
    for row in reader:
        popolazione.append(row)

# elimino la prima riga, per elaborare la colonna della
# popolazione
popolazione.pop(0)

popolazioneTot = 0.; # da notare il punto dopo lo 0 ad indicare che
                    # la variabile conterrà un valore reale

print("somma della popolazione residente:")
for popol in popolazione:
    popol[3] = popol[3].replace(".", "") # elimino i punti
    popol[3] = popol[3].replace(",", ".") # convertito la virgola in punto

    popolazioneTot += float(popol[3]) # converto in numero reale

print("la popolazione totale è: ", popolazioneTot)
```

Si noti che nell'esempio precedente tutti i valori importati dal file sono considerati stringhe, Quindi hanno bisogno di una conversione attraverso un'operazione di casting. In quanto stringa, Python non è capace di interpretare i caratteri “.” che il foglio elettronico ha inserito all'interno dei numeri per indicare le migliaia e le centinaia di migliaia. Questo significa che un numero del tipo *100.000,34* non è riconoscibile da Python in quanto presenta caratteri non consoni alla conversione numerica. nell'esempio precedente, infatti dopo aver eliminato la prima riga (che non avrebbe permesso la conversione numerica della stringa relativa al campo della popolazione) si è proceduto alla eliminazione dei punti e dalla sostituzione della “.” con il carattere “,”, così da permettere un'operazione di casting corretta.

Si noti inoltre che il ciclo di estrapolazione delle righe dal file genera una lista di liste, quindi in sostanza una matrice dove ogni riga è una riga del file indicizzato colonna per colonna. diventa comprensibile quindi il perché all'interno del ciclo che somma tutti i valori relativi alla popolazione viene scelto l'indice 3 per la variabile *popol*.

JSON

Tutti i linguaggi moderni permettono di gestire le informazioni anche attraverso il formato **Json** (JavaScript Object Notation). Si tratta di un formato testuale per la rappresentazione di dati strutturati e permette di padroneggiare informazioni per una vastissima gamma di utilizzo.

JSON permette di aggregare dati per creare informazioni di più alto livello, rappresentando gli elementi di un database attraverso le caratteristiche che li contraddistinguono. La differenza con il formato csv è che in questo caso il peso maggiore viene dato al singolo elemento all'interno della struttura e non alla struttura stessa. Nel csv è stato posto come elemento fondamentale la struttura dei dati intesa come tabella, infatti è stato affrontato in correlazione ad un foglio elettronico. In questo caso, invece, il peso viene dato all'elemento appartenente alla struttura dati, questo vuol dire che un singolo elemento può avere anche attributi diversi rispetto ad un altro elemento appartenente alla stessa struttura.

Se si volesse rappresentare l'insieme degli allievi iscritti ad una scuola, si potrebbe usare una scrittura di questo tipo:

```
# oggetto dictionary
{
    "nome": "Edoardo",
    "cognome": "Rossi",
    "matricola": "S123456",
```

```
"corsi_precedenti":2,  
"laureato":true  
}  
  
# oggetto JSON, incluso negli apici  
{  
  "nome":"Edoardo",  
  "cognome":"Rossi",  
  "matricola":"S123456",  
  "corsi_precedenti":2,  
  "laureato":true  
}
```

Si noti come l'intera struttura viene racchiusa all'interno di parentesi graffe, che identificano un blocco. Ogni elemento è contraddistinto da una coppia **chiave-valore** separata da una virgola rispetto alla coppia successiva, inoltre una tale struttura permette ai linguaggi di programmazione di distinguere una stringa ("Eduardo") da un valore numerico (2) oppure booleano (true).

È possibile sintetizzare la costruzione di un oggetto JSON attraverso questi elementi:

- Un file di testo che contiene gli oggetti racchiusi tra parentesi graffe.
- Un insieme di coppie chiave-valore separate da due punti (:) e distinte tra loro attraverso la virgola (,).
- Per dichiarare un oggetto JSON nel codice bisogna racchiuderlo tra apici

L'esempio precedente include un solo oggetto all'interno del quale ci sono i dati relativi ad uno studente. Nel caso si volessero rappresentare più oggetti di quel tipo, anche se con tipologie di chiavi differenti, sarà sufficiente includere tutti gli oggetti all'interno di un Array separandoli con le virgole ed includendoli all'interno di parentesi quadre.

```
[  
  {  
    "nome":"Edoardo",  
    "cognome":"Rossi",  
    "matricola":"S13663",  
    "corsi_precedenti":2,  
    "laureato":true  
  },  
  {  
    "nome":"Francesca",  
    "cognome":"Bianchi",  
    "matricola":"S13664",  
    "corsi_precedenti":1,  
    "laureato":false  
  }  
]
```

```
[
  {
    "nome": "Paolo",
    "cognome": "Bianchi",
    "matricola": "S643577",
    "corsi_precedenti": 3,
    "laureato": true
  },
  {
    "nome": "Renato",
    "cognome": "Gialli",
    "matricola": "S753578",
    "corsi_precedenti": 4,
    "laureato": false
  }
]
```

Si tenga presente che nell'esempio appena illustrato la struttura dati contiene tutti oggetti con le stesse chiavi, questo però non è sempre vero ed è la sostanziale differenza con il formato csv. In questo caso, infatti, ogni studente avrebbe potuto avere campi diversi nel caso in cui ci fossero state delle personalizzazioni maggiori rispetto alle caratteristiche del singolo studente.

In Python il modulo JSON è builtin, è possibile quindi convertire agevolmente strutture dati di Python in oggetti JSON e viceversa. La combinazione di liste e dizionari permette così strutture molto versatili, per esempio è possibile costruire archivi formati da liste di dizionari di liste con un numero variabile di elementi.

In particolare possiamo distinguere quattro metodi fondamentali di questo modulo:

- **json.dumps**, per convertire in un oggetto JSON un oggetto di tipo lista o dictionary.
- **json.loads**, per fare il parsing (lettura) di un oggetto JSON e convertirlo nel corrispondente oggetto Python.
- **json.dump**, per salvare su file un oggetto JSON.
- **json.load**, per leggere da file un oggetto JSON.

È possibile convertire in oggetti JSON i seguenti elementi:

- dict

- list
- tuple
- string
- int
- float
- True
- False
- None

elementi_base/json_conversioni.py

```
'''  
STRUTTURA DI UN OGGETTO JSON  
[  
    {  
        "nome": "Edoardo",  
        "cognome": "Rossi",  
        "matricola": "S13663",  
        "corsi_precedenti": 2,  
        "laureato": true  
    },  
    {  
        "nome": "Paolo",  
        "cognome": "Bianchi",  
        "matricola": "S643577",  
        "corsi_precedenti": 3,  
        "laureato": true  
    },  
    {  
        "nome": "Renato",  
        "cognome": "Gialli",  
        "matricola": "S753578",  
        "corsi_precedenti": 4,  
        "laureato": false  
    }  
]
```

```
]

'''

import json

# oggetto dictionary:
x = {
    "name": "John",
    "age": 30,
    "city": "New York"
}

# conversione in JSON:
z = json.dumps(x)

# risultato:
print("convertiamo un dizionario", z, "\n")

# oggetto lista:
y = [ "name", "John", "age", 30, "city", "New York"]

# conversione in JSON:
z = json.dumps(y)

# risultato:
print("convertiamo una lista", z, "\n")

# conversione in JSON, ma con indentazione:
z = json.dumps(x, indent=4)

# risultato:
print("convertiamo una lista, ma con l'indentazione", z)

dic = json.loads(z)
```



```
print("z: ",type(z))
print("dic: ",type(dic),"\\n")

print("alcune conversioni da oggetto Python a stringa di tipo JSON")
print("True ",json.dumps(True))
print("False ",json.dumps(False))
print("None ",json.dumps(None))
print("lista ",json.dumps(["apple", "banana"]), " in vettore")
print("Tupla ",json.dumps(("apple", "banana")), " in vettore")
```

[illegible]

convertiamo un dizionario {"name": "John", "age": 30, "city": "New York"}

convertiamo una lista ["name", "John", "age", 30, "city", "New York"]

convertiamo una lista, ma con l'indentazione {

```
"name": "John",  
"age": 30,  
"city": "New York"
```

}

```
z: <class 'str'>
```

```
dic: <class 'dict'>
```

alcune conversioni da oggetto Python a stringa di tipo JSON

True true

False false

None null

lista ["apple", "banana"] in vettore

Tupla ["apple", "banana"] in vettore

- Si noti come il risultato di una conversione di lista o di dictionary sia lo stesso, in particolare gli elementi di una lista vengono associati a due a due.
- La tipologia di dati python viene convertita in oggetto Json, mutando il tipo di valore. Ad esempio *None* diventa *Null*, *True*, diventa *true*, una *lista* prende la forma di un *vettore* e così via.

- Un oggetto JSON è in sostanza un stringa, per rappresentarlo in modo leggibile, quindi con indentazione va indicato come secondo parametro in *dumps*.

Si osservi il seguente esempio:

accessoFile/lettura_scrittura_json.py

```
'''
Salvare su file una struttura dati complessa
e successivamente ricaricarla dal file
'''
import json

dati = {

    "uno" : [1],
    "due" : {"uno": "a", "due": "b"},
    "tre" : [(1,), (4,3), (2,6,8)],
    "quattro": "una stringa",
    "cinque" : {"uno": 1, "due": 2, "tre":3, "quattro": 4}

}

# salvo su file

with open("data.json", "w") as f:
    json.dump(dati, f)
    # json.dump(dati, f, indent=4)

f.close()

# estrapolo da file

with open("data.json", "r") as f:
    dati_da_file = json.load(f)
```

[illegible]

Il metodo `json.dump` ha bisogno dell'oggetto da salvare ed il nome del file all'interno del quale andare a scrivere, invece il metodo `json.load` ha bisogno solo del nome del file e della variabile all'interno della quale salvare i valori estratti dal file.

Bibliografia

<https://www.python.org/>

<https://docs.python.org/3.8/>

<https://www.python.it/>

<https://www.w3schools.com/python/default.asp>

<https://it.wikipedia.org/wiki/Python>

Nuerical Python, Scientific Computing and Data Science Application whith Numpy, SciPy and Matplotlib - Second Edition. Robert Johansson, Apress.

Python By Example. Nichola Lacey, Cambridge

Creative coding in PYTHON. Sheena Vaidyanathan, Quarry