

Sommario

01. Introduzione e Installazione di Python	2
02. Valori e Operatori Numerici	6
03. Variabili e Stringhe	12
04. Conversioni tra tipi di dato, funzioni print e input.....	16
05. Controllo del Flusso, Algebra Booleana e Operatori di Comparazione	20
06. Istruzioni if, elif ed else	28
07. Ciclo while, istruzioni break e continue	34
08. Il Ciclo for e la funzione range	39
09. I Moduli della Standard Library.....	43
10. Crea una Calcolatrice con Python.....	56
11. Le Funzioni in Python.....	59
12. Variabili Globali e Variabili Locali	65
13. Le Liste	69
14. Metodi delle Stringhe	78
15. Tuple e Set.....	85
16. I Dizionari	92
17. Gestione degli Errori.....	98
18. Ambienti Virtuali in Python con venv.....	102

01. Introduzione e Installazione di Python

Questo corso è realizzato per tutte quelle persone che intendono imparare il linguaggio di programmazione Python, ma anche per tutte quelle persone che intendono imparare a programmare iniziando con Python. Per questo motivo oltre a vedere assieme tutte le basi del linguaggio faremo anche un breve accenno ai concetti di Algoritmo e Diagramma di Flusso, utili a permetterci di iniziare a sviluppare l'elasticità mentale necessaria a creare applicazioni complesse.

Perché imparare Python?

Perché pochi linguaggi possono offrirti la potenza e la semplicità che può offrirti Python!

Se diamo uno sguardo alla [pagina Wikipedia del suo creatore Guido van Rossum](#), leggiamo che nel 1999 van Rossum presentò un progetto chiamato "Programmazione per tutti" al DARPA, ovvero l'agenzia americana per i progetti di ricerca avanzata per la difesa, descrivendo Python come:

- semplice, intuitivo e potente
- open source
- facilmente comprensibile e con tempi di sviluppo brevi

Col tempo Python è cresciuto tantissimo, diventando uno dei linguaggi di programmazione più usati al mondo, e assieme a lui è cresciuta una grande comunità di professionisti e appassionati da tutto il mondo che ogni giorno usano Python come linguaggio principale, così che al giorno d'oggi con Python è possibile fare davvero di tutto.

Oltre agli ambiti di sviluppo "tradizionali", come la creazione di script o semplici applicativi desktop, Python trova ampio utilizzo anche in ambito scientifico e matematico, e avrete sentito parlare di Data Science e Machine Learning, ma anche di siti come Instagram, YouTube, o Dropbox, anch'essi scritti in gran parte proprio con questo linguaggio.

Quindi che siate studenti o studentesse, semplici appassionati o imprenditori intraprendenti, scegliendo di imparare Python con questo corso avete fatto davvero un'ottima scelta!

Questo corso va bene per il mio sistema operativo?

Questo corso è stato registrato su Windows 11, ma è pienamente compatibile anche con Windows 10, Mac OS e Linux.

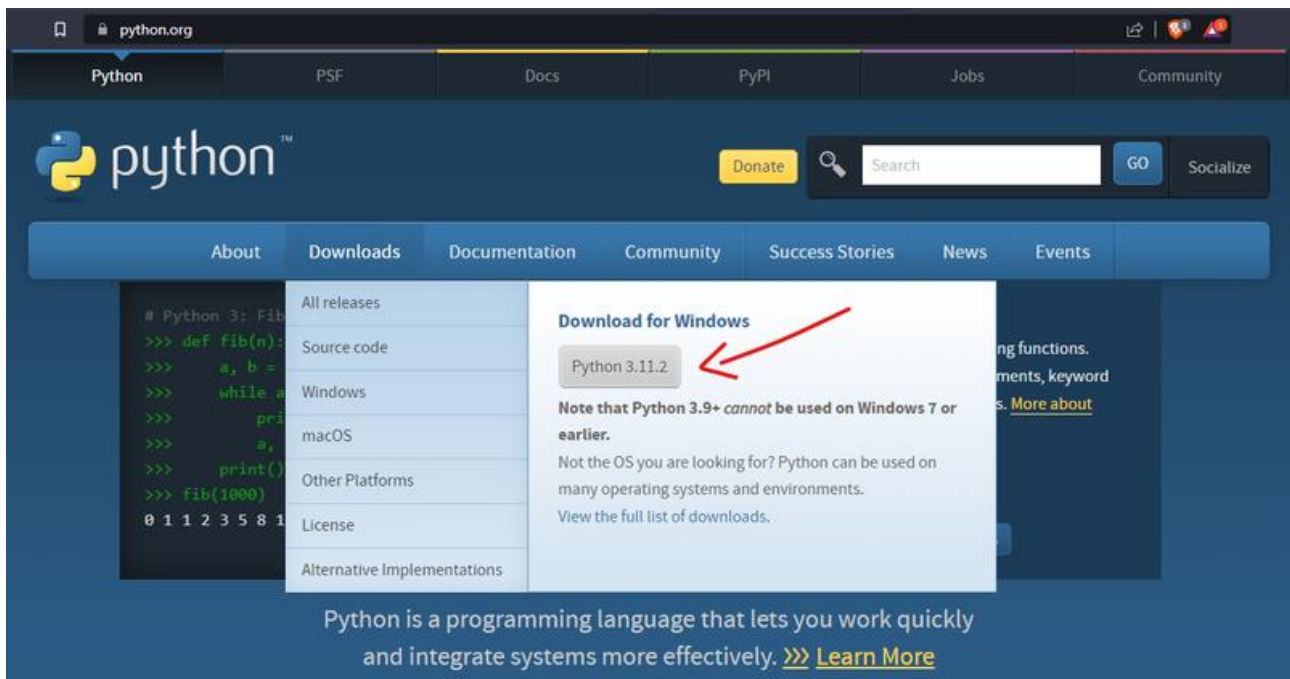
Per quanto riguarda Mac OS e Linux ciò che cambia è che verosimilmente non avrete bisogno di installare Python, in quanto si trova spesso già installato come parte del software del Sistema Operativo.

Come installare Python su Windows 10 o Windows 11

Installare Python su Windows è molto semplice: vi basterà andare sul [Sito Ufficiale della Python Software Foundation](https://www.python.org) e cliccare sul tasto di download per scaricare l'ultima versione disponibile.

In fase di installazione assicuratevi di dare la spunta per l'aggiunta di Python alla [variabile d'ambiente PATH](#), questo è molto importante per facilitarne l'utilizzo soprattutto in alcune lezioni del corso e in contesti di sviluppo reali.

Per tutti i dettagli relativi a quali spunte lasciare attive, seguite il video allegato a questa lezione!

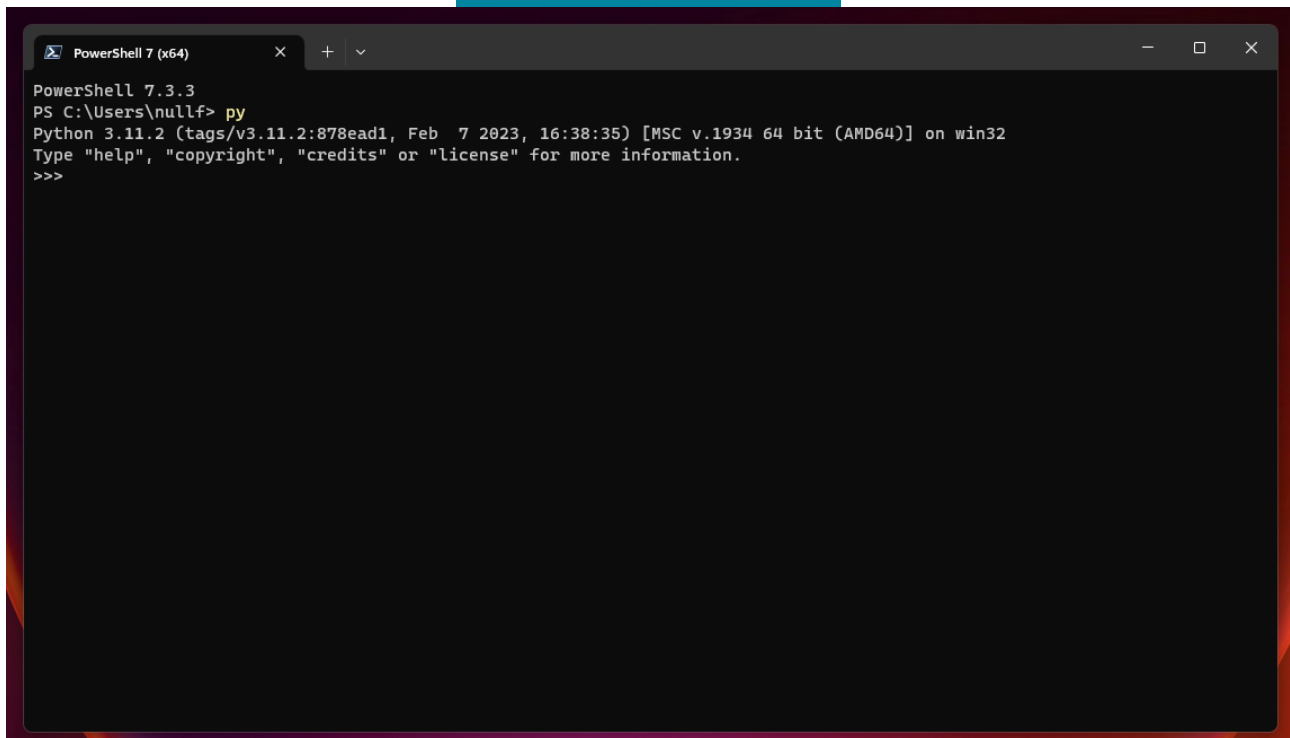


Come verificare la corretta installazione di Python

Qualsiasi sia il vostro sistema operativo, potete verificare la corretta installazione di Python aprendo un terminale di sistema e provare a dare il comando **python**, come mostrato nel video di questa lezione.

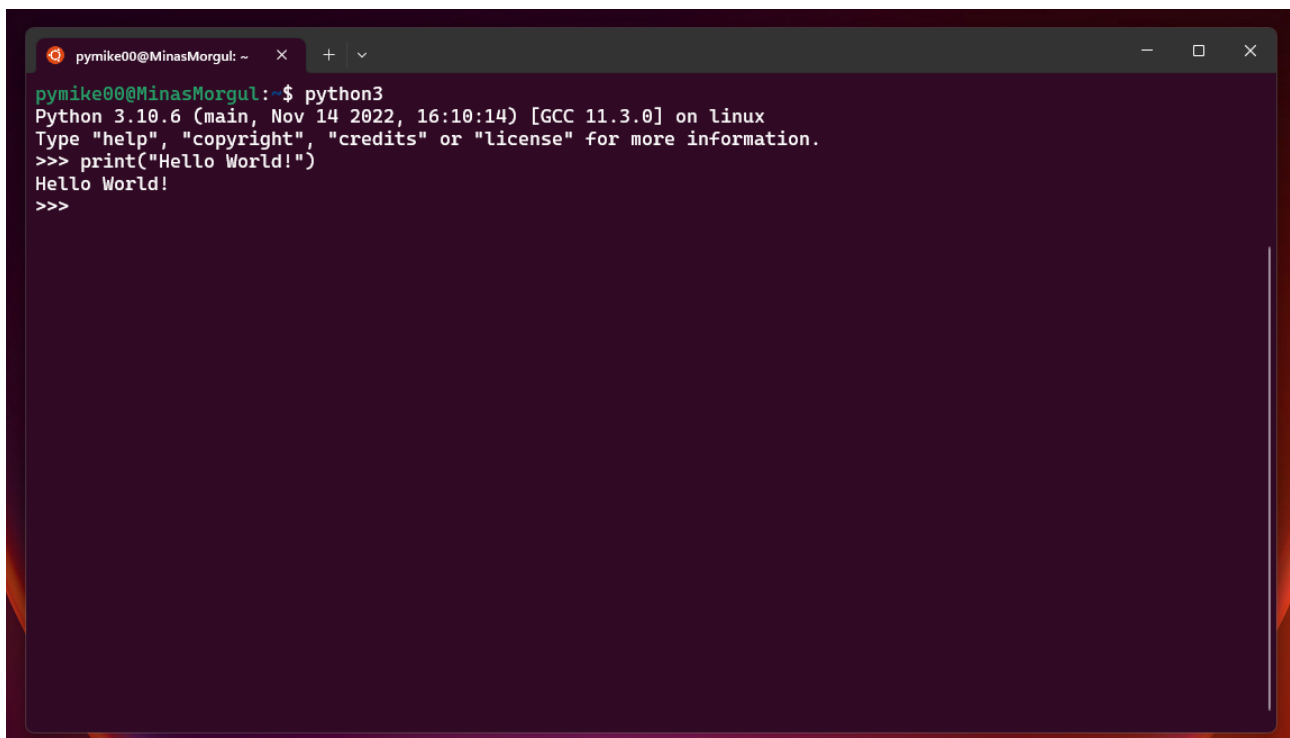
Su Windows dovreste inoltre poter usare il comando **py** e su Linux o Mac OS potreste dover usare il comando **python3** invece di **python**.

Se Python è stato installato correttamente o se è già presente nel vostro sistema operativo, così facendo dovrebbe aprirsi una shell interattiva di Python, con cui prenderemo confidenza nelle prossime lezioni mentre ci addentriamo nell'uso di questo fantastico linguaggio di programmazione.



```
PowerShell 7 (x64)
PowerShell 7.3.3
PS C:\Users\nullf> py
Python 3.11.2 (tags/v3.11.2:878ead1, Feb 7 2023, 16:38:35) [MSC v.1934 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Shell di Python - PowerShell su Windows 11



```
pymike00@MinasMorgul: ~
pymike00@MinasMorgul:~$ python3
Python 3.10.6 (main, Nov 14 2022, 16:10:14) [GCC 11.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello World!")
Hello World!
>>>
```

Shell di Python - Linux Ubuntu

Come Aggiornare Python su Windows

Potrebbe capitarvi di dover utilizzare una versione diversa di Python, o di voler aggiornare la versione di sistema.

Per aggiornare Python sul sistema operativo Windows, sia questo 10 o 11, è sufficiente scaricare ed installare un nuovo eseguibile. Potrete poi usare il launcher da PowerShell o CMD, per selezionare la versione che intendete utilizzare:

```
py -3.X  
# esempio:  
py -3.9  
py -3.11
```

Come Aggiornare Python su Linux, Mac OS e WSL

Aggiornare Python su Linux e Mac OS (così come WSL, Windows Subsystem for Linux) risulta più complicato. Essendo questi sistemi operativi dove Python viene spesso utilizzato nativamente, sostituendo la versione principale del sistema potreste creare danni. Fortunatamente possi

amo usare [pyenv](#). Si tratta di uno strumento che permette di installare comodamente qualsiasi versione di Python. Abbiamo un [tutorial dedicato a pyenv](#) che vi consigliamo di visionare!

02. Valori e Operatori Numerici

Nella lezione precedente abbiamo visto come installare Python su Windows e abbiamo detto che verosimilmente, su Linux e Mac OS Python si trova già installato. Quanto vedremo d'ora in avanti sarà pienamente compatibile tra i diversi sistemi operativi perché le istruzioni saranno le stesse.

In questa lezione vedremo come usare Python, introdurremo i due primi tipi di dato del corso, *integer* e *float*, e parleremo di **valori** e **operatori**.

Come si "usa" Python?

Anzitutto diciamo che è possibile utilizzare Python anche da un semplice terminale di sistema. Possiamo usare i comandi **python** o **py** su Windows, e **python** o **python3** su Linux e Mac OS.

Visual Studio Code

Per la scrittura di programmi e script, nella gran parte delle lezioni di questa serie noi abbiamo utilizzato [Visual Studio Code](#), un editor di codice **open source** sviluppato da Microsoft e compatibile con Windows, Linux e MacOS. Vi consigliamo di provarlo perché ha tantissime funzionalità che semplificano notevolmente la vita degli sviluppatori: è completamente personalizzabile, permette di lavorare contemporaneamente su più script e ha funzionalità utilissime come il debugging integrato, l'integrazione con [Git](#) e tante altre.

Soprattutto è possibile installare moltissimi plugin ed estensioni per espandere le funzionalità dell'editor, supporta moltissimi linguaggi di programmazione oltre a Python ed è perfetto per gestire progetti complessi. Vedremo come utilizzarlo più avanti nel corso.

IDLE

Se vi state avvicinando per la prima volta al mondo del linguaggio Python potete usare anche **IDLE**, acronimo di **Integrated Development and Learning Environment**, traducibile come ambiente di sviluppo e apprendimento integrato. Troverete IDLE di default su Windows in quanto fa parte del pacchetto di installazione di Python, mentre su Linux e Mac OS potreste doverla installare separatamente.

Tuttavia è bene precisare che IDLE mette a disposizione una "versione ottimizzata" (per l'apprendimento) della stessa shell interattiva che si può ottenere richiamando Python da terminale con uno dei comandi citati in precedenza, e che quindi tutti i comandi che daremo saranno identici.

Qualora vi troviate su Linux o Mac OS e non vogliate installare IDLE, potete seguire il corso da una qualsiasi shell.

La Shell Interattiva

La Shell Interattiva di Python esegue le istruzioni una alla volta man mano che queste vengono inserite dagli utenti.

È comoda per fare semplici calcoli o per sperimentare, imparando cosa facciano le diverse istruzioni di Python. Scriviamo:

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

Non appena daremo INVIO, l'istruzione verrà processata e all'istante ci verrà restituito un risultato:

```
Hello, World!
```

In questo caso l'istruzione **print()** ci restituisce il famoso **Hello World**, che è tradizionalmente il primo programma che viene lanciato in ciascun linguaggio di programmazione. Complimenti, avete ufficialmente scritto la vostra prima riga di codice Python!

Come commentare in Python

I **commenti di Python** permettono di inserire del testo all'interno del codice sorgente che viene **ignorato** dall'interprete Python. Vengono utilizzati per descrivere il codice e per renderlo più comprensibile sia agli sviluppatori che lavorano in team sia a chi scrive codice da solo, perché dopo un po' di tempo è facile dimenticarsi dettagli utili riguardo ai propri script e fa comodo avere dei punti di riferimento.

Per inserire un commento in Python, basta utilizzare il carattere **#** all'inizio di una riga: tutto ciò che lo segue (sulla stessa riga) non verrà eseguito. Ad esempio, se vogliamo inserire un commento per descrivere una variabile *x*, possiamo scrivere:

```
# Inizializziamo la variabile x a 0
```

```
x = 0
```

Si possono inserire commenti anche all'interno del codice. Questo può essere utile per spiegare parti specifiche del codice o per disabilitare temporaneamente alcune righe di codice mentre si scrive (come durante il debugging). Ad esempio:

```
x = 2 # Inizializziamo la variabile x a 2
```

```
# Commentiamo la riga che segue per disabilitarla temporaneamente
```

```
# x = 3
```

```
print(x)
```

```
# output
```

```
2
```

In questo caso, il commento sulla seconda riga disabilita temporaneamente l'assegnazione della variabile *x* e la stampa del valore della variabile *x* sarà 2.

È comunque consigliabile evitare di scrivere commenti superflui o ovvi: dovrebbero essere utili e fornire informazioni aggiuntive, come il ragionamento dietro una particolare implementazione o la logica di un algoritmo o la segnalazione di eventuali problemi o bug.

Vedremo più avanti anche un altro modo per commentare il codice e fornire informazioni, soprattutto per quanto riguarda lo scopo di classi e funzioni: le docstring.

Espressioni, Valori e Operatori

Con la shell di Python possiamo effettuare anche semplici **Operazioni Numeriche**. Scrivete $3 + 3$ e date Invio:

```
>>> 3 + 3
```

```
# output
```

```
6
```

Ecco che otteniamo il risultato istantaneamente.

Questa è definita un'espressione, con i numeri chiamati **valori** e il simbolo del **+** chiamato **operatore**: questo è importante perché ci inizia a predisporre mentalmente per pensare al codice come a un insieme di istruzioni dove a ciascuna corrisponde un ruolo ben preciso.

Come vedremo, nel mondo della programmazione esistono diversi tipi di valori e operatori, non solo numerici. Per ora, proseguiamo imparando il resto degli operatori numerici, anche perché ci sono alcune considerazioni da effettuare.

Come intuibile, usiamo il simbolo **-** per effettuare sottrazioni:

```
>>> 6 - 3
```

```
# output
```

```
3
```

Lo slash avanti **/** per effettuare divisioni:

```
>>> 9 / 3
```

```
# output
```

```
3.0
```

Mentre l'operatore per la moltiplicazione è l'asterisco *****:

```
>>> 5 * 5
```

```
# output
```

```
25
```

Attenzione a non confondere l'asterisco con una **x**, in quanto questa non è un operatore valido in Python!

Usare una **x** per tentare di effettuare moltiplicazioni porterà al manifestarsi di errori di programmazione, di cui parleremo più avanti nel corso.

```
>>> 5 x 5
```

```
# output
```

```
SyntaxError: invalid syntax
```


In questi primi esempi abbiamo usato valori appartenenti al primo tipo di dato di Python: l'**integer** o intero.

I numeri con la virgola sono invece chiamati **float**.

Float e Integer sono due tipi di dato di Python.

Per definire un *float* utilizziamo un punto al posto della virgola.

```
>>> 0.25
```

Otteniamo inoltre sempre un *float* come risultato di una divisione numerica:

```
>>> 9 / 3
```

```
# output
```

```
3.0
```

La funzione integrata **type()**

Ora, come fare a riconoscere i vari tipi di dato di Python? Chiaramente durante questo corso impareremo a riconoscerli e a usarli, ma come fare ciò tramite Python?

Su Python, per fare ciò possiamo usare la funzione integrata **type()**. Creiamo una **variabile** (parleremo di variabili nel dettaglio più avanti nel corso) chiamata *x* e assegniamo a questa il numero intero 69.

Possiamo quindi passare *x* alla funzione **type()** per far sì che Python ci dica che tipo di dato contiene:

```
>>> x = 69
```

```
>>> type(x)
```

```
# output
```

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

Chiaramente ci riferisce che si tratta di un intero! Ma proviamo ora a cambiare il valore da 69 a 4.80 e richiamiamo nuovamente la funzione **type()**:

```
>>> x = 4.80
```

```
>>> type(x)
```

```
# output
```

```
<class 'float'>
```

Ed ecco che ora ci troviamo proprio un *float*!

Oltre alle quattro operazioni fondamentali, Python vi semplifica il lavoro coi numeri permettendovi di effettuare altre operazioni, ciascuna con un suo operatore specifico.

Per effettuare un **calcolo esponenziale** usiamo come **operatore** ******:

```
>>> 3 ** 3  
27
```

Possiamo effettuare "divisioni intere" tramite un operatore detto **floor division**, rappresentato in Python da **//**:

```
>>> 30 // 7  
4
```

E infine possiamo ottenere il **resto** di una divisione utilizzando un operatore chiamato **modulo**, rappresentato dal simbolo **%**:

```
>>> 30 % 7  
2
```

La funzione `isinstance()`

Un'altra funzione integrata molto utile per il confronto dei tipi di dato in Python è **`isinstance()`**. Questa funzione viene utilizzata per verificare se un oggetto è di un certo **tipo** o se è una sottoclasse di quel tipo. La sintassi è la seguente:

```
isinstance(object, type)
```

Dove *object* è l'oggetto da verificare e *type* è il tipo o la classe da confrontare.

NOTA: Parleremo nel dettaglio di oggetti e classi nelle lezioni dedicate alla [programmazione a oggetti](#), per ora ci basta sapere che in Python **tutto è un oggetto**, e il tipo di un oggetto definisce il suo comportamento e le operazioni che possono essere eseguite su di esso.

La funzione **`isinstance()`** restituisce *True* (in italiano **Vero**) se l'oggetto è di dello stesso tipo di *type* o una sua sottoclasse, altrimenti restituisce *False* (in italiano **Falso**). Ad esempio:

```
>>> x = 5  
  
>>> isinstance(x, int)  
  
# output  
True
```

La funzione verifica se la variabile *x* è di tipo **`int`**: dal momento che *x* è 5, la funzione restituisce *True* e viene stampato il messaggio.

True e *False* sono un tipo di dato di cui parleremo nel dettaglio quando ci occuperemo di **logica booleana**.

`isinstance()` è molto utile quando è necessario verificare il tipo di un oggetto prima di eseguire un'operazione su di esso: ad esempio si potrebbe utilizzare per verificare se un valore inserito dall'utente è di tipo numerico prima di eseguire un calcolo matematico.

Gli Operatori Numerici che abbiamo visto sono:

- addizione: +
 - sottrazione: -
 - divisione: /
 - moltiplicazione: *
 - esponente: **
 - quoziente: /
 - /
 - resto: %
-

03. Variabili e Stringhe

In questa lezione introdurremo il concetto di **variabile** e un nuovo tipo di dato chiamato **stringa**, che ci consente di utilizzare del testo nei nostri programmi.

Cosa sono le variabili?

Vi siete mai chiesti come sia possibile in un programma salvare dei valori nella memoria del computer e accedervi poi in un secondo momento? Questo avviene in parte per mezzo di quelle che chiamiamo **variabili**.

Pensate a una variabile come ad un contenitore che può contenere un valore, e l'inserimento del valore in questo contenitore avviene tramite un processo chiamato **assegnazione**, rappresentato dal simbolo dell'uguale.

```
>>> x = 3.14
```

Semplificando il processo per fini didattici, possiamo dire che con questo comando è come se avessimo creato una "scatola" nella memoria del computer, chiamandola x. All'interno di questa scatola abbiamo quindi depositato il numero 3.14.

Fino a che lasceremo la scatola lì senza metterci nient'altro dentro potremo accedere di nuovo al valore ad essa assegnato utilizzando il nome della scatola, quindi della variabile, come collegamento al valore contenuto.

Provate a scrivere il nome della variabile sulla Shell Interattiva, e date invio:

```
>>> x
```

```
3.14
```

Ed ecco che ci riappare il numero che avevamo appena depositato.

Nell'universo Python vi capiterà spesso di incontrare nomi quali *spam*, *eggs* e *bacon*, in onore di uno sketch del famoso gruppo comico [Monty Python](#), da cui prende il nome il linguaggio stesso. Se siete interessati a qualche altra curiosità riguardante il linguaggio, potete dare uno sguardo a questo video [Cinque Fatti e Curiosità su Python](#).

In maniera analoga in Italia si usano nomi come *pippo* e *pluto*.

Se questi sono nomi perfetti per fare dei semplici esempi, tenete sempre a mente che per quanto possibile, in contesti reali è importante dare alle variabili dei nomi significativi.

Nel nostro caso, il nome più appropriato per la nostra variabile è sicuramente pi

```
>>> pi = 3.14
```

È un po' come prendere un barattolo, metterci dentro dello zucchero o del caffè, scriverci sopra il nome del contenuto con un'etichetta, e poi lasciarlo in cucina.

Chiunque passando in cucina potrà così facilmente intuire cosa sia presente all'interno del barattolo, ed accedervi quando necessario.

Lavorare con le variabili

Una volta che avrete richiamato una variabile, sarà possibile fare qualsiasi tipo di operazione sul valore ad essa assegnato, come ad esempio:

```
>>> pi * 3
```

```
9.42
```

L'operazione $pi * 3$ ci restituisce il numero 9.42, che è il triplo del valore contenuto in `pi`.

In Python ci sono delle regole per quanto riguarda i nomi che è possibile dare alle variabili: questi infatti possono contenere solo **lettere**, **numeri** e **underscore**, ovvero il trattino basso, ma non possono iniziare con un numero! Venire meno a queste regole comporta la comparsa di errori di sintassi.

Proviamo a creare una variabile e ad usare come primo carattere del nome un carattere numerico:

```
>>> 1esempio = 2
```

```
File "stdin", line 1
```

```
1esempio = 2
```

```
^
```

```
SyntaxError: invalid syntax
```

Per lo stesso motivo sopracitato, otteniamo un errore anche in questo caso:

```
>>> esem-pio = 5
```

```
File "", line 1
```

```
SyntaxError: can't assign to operator
```

Una delle caratteristiche più interessanti delle variabili è che potete assegnarle dei nuovi valori quante volte vi pare.

Se finora state pensando alle variabili come scatole a cui viene dato un nome e su cui è possibile inserire un valore, potete anche immaginare che a seconda del caso ci sia bisogno di inserirvi degli altri valori, giusto?

Un'altra caratteristica molto importante che distingue Python da altri linguaggi di programmazione, è che le variabili **non** sono di un tipo specifico: potete quindi assegnar loro qualsiasi classe di valore vogliate.

Facciamo un esempio definendo una nuova variabile `val`.

```
>>> val = 9.81
```

Usiamo ora la funzione integrata **type()** per vedere che tipo di valore è presente nella variabile `val`:

```
>>> type(val)
```

```
class 'float'
```

Si tratta chiaramente di un *float*! Proviamo ora ad assegnarle un intero.

```
>>> val = 1861
```

Andiamo ora a ricontrollare:

```
>>> type(val)
```

```
class 'int'
```

Ed ecco ora che nella nostra variabile *val* è presente un **integer (int)**.

Le stringhe in Python

Come accennato in precedenza, parliamo ora di un nuovo tipo di dato, le **stringhe** (in inglese **str**, abbreviazione di **string**): si tratta di valori di tipo testuale, cioè un insieme di caratteri inseriti tra apici o doppi apici, numeri e simboli compresi.

Creiamo per esempio una variabile chiamata *frase* e inseriamoci dentro la stringa *'Una volta imparato, Python si dimostra estremamente versatile'* racchiusa tra singoli apici:

```
>>> frase = 'Una volta imparato, Python si dimostra estremamente versatile'
```

Se ora andiamo ad effettuare un controllo con la funzione **type()** notiamo che la classe di dato contenuta nella variabile *frase* è appunto una **stringa (str)**

```
>>> type(frase)
```

```
class 'str'
```

Creiamo ora una seconda variabile chiamata *frase2* e inseriamoci dentro la stringa *"Python è uno dei migliori linguaggi per iniziare a programmare"*, ma stavolta includiamola tra doppi apici.

```
>>> frase2 = "Python è uno dei migliori linguaggi per iniziare a programmare"
```

Proviamo a richiamarne il valore dalla shell:

```
>>> frase2
```

```
'Python è uno dei migliori linguaggi per iniziare a programmare'
```

Ma come mai anche questa stringa ci viene restituita tra singoli apici quando noi l'abbiamo scritta usando invece doppi apici? Questo semplicemente perché Python è stato ideato in questa maniera, le stringhe vengono restituite in shell tra singoli apici.

Per rigor di logica, ci sono alcuni caratteri che non possono essere inclusi direttamente nelle stringhe, come ad esempio doppi apici in una stringa definita usando propri questi. In questo caso ci troveremo quindi impossibilitati a fare ad esempio delle citazioni:

```
>>> "e Dante scrisse: "Nel mezzo del cammin di nostra vita..."
```

```
File "", line 1
```

```
"e Dante scrisse: "Nel mezzo del cammin di nostra vita..."
```

^

SyntaxError: invalid syntax

Per Python è come se avessimo interrotto la stringa prima della sua stessa fine, e questo fa sì che ci restituisca un errore. Per ovviare a questo problema possiamo utilizzare gli apici singoli per definire la stringa, oppure utilizzare un **backslash** (\) prima dei caratteri "scomodi".

Riscriviamo ora la stessa stringa di prima, ma impiegando i backslash:

```
>>> "e Dante scrisse: \"Nel mezzo del cammin di nostra vita...\""
```

```
'e Dante scrisse: "Nel mezzo del cammin di nostra vita..."
```

Le stringhe possono inoltre essere concatenate tra di loro per ottenere delle concatenazioni di testo:

```
>>> eggs = "Meglio un uovo oggi..."
```

```
>>> bacon = "o una gallina domani?"
```

```
>>> eggs + bacon
```

```
'Meglio un uovo oggi...o una gallina domani?'
```

Come notate gli spazi non sono inclusi automaticamente, a quello dovrete pensare voi!

```
>>> eggs + " " + bacon
```

```
'Meglio un uovo oggi... o una gallina domani?'
```

Approfondiremo la nostra conoscenza del tipo di dato Stringa, introducendo metodi e sistemi di formattazione avanzata, [in una lezione dedicata](#).

04. Conversioni tra tipi di dato, funzioni print e input

In questa lezione parleremo di **conversioni tra tipi diversi di dato** e di due funzioni integrate in Python molto utilizzate, la funzione **print()** e la funzione **input()**.

Concluderemo inoltre la lezione scrivendo il nostro primo programma su **Visual Studio Code**!

Conversioni tra tipi di dato in Python

Finora abbiamo introdotto tre tipi di dato: *integer*, *float* e *string*: non abbiamo però spiegato come fare a combinare questi tipi di dato differenti tra di loro! Come fare ad esempio a combinare un numero intero con una stringa?

Magari state pensando di utilizzare un operatore come il **+**, e in questo caso ci siete andati vicini, ma non è così immediato. Facciamo delle prove: definiamo una variabile *age* (età) e assegniamole un numero intero:

```
>>> age = 30
```

Definiamo ora un'altra variabile e assegniamole una stringa:

```
>>> text = "La mia età è: "
```

Proviamo ora a combinare *age* e *testo* e vediamo cosa succede!

```
>>> text + age
```

```
# output
```

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

```
File "<pyshell#2>", line 1, in <module>
```

```
text + age
```

```
TypeError: can only concatenate str (not "int") to str
```

Ecco che, come forse alcuni di voi avranno sospettato, Python ci restituisce un errore.

Python sa che state cercando di effettuare una somma tra tipi di dato diversi, e vi avverte che l'operatore **+** non è compatibile per somme tra interi e stringhe. In sostanza vi avverte che non può effettuare la conversione necessaria ad effettuare la somma, a meno che non siate voi a chiederglielo in maniera esplicita.

Come chiedere quindi a Python di effettuare una conversione? Per fare questo esistono delle **funzioni integrate** che fanno la conversione per noi. Per convertire un valore in stringa ad esempio, usiamo la funzione **str()**.

Come capita spesso con le funzioni, come vedremo più avanti nel corso, le passiamo quello un **parametro** tra parentesi, e nel nostro caso il parametro sarà la variabile `age`:

```
>>> testo + str(age)
```

```
# output
```

```
'La mia età è... 30'
```

Analizziamo ora il caso opposto, ovvero il caso in cui abbiamo una stringa composta solo da caratteri numerici, e vogliamo convertirla in numero intero per usarla così in dei calcoli matematici.

Creiamo una variabile chiamata `arance_per_sacco` e assegniamole una stringa composta da caratteri numerici, in questo caso magari, un 13:

```
>>> arance_per_sacco = "13"
```

Come fare ora ad ottenere il numero totale di arance presenti in tre sacchi?

Guardate cosa succede se proviamo a moltiplicare la variabile per 3:

```
>>> arance_per_sacco * 3
```

```
# output
```

```
'131313'
```

Mmm... per quanto buone, cento trentunomila e tre centotredici arance sono davvero troppe!

La funzione che utilizzeremo in questo caso è, come è possibile immaginare per logica, la funzione **int()**.

Facciamo quindi:

```
>>> int(arance_per_sacco) * 3
```

```
39
```

Ed ecco che le arance sono 39!

Attenzione però, perché in questo modo non stiamo cambiando il valore associato alla variabile: sono le funzioni di conversione a restituirci il valore che stiamo poi utilizzando. Qualora dovessimo alterare il valore associato alla variabile dovremmo effettuare nuovamente l'assegnazione:

```
>>> arance_per_sacco
```

```
'13'
```

```
>>> type(arance_per_sacco)
```

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

```
>>> arance_per_sacco = int(arance_per_sacco)
```

```
>>> arance_per_sacco
```

```
13
```

```
>>> type(arance_per_sacco)
```

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

In maniera molto simile potete ottenere un *float* tramite la funzione **float()**.

Volendo convertire la variabile *age* del primo esempio facciamo:

```
>>> float(age)
30.0
```

La funzione print in Python

Bene, è arrivato il momento di introdurre quella che è forse la funzione più famosa nel regno di Python, la funzione **print()**.

Ciò che **print()** fa è mostrarci in output ciò che le passiamo come parametro:

```
>>> print(str(33) + " trentini entrarono a Trento... passando per la funzione print!")
# output
'33 trentini entrarono a Trento... passando per la funzione print!'
```

Per andare a capo si utilizza il carattere **newline**, ovvero **\n**, in questo modo:

```
>>> print("Prima riga\nSeconda riga\necc ecc...")
# output
Prima riga
Seconda riga
ecc ecc...
```

Oppure potete direttamente effettuare un **print multilinea** utilizzando 3 apici o doppi apici, in questo modo:

```
>>> print("""prima linea
seconda linea
terza linea...""")
prima linea
seconda linea
terza linea...
```

La funzione **print()** viene usata davvero ovunque, è raro trovare uno script in cui non sia presente almeno una volta.

Faremo degli esempi più complessi man mano che introdurremo nuovi concetti.

La funzione input in Python

La funzione **input()** invece ci serve per l'inserimento di dati da parte dell'utente, in fase di esecuzione.

Questo ci permette di rendere le cose estremamente interattive; **ciò che inserirete tramite input() verrà passato a Python sotto forma di stringa.**

Ad esempio, definiamo una variabile *x* ed assegniamole un valore introdotto tramite la funzione **input()**:

```
>>> x = input()
```

```
23
```

Come possiamo verificare con la funzione integrata **type()**, anche se abbiamo inserito apparentemente un numero, questo viene assegnato alla variabile sotto forma di stringa!

```
>>> type(x)
```

```
class 'str'
```

Scriviamo il nostro primo programma Python!

È arrivato ora il momento di scrivere il nostro primo programma!

In questo corso usiamo [Visual Studio Code](#), ma voi potete usare volendo qualsiasi editor testuale.

Vi rimando al video di questa lezione per un commento approfondito del codice qui mostrato!

```
# il tuo primo programma Python
robot_name = "Chappie"
robot_age = 1
print("Ciao! Il mio nome è " + robot_name + " e ho " + str(robot_age) + " anno")
user_name = input("Tu come ti chiami? ")
print("Ciao " + user_name + "!")

user_age = int(input("Quanti anni hai? "))
age_difference = user_age - robot_age

print(str(user_age) + " anni!? Sono " + str(age_difference) + " più di me!")
print("A presto!")
```

05. Controllo del Flusso, Algebra Booleana e Operatori di Comparazione

In questa lezione introdurremo i concetti di **Algoritmo** e **Diagramma di Flusso**, parleremo di **Operatori di Comparazione** in Python e vedremo le basi della **Logica Booleana**.

Il Flusso di Esecuzione

Nella lezione precedente abbiamo scritto il nostro primo programma Python.

Chiaramente trattandosi del primo programma del corso, il livello di complessità era piuttosto basso e l'ordine di esecuzione delle istruzioni lineare.

In contesti reali i programmi sono però spesso di gran lunga più complessi ed è presente una fetta sostanziosa di logica.

Ora, per creare dei programmi usando un linguaggio di programmazione come Python sarà fondamentale conoscere tutti i comandi base del linguaggio, così da poter rappresentare in codice non solo i dati che intendiamo elaborare, ma anche l'ordine di esecuzione delle varie istruzioni, detto **flusso di esecuzione**.

In questa lezione introdurremo molte di queste istruzioni fondamentali, ma prima di fare ciò è bene farsi una domanda. come ci si approccia alla scrittura di un programma?

Algoritmi e Diagrammi di Flusso

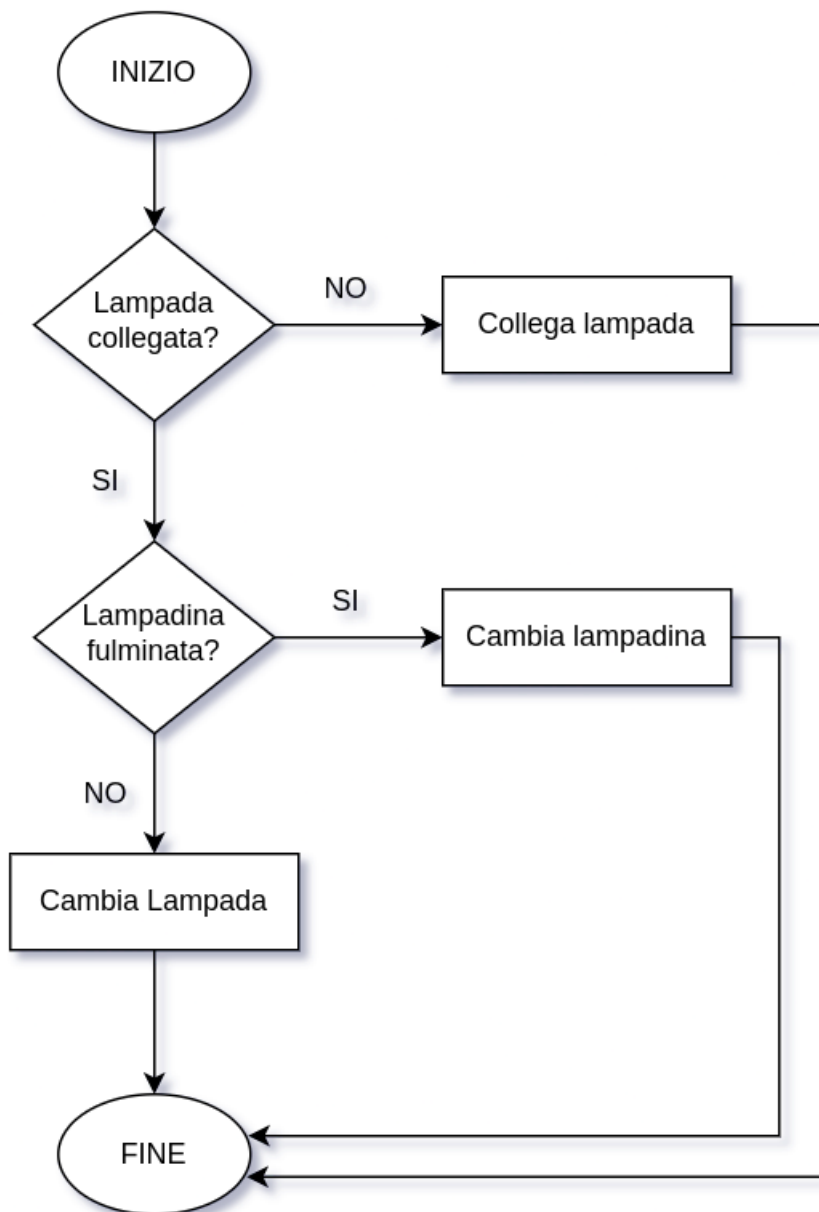
Vengono delineati uno o più **algoritmi**, che sono, citando la [pagina Wikipedia dedicata](#), "strategie atte alla risoluzione di un problema, costituite da una sequenza finita di operazioni (dette anche istruzioni)"

Si tratta quindi di procedimenti per risolvere un determinato problema attraverso un numero finito di passi elementari, e l'abilità di saper scomporre problemi apparentemente molto complessi in semplici step è una delle abilità più preziose che si possa acquisire tanto nella vita reale come anche nel mondo della programmazione.

Gli algoritmi possono essere anche molto articolati e in questi casi, per rappresentarne il flusso di esecuzione e controllo da un punto di vista visivo, è possibile utilizzare dei diagrammi come i [Diagrammi di Flusso](#).

Potete pensare a questi diagrammi come a una sorta di **mappa** che vi guida alla comprensione logica del vostro problema, e quindi del vostro programma.

In questo esempio di diagramma di flusso vengono illustrati alcuni possibili passi da compiere se ci si rende conto che una lampada in casa propria non funziona più.



Per leggere questo diagramma si parte dal punto di inizio rappresentato da un'ellisse, si seguono le frecce fino a che non si arriva alla fine dello schema, anch'esso rappresentato come ellisse, e si possono prendere strade diverse a seconda del verificarsi o meno di condizioni diverse: queste sono racchiuse all'interno di rombi, mentre le azioni da compiere sono racchiuse in rettangoli.

Mediante questo schema possiamo comprendere i vari step a nostra disposizione riducendo eventuali ambiguità.

Ora che abbiamo spiegato che metodologia utilizzare per "pensare da developer", possiamo iniziare a pensare di **rappresentare questi SI e NO** all'interno dei nostri programmi in Python, e per fare ciò parleremo ora di **Logica Booleana**.

Logica Booleana

L'Algebra di Boole è quel ramo dell'Algebra in cui non si utilizzano numeri ma solamente due valori detti Valori di Verità (proprio i Vero e Falso che usiamo nei nostri diagrammi) e che mediante i suoi operatori permette di effettuare calcoli logici.

Valori Booleani di Python: True e False

Gli altri tipi di dato di cui abbiamo parlato nei video precedenti, Interi, Float e Stringhe, hanno un numero illimitato di possibili valori, purché questi rispettino le caratteristiche del tipo: Il tipo di dato booleano invece come abbiamo detto, ha solo due valori possibili, e questi sono **True** e **False**, in italiano **Vero** e **Falso**, che rappresentano sostanzialmente i valori binari **0** ed **1**.

True

False

Supponiamo di essere alle prese con la scrittura di un algoritmo che gestisca l'ingresso di automobili all'interno di un garage; per fare in modo che le macchine possano entrare, il cancello dovrà essere aperto.

In questo caso potremmo avere una variabile chiamata cancello, a cui assegniamo il valore *True* qualora il cancello sia aperto, permettendo così alle macchine di entrare, oppure *False* in caso contrario:

```
cancello = True
```

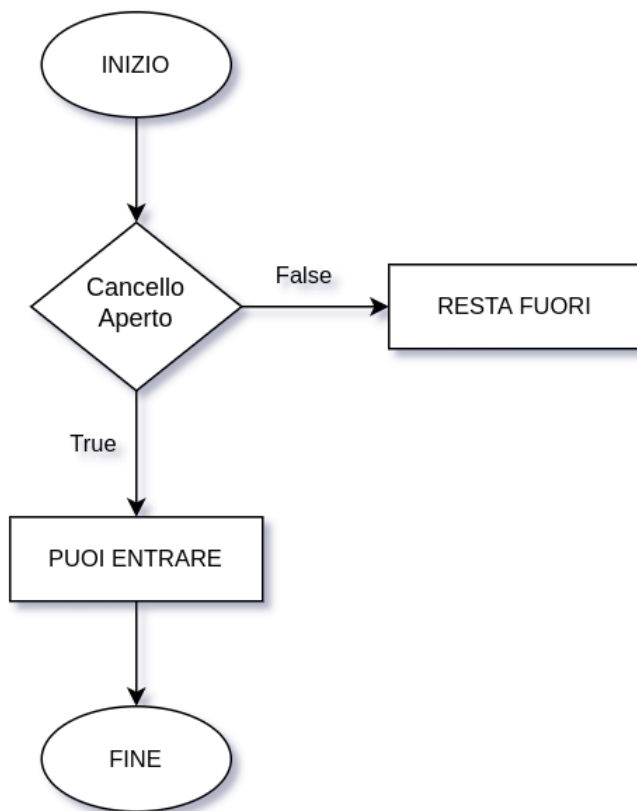
oppure

```
cancello = False
```

```
>>> type(cancello)
```

```
class 'bool'
```

True e *False* fanno parte del "sistema" utilizzato in Python per esprimere in codice i vari SI e NO dei diagrammi di flusso:



Operatori di Comparazione in Python

Sono operatori che servono a comparare valori tra di loro, e che ci restituiscono sempre risultati di natura booleana, quindi *True* o *False*. Se affermassi che 5 è uguale a 5, potremo dire che quest'affermazione è veritiera, quindi *True*, mentre se affermassi che 5 è uguale a 6, l'affermazione sarebbe *False*

Gli operatori di comparazione sono 6:

OPERATORE	SIGNIFICATO
==	Uguale a
!=	Non uguale a
<	Minore di
>	Maggiore di
<=	Minore o uguale a
>=	Maggiore o uguale a

Un'importante distinzione da fare è quella tra `=` e `==`: il primo è il simbolo del processo di assegnazione (che utilizziamo cioè per assegnare un valore a una [variabile](#)), mentre il secondo è l'operatore di comparazione dell'uguaglianza.

Facciamo alcuni esempi di utilizzo di questi operatori nella Shell Interattiva, in modo da rendere più vivido il concetto e mettere a fuoco le idee

```
>>> 5 == 5
True
>>> 5 == 6
False
>>> 23 < 50
True
>>> "ciao" != "arrivederci"
True
>>> x = 18
>>> x >= 16
True
>>> 33 = "33"
False
```

Operatori Booleani in Python: and, or, not

Come nell'algebra classica abbiamo operatori quali `+` e `-`, nell'Algebra Booleana esistono 3 **operatori booleani**, e questi sono **AND**, **OR**, **NOT**. Gli operatori booleani sono utilissimi perché ci consentono di creare espressioni ben più articolate e descrittive di un semplice `5 == 5`. Vengono utilizzati in combinazioni con gli operatori di comparazione appena descritti. Andiamo ad analizzarne il comportamento tramite quelle che vengono chiamate **Tabelle di Verità**, che rappresentano tutto le possibili combinazioni di questi operatori e relativi risultati.

Nota: nelle tabelle di verità tutte le lettere degli operatori sono maiuscole, ma la sintassi corretta dei vari operatori in Python prevede l'uso di sole lettere minuscole (`and`, `or`, `not`)

Operatore and

Espressione	Risultato
True AND True	True
True AND False	False
False AND True	False
False AND False	False

La logica dietro **and** è che affinché un'espressione con questo operatore dia come risultato *True*, entrambe le parti dell'espressione devono risultare veritiere, in caso contrario otterremo sempre un *False*. Tenendo un occhio sulla tabella di verità di **and**, scriviamo sulla Shell degli esempi concreti utilizzando anche gli **operatori di comparazione**:

```
>>> 21 > 1 and 3 < 5
True
>>> 22 == 22 and 1 > 2
False
>>> 2 < 1 and "asd" == "asd"
False
>>> 23 == 15 and 33 != 33
False
```

Operatore or

+-----+-----+	
Espressione	Risultato
+=====+	
True OR True	True
+-----+-----+	
True OR False	True
+-----+-----+	
False OR True	True
+-----+-----+	
False OR False	False
+-----+-----+	

Nel caso di **or**, affinché il risultato sia *True* almeno una delle due comparazioni deve restituire *True* e chiaramente, tutti i casi risultano veritieri eccetto l'ultimo, in cui entrambi i valori sono falsi.

Facciamo degli esempi sulla Shell Interattiva per fissare il concetto:

```
>>> 25 >= 25 or 23 <= 25
True
>>> "io" == "io" or "io" == "robot"
True
>>> 1 != 1 or 1 == 1
True
>>> 4 == 5 or 5 == 6
False
```

Operatore not

Parliamo ora dell'ultimo operatore booleano, il **not**, forse il più semplice da capire.

+-----+	+-----+
Espressione Risultato	
+=====+	+=====+
NOT True False	
+-----+	+-----+
NOT False True	
+-----+	+-----+

Se una comparazione risulta non *True* sarà chiaramente *False*, e viceversa.

```
>>> not "io" == "robot"
```

```
True
```

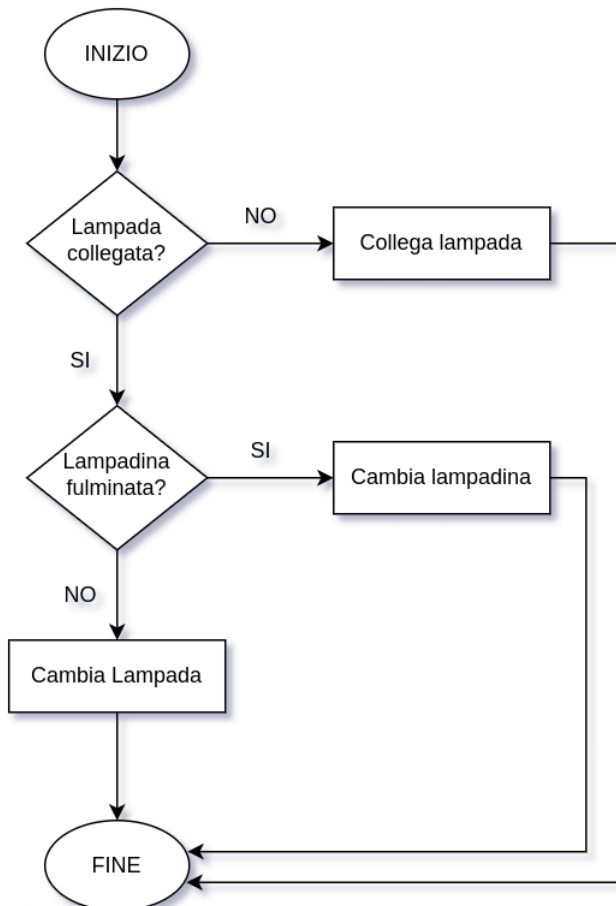
```
>>> not 3 == 3
```

```
False
```

06. Istruzioni if, elif ed else

In questa lezione continueremo il discorso iniziato in precedenza sul controllo del flusso parlando di tre istruzioni (in inglese **statement**) fondamentali: **if**, **elif** ed **else**.

Direi di ripartire dal diagramma di flusso analizzato nella [lezione precedente](#):



Come ricorderete si tratta di uno schema che illustra un insieme di possibili azioni che si possono prendere rendendosi conto che una lampada di casa propria non funziona. Abbiamo imparato che i Si e No del diagramma sono rappresentati dai due valori *True* e *False* del tipo di dato booleano.

In questa lezione andremo a fondo nella tana del Bianconiglio focalizzandoci su come sia possibile delineare i diversi blocchi di controllo delle condizioni. Se finora avete pensato ai diagrammi di flusso come a delle mappe del vostro programma, potete immaginare ciò che spiegheremo ora come delle specie di incroci stradali su queste mappe.

La prima istruzione di controllo che analizzeremo è l'istruzione **if**.

L'istruzione if in Python

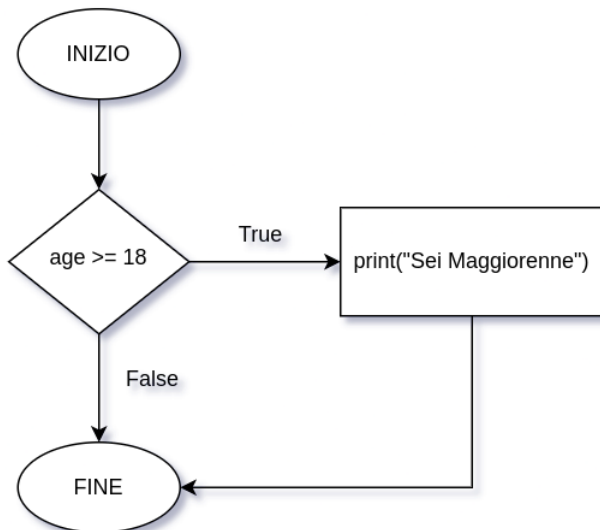
Aprirete **Visual Studio Code** (o il vostro editor di scelta) e aggiungete questo codice in un nuovo file, salvandolo con estensione `.py` in maniera analoga a quanto fatto nella lezione dove abbiamo scritto il nostro primo programma.

```
age = 26
if age >= 18:
    print("Sei maggiorenne")
```

Eseguiamo lo script da terminale col comando `python nome_file.py` ed ecco che in output otteniamo:

Sei Maggiorenne

Analizziamo da vicino i dettagli del funzionamento del nostro codice tramite la lettura del suo diagramma di flusso:



In Italiano, **if** si traduce con **se**.

Se la condizione `age >= 18` risulta veritiera, quindi se il risultato di questa espressione restituisce valore booleano `True`, il programma esegue questa porzione di codice e restituisce la stringa "Sei Maggiorenne". Se invece la condizione fosse risultata falsa, quindi se alla variabile `age` fosse stato assegnato un valore numerico inferiore a 18, l'esecuzione sarebbe saltata direttamente alla fine del programma, ignorando questa porzione di codice.

Verifichiamo questa possibilità andando ad alterare la variabile `age`, assegnandole ora il valore 16:

```
age = 16
if age >= 18:
    print("Sei maggiorenne")
```

Lanciamo nuovamente lo script.

Python ignora ora la porzione di codice col `print('Sei maggiorenne')` saltando direttamente alla fine (senza quindi restituirci nulla in output!) perché, ripetiamolo, la condizione di controllo `age >= 18` ha restituito valore booleano **False**

L'indentazione in Python è obbligatoria

In Python, i blocchi di codice vengono delineati tramite quella che in programmazione chiamiamo **indentazione**, ovvero l'aggiunta di spazi (tipicamente 4) o tab (pressioni del tasto corrispondente nella tastiera!) come delimitatore della nuova porzione di codice.

È importante specificare che a differenza di altri linguaggi di programmazione l'indentazione in Python è **obbligatoria**, e non rispettarla comporta la comparsa di errori.

Modifichiamo ora il codice precedente togliendo lo spazio davanti alla chiamata alla funzione **print()**:

```
age = 16
if age >= 18:
print("Sei maggiorenne")
```

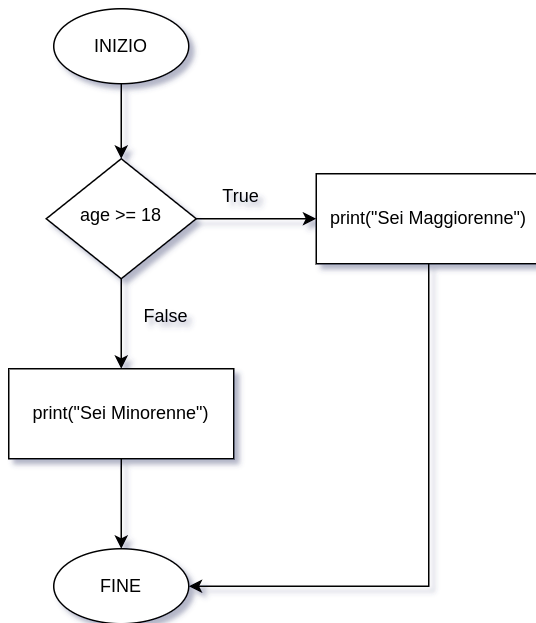
Provate ad eseguire: come noterete Python si rifiuta di eseguire il codice perché non correttamente indentato: non rispetta cioè i suoi standard di logica!

Curiosità: nonostante Python sia un linguaggio molto permissivo nella scelta delle "unità" di indentazione utilizzabili, richiedendo come requisito fondamentale sostanzialmente solo il mantenimento di uno stile uniforme (potete usare anche 1 spazio, 2 spazi ecc), [le linee guida per lo stile ufficiali del linguaggio definite nel documento PEP8](#) prevedono l'utilizzo di indentazione a blocchi di 4 spazi. Tuttavia oggi, l'impiego di tab è estremamente comune e anche tutti gli esempi di questo sito sono indentati in questo modo.

L'istruzione else in Python

Come fare ad eseguire del codice selezionato qualora la condizione di controllo restituisca invece False?

Per questo scopo ci viene in aiuto l'istruzione **else**, traducibile in Italiano con **altrimenti** o **diversamente**.



Ora, se l'espressione restituisce un valore booleano *False*, viene eseguito un altro blocco di codice prima che il programma finisca. Il blocco else è sempre l'ultimo ad essere analizzato e eseguito solo se tutte le condizioni precedenti ad else sono risultate false. Modifichiamo il nostro codice in maniera appropriata:

```
age = 16
if age >= 18:
    print("Sei maggiorenne")
else:
    print("Sei minorenne")
```

Eseguiamo lo script:

Sei minorenne

...e come vedete, ora ci viene restituita la stringa "Sei minorenne" prima che il programma finisca!

L'istruzione elif in Python

Ci saranno dei momenti in cui i vostri programmi dovranno essere pronti a poter fronteggiare più di due possibili strade, come un incrocio stradale un po' più complesso.

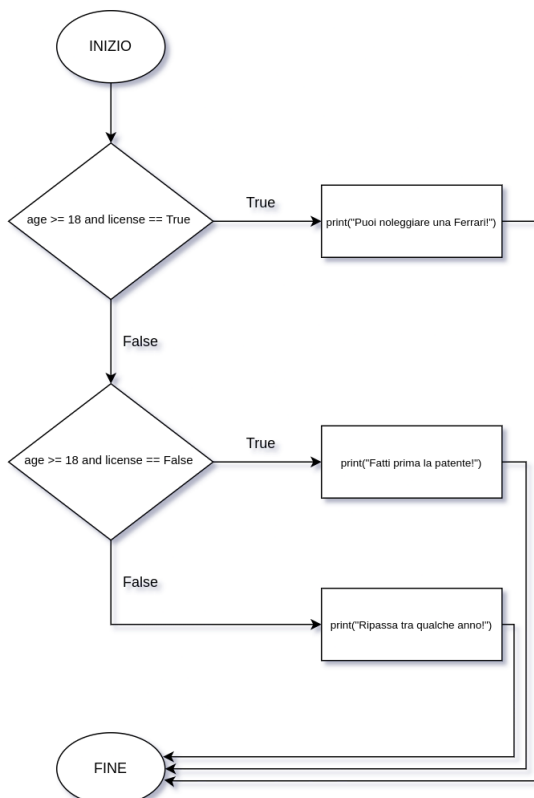
Per fare questo entra in gioco l'istruzione **elif**, che significa **ELse IF**, ovvero **altrimenti se** che come per magia vi permette di aggiungere tanti possibili blocchi di controllo quanti ne avete bisogno.

L'istruzione **elif** viene eseguita solo se la sua espressione di controllo restituisce *True* e l'**if** o eventuali **elif** precedenti hanno restituito *False*.

Ad esempio, rendiamo un po' più complesso il nostro programma, aggiungendo una variabile *license* (patente) e l'operatore booleano **and**:

```
age = 18
license = False
if age >= 18 and license == True:
    print("Puoi noleggiare una Ferrari!")
elif age >= 18 and license == False:
    print("Fatti prima la patente!")
else:
    print("Ripassa tra qualche anno!")
```

In accordo con questo esempio, il diagramma di flusso potrebbe essere simile a questo:



L'istruzione **pass**

L'istruzione **pass** viene utilizzata quando è necessario specificare un'istruzione in un determinato punto del codice, ma non si vuole eseguire alcuna operazione. Ad esempio, può essere utilizzata come **segnaposto** quando si sta lavorando su una struttura di controllo del flusso (come un'istruzione **if**) e si desidera evitare un errore di sintassi. Ecco un esempio di come utilizzarla:

```
if x < 0:  
    pass # Qui potrebbe essere inserito del codice in futuro  
else:  
    print("x è maggiore o uguale a 0")
```

In questo caso, se x è minore di zero, l'istruzione **pass** viene eseguita e il flusso del programma continua senza fare nulla. Se invece x è maggiore o uguale a zero, viene stampato il messaggio.

L'istruzione si può utilizzare in qualsiasi contesto, anche nei loop o come segnaposto quando si definisce una classe o una funzione ma se si vuole implementare in futuro lasciando vuoto il blocco di codice.

07. Ciclo while, istruzioni break e continue

Anche questa è una puntata dedicata alla gestione del Flusso: introdurremo infatti uno dei due cicli di Python, il **ciclo while**, e due nuove istruzioni associate ai cicli: **break** e **continue**.

In programmazione, un **loop** (o **ciclo**) è un costrutto che permette di ripetere una o più istruzioni per un certo numero di volte o finché permane una determinata condizione.

L'Istruzione while in Python

Il nome **while loop** racchiude in sé la spiegazione del suo funzionamento e il perché della sua esistenza.

La parola **while** può infatti essere tradotta in italiano come **finché** o **mentre**, e il ciclo while ci permette proprio **di eseguire un blocco di codice finché una determinata condizione è e resta True**.

Facciamo subito un esempio di codice per prendere familiarità con la sintassi e chiarirci le idee:

```
counter = 0
while counter <= 10:
    print(counter)
    counter += 1
```

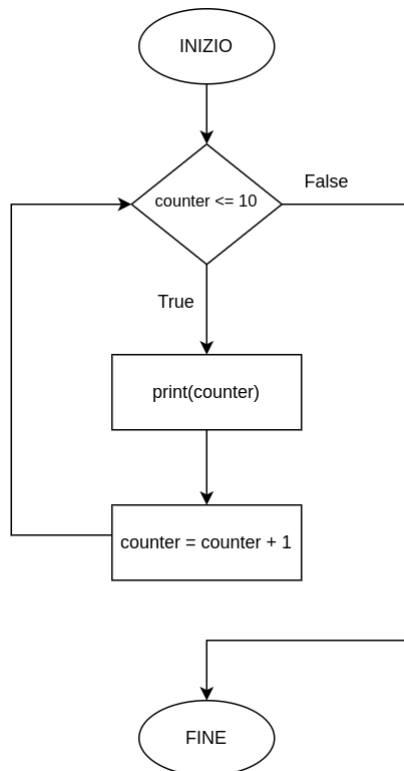
Nota: il codice `counter += 1` è equivalente a `counter = counter + 1`, e ci permette di riassegnare alla variabile `counter` il valore che aveva in precedenza + 1.

Eseguiamo il codice:

```
# output:
```

```
0
1
2
3
4
5
6
7
8
9
10
```

Che è successo? Diamo uno sguardo al diagramma di flusso:



L'istruzione **while** controlla se alla variabile *counter* è associato un valore minore o uguale a 10, e visto che la condizione risulta inizialmente veritiera, il programma entra nel **loop**, mandando in print la variabile *counter*, e quindi sommandoci 1.

Questo comportamento si ripete finché la condizione *counter* <= 10 risulta *True*, cioè fino a che nel contatore non è presente un valore superiore a 10, caso in cui il ciclo **while** verrà interrotto e il codice associato non verrà più eseguito. La differenza con quanto visto nella lezione precedente in merito all'istruzione **if** è quindi abbastanza evidente.

Modifichiamo il codice sostituendo **while** con **if** ed eseguiamolo nuovamente:

```

counter = 0
if counter <= 10:
    print(counter)
    counter += 1
  
```

output:

0

Ecco che il programma si comporta come ci si poteva aspettare: il codice interno al blocco viene eseguito un'unica volta, e il programma quindi termina.

Cicli while Infiniti

Ma cosa succederebbe qualora la condizione di controllo restasse perennemente *True*? In questo caso otterremmo un un **loop infinito**. Proviamo ad innescarne uno:

```
while 15 == 15:  
    print("dentro al ciclo!!!")
```

Eseguiamo:

```
# output:  
dentro al ciclo!!!  
dentro al ciclo!!!  
dentro al ciclo!!!  
dentro al ciclo!!!  
dentro al ciclo!!!  
dentro al ciclo!!!  
dentro al ciclo!!!  
dentro al ciclo!!!  
dentro al ciclo!!!  
dentro al ciclo!!!  
dentro al ciclo!!!  
dentro al ciclo!!!  
dentro al ciclo!!!  
dentro al ciclo!!!  
dentro al ciclo!!!  
(...)
```

Per quanto questo **infinite loop** non sia un errore del linguaggio di programmazione ma anzi, possa dimostrarsi estremamente utile, qualora vi troviate ad averne innescato uno per errore, volendo bloccarlo vi basterà cliccare **CTRL-C**.

Per rendere il tutto ancora più esplicito possiamo volendo modificare il codice scritto in precedenza usando una variabile inizializzata usando il booleano *True*.

```
counter = 0  
run = True  
while run == True:  
    print(counter)  
    counter += 1
```

Notiamo inoltre come in questo caso avremmo potuto semplificare il nostro codice escludendo l'operatore "uguale a", in quanto stiamo facendo un controllo su un Booleano!

Per tanto possiamo fare:

```
counter = 0  
  
run = True  
  
while run:  
    print(counter)  
    counter += 1
```

Vi invito a scoprire da voi l'output di questo codice!

Per proseguire, come abbiamo accennato a inizio lezione, ci sono due istruzioni correlate al ciclo **while** di cui andremo ora a parlare, e sono l'istruzione **break** e l'istruzione **continue**.

L'istruzione **break** in Python

L'istruzione **break** serve per terminare un ciclo **while** prematuramente: non appena quest'espressione viene letta e processata all'interno del ciclo, Python **blocca il loop istantaneamente**.

Modifichiamo il codice usato in precedenza per usare anche l'istruzione **break**:

```
run = True  
  
stop = 1000  
  
counter = 0  
  
while run == True:  
    print(counter)  
    counter += 1  
  
    if counter > stop:  
        print("Sto uscendo dal loop...")  
        break
```

Una volta che nel contatore è presente il numero 1000, quando al loop successivo viene aggiunto un 1 portandolo ad 1001, la condizione relativa all'istruzione **if** diventa *True* e viene così processato il **break**, che blocca il ciclo infinito all'istante.

L'istruzione **continue** in Python

L'istruzione **continue** invece è simile a **break**, ma invece di interrompere il ciclo fa saltare tutto il codice dopo l'istruzione e **fa ripartire Python dalla prima riga del ciclo**.

Facciamo un esempio:

```
run = True
skip = 5
counter = 0
while counter < 10:
    counter += 1
    if counter == skip:
        print("Sto saltando " + str(skip))
        continue
    print(counter)
```

Quindi, *counter* sempre inizializzato a 0, a inizio ciclo viene incrementato, se però risulta uguale a 3, allora non viene passato a **print()** ma il ciclo si ripete dall'inizio. Quando in *counter* è presente 9, il valore viene incrementato portandolo a 10, e una volta mandato in output, il ciclo non risulta più *True* e il programma si ferma.

output:

```
1
2
3
4
Sto saltando 5
6
7
8
9
10
```

08. Il Ciclo for e la funzione range

In questa lezione parleremo del secondo **loop** (o **ciclo**) di Python, il ciclo **for**, e introdurremo una nuova funzione molto importante, la funzione **range**.

Il Ciclo for in Python

Il **for loop** viene utilizzato in maniera simile al **while loop** di cui abbiamo parlato nella lezione precedente: essendo entrambi cicli, permettono di ripetere l'esecuzione di una determinata porzione di codice più volte. Tuttavia, le differenze nei meccanismi di funzionamento ne rendono preferibile uno rispetto all'altro a seconda del contesto in cui ci si trova.

A livello logico, la differenza col ciclo **while** è che invece di eseguire il blocco di codice interessato finché la condizione di controllo resta *True*, il ciclo **for** esegue questa sezione di programma per un numero specifico di cicli.

Qui di seguito uno dei programmi che abbiamo scritto nella lezione precedente, che come ricorderete da in output i numeri da 0 a 10:

```
counter = 0
while counter <= 10:
    print(counter)
    counter += 1
```

Riassumendone brevemente il funzionamento, abbiamo una variabile contatore inizializzata a 0, e fintato che il valore associato alla variabile risulta minore o uguale a 10, viene ripetuto un ciclo di codice in cui viene mandato in output il suo contenuto, sommandoci poi 1 prima di ripetere il loop.

Ma cosa mi direste se vi dicessi che possiamo ottenere il medesimo risultato più velocemente utilizzando stavolta il ciclo **for** di Python?

Per più velocemente intendo che possiamo eliminare sia la variabile contatore che la riga di codice necessaria ad incrementarne il valore: questo perché come abbiamo detto, la particolarità del **for** loop è che viene eseguito un numero definito di volte!

```
for numero in range(11):
    print(numero)
```

Eseguiamo nuovamente il codice:

output:

```
0
1
2
3
4
```

5
6
7
8
9
10

Vi starete forse chiedendo: com'è possibile che quando usiamo il `for` con la variabile `numero`, essa può essere riconosciuta automaticamente se non è stata prima definita?

In Python, quando si usa un ciclo `for`, la variabile specificata viene inizializzata dal ciclo stesso. Ad ogni iterazione del ciclo, questa assume il valore relativo nell'iterabile.

In altre parole, non è necessario definire o inizializzare la variabile `numero` prima del ciclo `for`. Il ciclo `for` si occupa sia della creazione che dell'assegnazione del valore alla variabile ad ogni passo.

La funzione `range` in Python

Abbiamo ottenuto lo stesso risultato ottenuto col `while` loop, ma utilizzando stavolta solo 2 righe di codice invece di 4. Eppure sono certo che in molti tra voi si staranno chiedendo: cos'è quella funzione `range()` e come mai le ho passato 11 invece di 10? E soprattutto, come funziona esattamente il ciclo `for`?

`for numero in range(11)`

↑
parola chiave

↑
nome variabile
(scelto da voi)

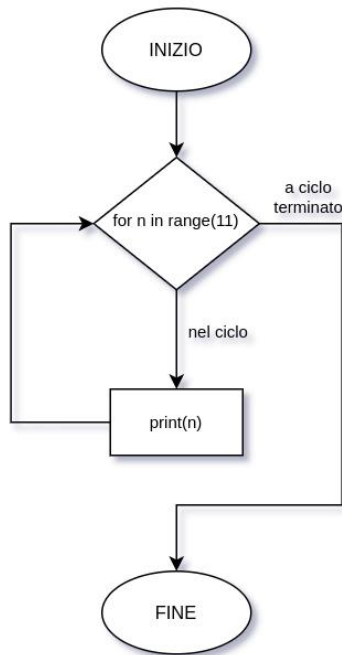
↑
parola chiave

↑
chiamata alla
funzione `range()`

La funzione `range()` ci permette di impostare un intervallo di esecuzione tanto ampio quanto il numero che le passiamo come parametro, senza però includere il numero passato in sé. Di default, la funzione `range()` di Python inizia a contare a partire da 0; per comodità potete pensare a questo intervallo come a una sorta di lista, in questo caso di 11 elementi, da 0 a 10 inclusi.

Come abbiamo detto, a differenza del ciclo `while`, il ciclo `for` esegue la sua porzione di codice un numero preimpostato di volte, giusto? L'intervallo di `range()` corrisponde al numero di iterazioni che verranno eseguite, ovvero al numero di volte che il `for` loop verrà processato.

Di seguito un diagramma di flusso per il ciclo `for` in cui viene illustrato il funzionamento del codice scritto in precedenza:



Una volta che si entra all'interno del loop, il codice indentato relativo viene eseguito in loop fino a che non si arriva alla fine dell'intervallo di **range()**, momento in cui il ciclo termina.

Nota: in questo esempio si è scelto di utilizzare *n* come nome della variabile invece di *numero*, ma il funzionamento resta lo stesso. Altro nome comuni in questi casi sono *i* (da index, o indice) e l'underscore *_*, usato spesso quando non si dà molta importanza alla variabile in sé.

I parametri della funzione range: start, stop e step

Ora che abbiamo imparato come funziona un ciclo for e a cosa serve la funzione **range()** di Python, vediamo un po' come poter ottenere il massimo da questa, in quanto dovete sapere che **possiamo passarle ben 3 parametri**.

Come impostare il comportamento di range() mediante 3 parametri



Questo ci permette di impostare oltre a un punto di inizio dell'intervallo anche un punto di fine personalizzato, e un passo di avanzamento.

Modifichiamo il **for** loop utilizzato prima: impostiamo un punto di inizio a 3, manteniamo il punto di fine come 11, e settiamo anche un passo di 2

```
for numero in range(3, 11, 2):  
    print(numero)
```

output

3
5
7
9

La differenza più grande è che stavolta il passo dell'intervallo è 2, quindi si passa da 3 a 5, da 5 a 7 e da 7 a 9. Come al solito il numero finale dell'intervallo viene saltato.

Modifichiamo ora il nostro codice in modo da invertire l'ordine dei valori restituiti rispetto all'esempio precedente, in modo quindi da ottenere i numeri da 10 a 0. Come punto di inizio dell'intervallo impostiamo 10, come valore finale -1 (ricordando che l'ultimo valore non viene considerato!) e come passo di avanzamento -1:

```
for numero in range(10, -1, -1):  
    print(numero)
```

output

10
9
8
7
6
5
4
3
2
1
0

09. I Moduli della Standard Library

In questa lezione parleremo della **Standard Library** di Python.

La Standard Library di Python

Finora abbiamo visto che ciascun programma che scriviamo può usare una serie di funzioni integrate di Python, come ad esempio la funzione **print()** e la funzione **input()**. In aggiunta a queste funzioni Python dispone di una collezione di **moduli** pronti all'uso in quella che viene chiamata la **Standard Library**.

Ciascun modulo è come una libreria (un insieme) di funzioni scritte da altri developer molto esperti, realizzate per svolgere compiti comuni come ad esempio la generazione di numeri casuali nel caso del modulo **random** o funzioni matematiche come nel caso del modulo **math**, interfacciamento col sistema operativo nel caso del modulo **os** e tanto altro ancora.

La Standard Library di Python è davvero molto potente e versatile: date uno sguardo a [questa pagina](#) del sito della Python Foundation per farvi un'idea del suo potenziale!

Nota: Questa trascrizione include degli approfondimenti e delle aggiunte per fornire maggiori dettagli sui moduli più utilizzati e sui loro metodi. Abbiamo deciso di integrare tali informazioni per arricchire il contenuto: in particolare, abbiamo fornito esempi di utilizzo dei metodi e spiegazioni dettagliate sulla sintassi e sul funzionamento dei moduli più importanti. Speriamo che le integrazioni rendano questa lezione ancora più utile!

Vedremo alcuni dei moduli della standard library più utili e più semplici da utilizzare:

- [random](#)
- [math](#)
- [datetime](#)
- [platform](#)
- [tkinter](#)
- [turtle](#)
- [os](#)
- [shutil](#)
- [re](#)
- [zipfile](#)
- [json](#)

Prima di potere utilizzare le funzioni, le classi e i metodi di questi moduli all'interno dei nostri programmi dovremo importarli, vediamo come!

Come importare un modulo

Ci sono tre diversi modi per importare un modulo nei nostri script di Python.

Il primo consiste semplicemente nel chiamare il modulo tramite il comando **import**. Con questo tipo di import potete accedere alle funzioni e alle classi del modulo nel vostro codice utilizzando la **dot notation**: *nome_modulo.nomeoggetto*.

```
import math
```

Il secondo consiste nell'importare solo specifiche classi, variabili e funzioni dal modulo utilizzando **from nome_modulo import nomeoggetto**: in tal caso possiamo utilizzare ciò che abbiamo importato chiamando gli oggetti direttamente per nome senza far ricorso alla dot notation.

```
from math import sqrt, exp
```

Il terzo consiste nel richiamare tutte le funzioni, le classi e le variabili incluse nel modulo tramite l'asterisco, e anche in questo caso non sarà necessario riscrivere il nome del modulo.

```
from math import *
```

Se state pensando di lavorare ad un progetto importante e articolato, è decisamente meglio utilizzare la prima o la seconda modalità che vi ho spiegato per mantenere il codice più ordinato e sicuro e sapere esattamente cosa si sta utilizzando in ogni file del nostro codebase.

Esiste inoltre un sistema che ci consente di installare moduli che non fanno parte della Standard Library, ampliando ulteriormente l'insieme di cose che è possibile fare, ma parleremo di ciò più avanti nel corso.

Il modulo random

Questo modulo fornisce una serie di funzioni per generare **numeri pseudo-casuali**. I numeri ottenuti con **random** possono essere utili in molte situazioni, per esempio quando è necessario simulare eventi, testare algoritmi o selezionare elementi casuali da una lista. Vediamo alcuni dei metodi di questo modulo:

Il metodo random()

Si tratta della funzione base del modulo, da cui dipendono quasi tutte le altre: utilizza l'algoritmo [Mersenne Twister](#) per generare un numero casuale di tipo *float* compreso tra 0 e 1:

```
random.random()
```

```
# output
```

```
0.5576628296719214
```

Il metodo `randint(a, b)`

Genera un numero intero casuale compreso tra i valori *a* e *b* (con *a* e *b* inclusi).

```
# questo programma manda in output 10 numeri pseudo-casuali tra 1 e 50
import random
for numero in range(10):
    val = random.randint(1, 50)
    print(val)
# output
7
27
22
9
19
11
31
4
36
32
```

Il metodo `randrange(start, stop[, step])`

Genera un numero intero casuale compreso tra *start* e *stop*, con *stop* escluso. Se si specifica anche il terzo parametro *step*, i numeri generati saranno suoi multipli.

```
# Genera solo numeri interi pari
random.randrange(0, 10, 2)

# Genera solo numeri interi dispari
random.randrange(1, 10, 2)
```

Il metodo `choice(seq)`

Seleziona un elemento casuale dalla sequenza *seq*, che può essere una lista, una tupla o una stringa.

```
lista = ['Io', 'Ganimede', 'Callisto', 'Europa']
random.choice(lista)
```

```
# output
```

```
'Europa'
```

Il modulo math

Questo modulo è utilizzato per **eseguire operazioni matematiche** e la maggioranza delle sue funzioni restituiscono valori di tipo *float*. Vediamo alcune delle funzioni più utilizzate di **math**:

Il metodo sqrt(x)

Restituisce la **radice quadrata** di x:

```
math.sqrt(25)
```

```
# output
```

```
5.0
```

Il metodo pow(x, y)

Restituisce x elevato alla **potenza** di y:

```
math.pow(10, 2)
```

```
# output
```

```
100.0
```

Il metodo exp(x)

Restituisce l'**esponenziale** di x, cioè e elevato x:

```
math.exp(x)
```

```
# output
```

```
7.38905609893065
```

I metodi ceil(x) e floor(x)

La prima restituisce l'**arrotondamento per eccesso** di x (il più piccolo numero intero maggiore o uguale a x), la seconda restituisce l'**arrotondamento per difetto** di x (il più grande numero intero minore o uguale a x).

```
math.ceil(33.23)
```

```
# output
```

```
34
```

```
math.floor(23.532)
```

```
# output
```

```
23
```

I metodi `sin(x)`, `cos(x)` e `tan(x)`

Restituiscono rispettivamente il **seno**, il **coseno** e la **tangente** di x in radianti.

```
math.sin(1.5707963267948966)
# output
1.0
math.cos(0)
# output
1.0
math.tan(0.7853981633974483)
# output
0.9999999999999999
```

I metodi `degrees(x)` e `radians(x)`

Convertono l'angolo x da **gradi** a **radianti** e viceversa.

```
math.degrees(1.0471975511965976)
# output
59.99999999999999
math.radians(45)
# output
0.7853981633974483
```

Il modulo `datetime`

In Python il tempo non rappresenta un tipo specifico di dato: questo modulo **permette di creare oggetti per rappresentare data e ora** e fornisce tutta una serie di metodi per effettuare con esse varie operazioni, come convertire gli oggetti in stringhe o calcolare durate. Alcune delle classi più utilizzate di **`datetime`** sono le seguenti:

La classe `date()`

Restituisce un oggetto **`date`** che rappresenta una data specificata dall'utente. Come parametri opzionali accetta l'anno, il mese e il giorno:

```
d = datetime.date(2027, 1, 1)
print(f"type(d): {d}")
# output
<class 'datetime.date': 2027-01-01
```

La classe `time()`

Restituisce un oggetto **time** che rappresenta un orario specificato dall'utente. Come parametri opzionali accetta l'ora, il minuto, il secondo e il microsecondo:

```
t = datetime.time(8, 57, 15)
print(f"{type(t)}: {t}")
# output
<class 'datetime.time': 08:57:15
```

La classe `datetime()`

Restituisce un oggetto **datetime** che rappresenta una data e un orario specificati dall'utente. Come parametri opzionali accetta l'anno, il mese, il giorno, l'ora, il minuto, il secondo e il microsecondo:

```
dt = datetime.datetime(2025, 5, 25, 1, 1, 1)
print(f"{type(dt)}: {dt}")
# output
<class 'datetime.datetime': 2025-05-25 01:01:01
```

La classe `timedelta()`

Restituisce un oggetto **timedelta** che rappresenta una durata di tempo. Come parametri opzionali accetta settimane, giorni, ore, minuti, secondi, millisecondi e microsecondi.

```
td = datetime.timedelta(days=5, hours=7, minutes=30, seconds=2)
print(f"{type(td)}: {td}")
# output
<class 'datetime.timedelta': 5 days, 7:30:02
```

Il modulo `platform`

Questo modulo fornisce una serie di metodi per **ottenere informazioni sul sistema operativo**, la macchina e l'ambiente in cui il codice Python viene eseguito. Queste informazioni possono essere utili per personalizzare il comportamento del codice in base all'ambiente di esecuzione.

```
import platform
platform.system()
# output
'Windows'
platform.machine()
# output
```



```
'AMD64'

platform.processor()
# output
'Intel64 Family 6 Model 141 Stepping 1, GenuineIntel'

platform.architecture()
# output
('64bit', 'WindowsPE')

platform.python_version()
# output
'3.10.3'
```

Il modulo tkinter

Questo modulo fornisce un'interfaccia di programmazione per **creare applicazioni grafiche multiplatforma**. La libreria contiene una serie di widget come pulsanti, etichette, caselle di testo, menu e finestre, che possono essere utilizzati per creare interfacce utente interattive.

È possibile creare finestre e widget, impostare il loro aspetto e la loro posizione, gestire gli eventi generati dall'utente e molto altro ancora. Creiamo come esempio una finestra che contiene un semplice pulsante cliccabile:

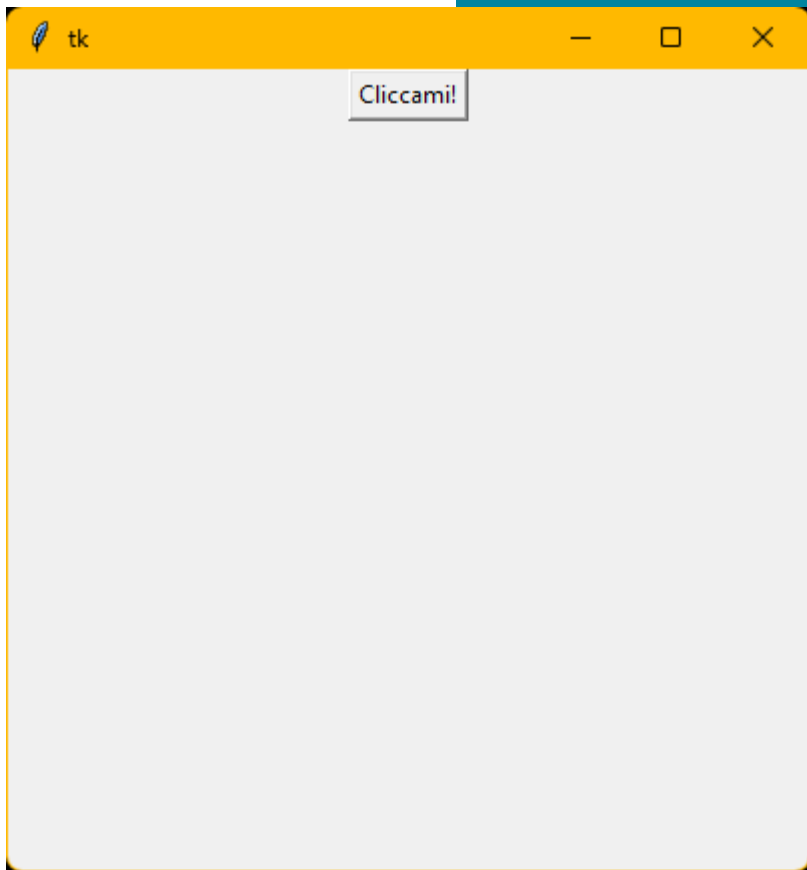
```
import tkinter as tk

root = tk.Tk()

root.geometry("400x400")

button = tk.Button(root, text='Cliccami!')

button.pack()
```

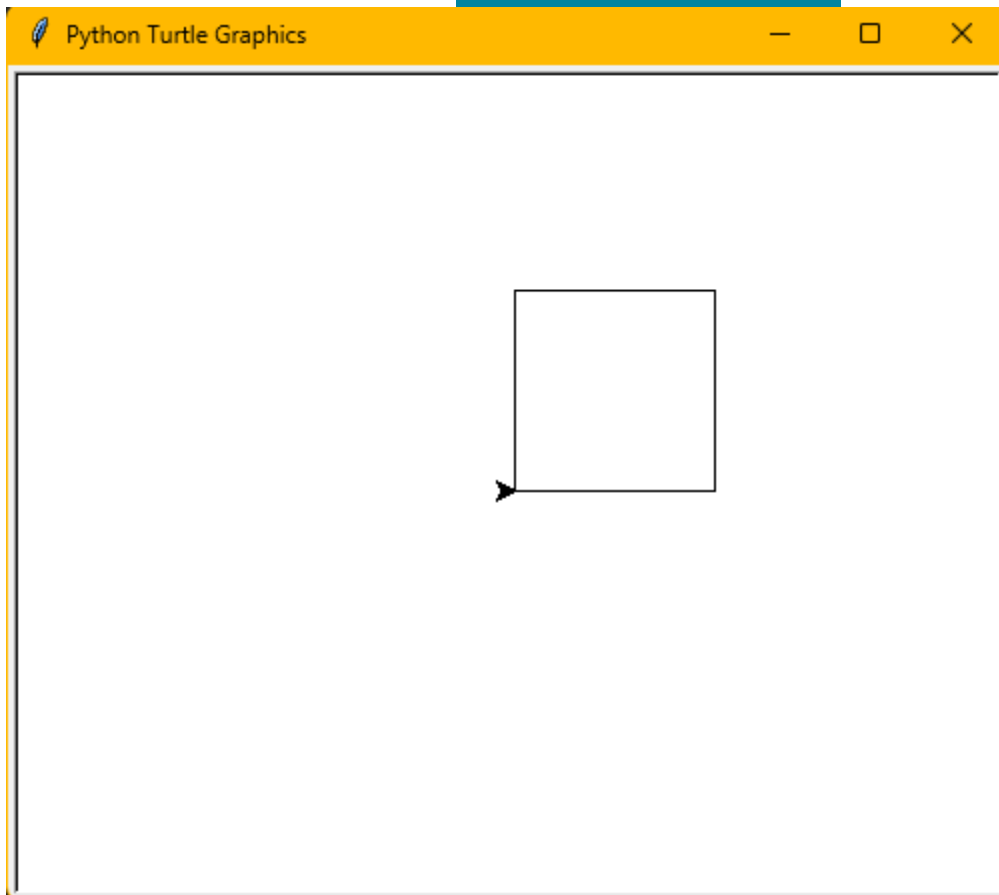


Se volete imparare a creare interfacce grafiche con Tkinter potete seguire la nostra [serie di lezioni dedicate](#).

Il modulo turtle

Questo modulo permette di **creare immagini e disegni grafici** utilizzando una "tartaruga" virtuale controllabile tramite una serie di comandi da inserire nel codice. Ad esempio, se avviate questo codice vedrete che la tartaruga disegnerà un quadrato:

```
import turtle  
  
t = turtle.Turtle()  
  
for i in range(4):  
    t.forward(100)  
    t.left(90)  
  
turtle.done()
```



Il modulo **turtle** è perfetto per imparare i concetti di base della programmazione e della grafica perché permette di sperimentare con diverse forme e colori per creare immagini complesse. Provate ad eseguire questo codice che utilizza la tartaruga per disegnare [curve di Hilbert](#):

```
import turtle

def hilbert_curve(t, order, size, orientation):
    if order == 0:
        return

    t.right(orientation * 90)
    hilbert_curve(t, order - 1, size, -orientation)
    t.forward(size)
    t.left(orientation * 90)
    hilbert_curve(t, order - 1, size, orientation)
    t.forward(size)
    hilbert_curve(t, order - 1, size, orientation)
    t.left(orientation * 90)
    t.forward(size)
```

```
hilbert_curve(t, order - 1, size, -orientation)
```

```
t.right(orientation * 90)
```

```
turtle.setup(width=800, height=800)
```

```
turtle.speed('fastest')
```

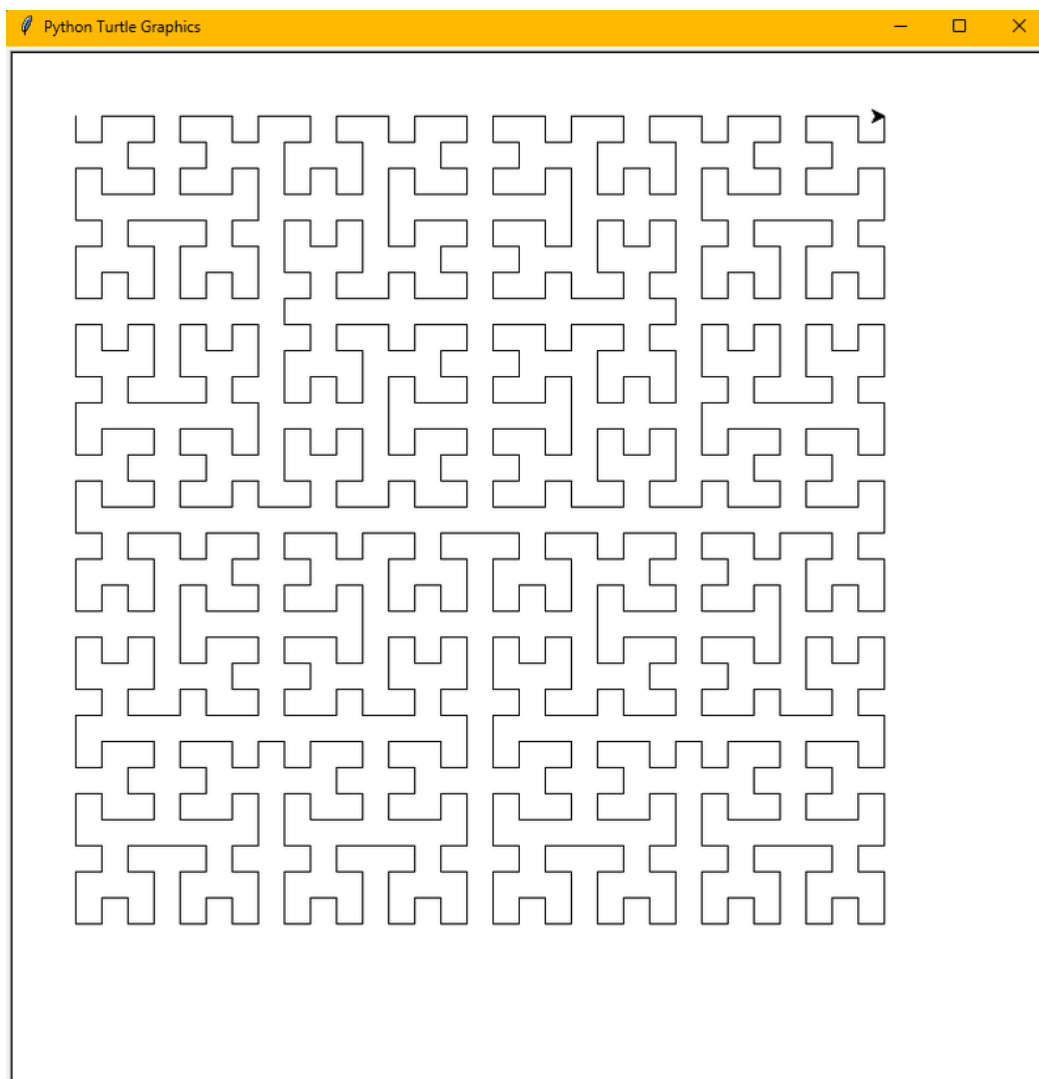
```
turtle.penup()
```

```
turtle.goto(-350, 350)
```

```
turtle.pendown()
```

```
hilbert_curve(turtle, 5, 20, 1)
```

```
turtle.exitonclick()
```



Il modulo os

Viene utilizzato per **interagire con il sistema operativo**. Le sue funzionalità sono tantissime, ma quelle che incontrerete più spesso riguardano la navigazione nelle cartelle del filesystem. Molti metodi di **os** accettano come parametro un percorso (in inglese **path**):

Il metodo listdir(path)

Questo metodo restituisce una lista contenente i nomi di tutti i file e tutte le cartelle della directory specificata dal parametro *path*, in ordine arbitrario:

```
os.listdir('C:\\Users\\Username\\Documents')
```

```
# output
```

```
['Lavoro', 'Testi Canzoni', 'Pdf', 'Nuova Cartella']
```

Per vedere nel dettaglio alcuni dei metodi più utilizzati del modulo **os** potete andare a leggere la serie di [lezioni dedicate alla gestione di file e cartelle](#).

Il modulo shutil

Questo modulo (abbreviazione di **Shell Utility**) fornisce un insieme di funzioni per la **gestione del filesystem** e semplifica molte operazioni come copiare, spostare o eliminare file o directory, archiviare in formato ZIP, ecc. Ne parliamo in modo approfondito durante le [lezioni dedicate alla gestione di file e cartelle](#).

Il modulo re

Questo modulo permette di compiere vari tipi di **operazioni con le espressioni regolari** (**re** è l'acronimo di **regular expression**), ovvero sequenze di caratteri utilizzate per cercare, sostituire e manipolare stringhe di testo.

Tutti i metodi di **re** che vedremo accettano come parametro opzionale anche *flags*, che permette specificare comportamenti aggiuntivi per la funzione, ad esempio **re.IGNORECASE**, che esegue la ricerca senza considerare maiuscole e minuscole, e **re.MULTILINE**, che esegue la ricerca su più righe del testo.

Il metodo search(pattern, string)

Questo metodo cerca la stringa *pattern* all'interno della stringa *string* e restituisce un oggetto **match** corrispondente alla prima occorrenza trovata, dove *span* è una tupla contenente gli indici della posizione iniziale e finale della corrispondenza nella stringa.

```
string = "All work and no play makes Jack a dull boy"
```

```
pattern = "[l]"
```

```
re.search(pattern, string)
```

```
# output
```

```
<re.Match object; span=(1, 2), match='l'>
```

Il metodo `findall(pattern, string)`

Questo metodo cerca la stringa *pattern* all'interno della stringa *string* e restituisce una lista di stringhe.

```
string = "All work and no play makes Jack a dull boy"
```

```
pattern = "jack"
```

```
re.findall(pattern, string, re.IGNORECASE)
```

```
# output
```

```
['Jack']
```

Il metodo `sub(pattern, repl, string)`

Questo metodo sostituisce tutte le occorrenze della stringa *pattern* all'interno della stringa *string* con la stringa *repl*.

```
stringa = "La mia email è mia.mail@gmail.com."
```

```
pattern = r"\b\w+\.\w+@\w+\.\w+\b"
```

```
nuova_stringa = re.sub(pattern, "EMAIL", stringa)
```

```
print(nuova_stringa)
```

```
# output
```

```
La mia email è EMAIL.
```

Il modulo `zipfile`

Questo modulo fornisce diverse funzionalità per la **creazione, la lettura e la modifica di archivi ZIP**, che sono file compressi contenenti uno o più file o directory. Tramite **zipfile** possiamo estrarre i file contenuti nell'file .zip e aggiungerli o rimuoverli senza dover decomprimere e ricreare l'intero archivio. Per imparare ad utilizzare questo modulo potete leggere la [lezione sugli archivi zip](#) nella serie dedicata alla gestione di file e cartelle.

Il modulo `json`

Questo modulo serve per **codificare e decodificare i dati in formato JSON** ([JavaScript Object Notation](#)): si tratta di un formato molto comune utilizzato soprattutto nella [comunicazione tra client e server](#) e come file di configurazione (anche su **VSCode**). La struttura di un oggetto JSON è simile a quella di un dizionario: i dati sono strutturati come coppie **chiave: valore** (dette **proprietà**) circondate da parentesi graffe e separate tra loro da una virgola.

Per aprire i file JSON si utilizza l'[istruzione with](#).

La funzione `load(fp)`

Accetta come argomento un file da cui caricare i dati JSON e restituisce un oggetto Python corrispondente in base alla [tabella di conversione](#):

```
with open('pianeti.json', 'r') as file:
    data = json.load(file)
print(data)
# output
{'pianeta': 'Giove', 'massa terrestre': 317.83, 'satelliti': 95}
```

La funzione loads(s)

Accetta come argomento una stringa *s* contenente dei dati JSON e la converte in un oggetto Python in base alla [tabella di conversione](#):

```
elemento = '{"nome": "Cloro", "simbolo": "Cl", "numero atomico": 17}'
data = json.loads(elemento)
print(data)
# output
{'nome': 'Cloro', 'simbolo': 'Cl', 'numero atomico': 17}
```

La funzione dump(obj, fp)

Accetta come argomento *obj* un oggetto da convertire in formato JSON e come argomento *fp* un file su cui scrivere i dati:

```
data = {'nome': 'Silvia', 'età': 30, 'città': 'Milano'}
with open('data.json', 'w') as file:
    json.dump(data, file)
```

La funzione dumps(obj)

Accetta un oggetto (*obj*) da convertire in formato JSON e restituisce una stringa contenente i dati:

```
data = {'nome': 'Mario', 'età': 45, 'città': 'Roma'}
json_data = json.dumps(data)
print(json_data)
# output
{"nome": "Mario", "et\u00e0": 45, "citt\u00e0": "Roma"}
```

10. Crea una Calcolatrice con Python

In questa lezione faremo il punto della situazione su ciò che abbiamo appreso finora mediante la costruzione di una semplice calcolatrice. Vi farà piacere sapere che se avete seguito questa serie fino a questo punto, disponete ora di tutti gli strumenti necessari per poter scrivere questo programma!

Come creare una calcolatrice in Python

Abbiamo finora appreso come usare varie funzioni e istruzioni proprie del linguaggio. Tra tutto quanto visto finora, cosa utilizzeremo in questa lezione - tutorial?

- Gli operatori numerici di Python e le variabili
- La funzione **print()** e la funzione **input()**
- Le istruzioni di controllo **if**, **elif** ed **else**
- Alcune funzioni di conversione
- Il ciclo **while** e le istruzioni **break** e **continue**

Di seguito potete trovare il codice del programma, e vi invito a seguire il video per una spiegazione passo passo di tutte le istruzioni!

```
# Crea una calcolatrice con Python!
```

```
from math import sqrt
```

```
hello_message = """
```

```
Benvenuti al programma calcolatrice!
```

```
Di seguito un elenco delle varie funzioni disponibili:
```

- Per effettuare un'Addizione, seleziona 1;
- Per effettuare una Sottrazione, seleziona 2;
- Per effettuare una Moltiplicazione, seleziona 3;
- Per effettuare una Divisione, seleziona 4;
- Per effettuare un Calcolo Esponenziale, seleziona 5;
- Per effettuare una Radice Quadrata, seleziona 6;
- Per uscire dal programma puoi digitare ESC;

```
"""
```

```
while True:
```

```
    print(hello_message)
```



```
action = input("Inserisci il numero corrispondete all'operazione da effettuare: ")
```

```
if action == "1":
```

```
    print("\nHai scelto: Addizione\n")
```

```
    a = float(input("Inserisci il Primo Numero -> "))
```

```
    b = float(input("Inserisci il Secondo Numero -> "))
```

```
    print("Il risultato dell'Addizione è: ", str(a + b))
```

```
elif action == "2":
```

```
    print("\nHai scelto: Sottrazione\n")
```

```
    a = float(input("Inserisci il Primo Numero -> "))
```

```
    b = float(input("Inserisci il Secondo Numero -> "))
```

```
    print("Il risultato dell'Sottrazione è: ", str(a - b))
```

```
elif action == "3":
```

```
    print("\nHai scelto: Moltiplicazione\n")
```

```
    a = float(input("Inserisci il Primo Numero -> "))
```

```
    b = float(input("Inserisci il Secondo Numero -> "))
```

```
    print("Il risultato dell'Moltiplicazione è: ", str(a * b))
```

```
elif action == "4":
```

```
    print("\nHai scelto: Divisione\n")
```

```
    a = float(input("Inserisci il Primo Numero -> "))
```

```
    b = float(input("Inserisci il Secondo Numero -> "))
```

```
    print("Il risultato dell'Divisione è: ", str(a / b))
```

```
elif action == "5":
```

```
    print("\nHai scelto: Calcolo Esponenziale\n")
```

```
    a = float(input("Inserisci la Base -> "))
```

```
    b = float(input("Inserisci l'Esponente -> "))
```

```
    print("Il risultato del Calcolo Esponenziale è: ", str(a ** b))
```

```
elif action == "6":
```

```
    print("\nHai scelto: Radice Quadrata\n")
```

```
    a = float(input("Inserisci il Numero -> "))
```

```
    print("Il risultato dell'operazione è: ", str(sqrt(a)))
```

```
elif action == "ESC":  
    print("\nL'Applicazione verrà chiusa!\n")  
    break  
  
new_action = input("\nDesideri continuare ad utilizzare l'Applicazione? S/N ")  
  
if new_action == "S" or new_action == "s":  
    print("Sto tornando al menù principale!\n")  
    continue  
  
else:  
    print("A presto!\n")  
    break
```

Per imparare Python al meglio dovete... sperimentare!

Vi invito a modificare il codice messo a disposizione cercando possibili modi per ottimizzarlo e aggiungere nuove funzionalità.

Volendo potete iniziare modificando la porzione finale dove chiediamo all'utente se intenda continuare a usare l'applicazione in modo da evitare l'utilizzo di continue ed else, in questo modo

```
new_action = input("\nDesideri continuare ad utilizzare l'Applicazione? S/N ")  
  
if new_action == "N" or new_action == "n":  
    print("A presto!\n")  
    break  
  
print("Sto tornando al menù principale!\n")
```

E se vi state chiedendo "come mai non ha usato una funzione?" Be, perché di questo ancora non abbiamo parlato.

11. Le Funzioni in Python

Introdurremo oggi un argomento più avanzato rispetto a quanto trattato finora, ma assolutamente fondamentale.

Parleremo di funzioni: vedendo cosa sono, a cosa servono e come si usano in Python!

Riutilizzo del codice

Uno dei concetti comuni nel mondo della programmazione è il concetto di **riutilizzo del codice**. Generalmente, più un programma diventa complesso e strutturato, più saranno le righe di codice che lo compongono. Affinché sia possibile tenere questo codice quanto più leggero possibile, uniforme e semplice da mantenere, è **molto importante evitare ripetizioni superflue**.

Finora abbiamo visto alcuni esempi di riutilizzo del codice: abbiamo imparato a usare i cicli di controllo, i moduli della Standard Library e alcune funzioni integrate come la funzione **print()**, usata più volte e in parti diverse dei nostri programmi.

In aggiunta alle funzioni integrate offerte da Python, **potete scrivere le vostre funzioni** per fargli fare ciò che desiderate: in questa lezione imparerete a fare proprio questo.

Ciò risulta molto importante perché ci permetterà come vedremo, di strutturare i nostri programmi in maniera efficiente ed elegante.

Le funzioni in Python

L'idea dietro al concetto di funzione è quella di assegnare una porzione di codice e delle variabili chiamate **parametri** ad un nome, e mediante questo nome che funge da collegamento richiamare tutto il codice associato.

Per certi versi il concetto è simile a quello di variabile che abbiamo visto nella [lezione numero 3](#), solo che in questo caso possiamo richiamare non solo dei dati ma anche delle istruzioni.

E così come possiamo creare dei semplici programmi salvandoli all'interno di file con estensione .py, per eseguirli tramite terminale, possiamo usare le funzioni per creare dei mini programmi ed eseguirli dall'interno del codice stesso.

Le funzioni sono quindi fantastiche per fare in modo di ridurre il numero di volte che un pezzo di codice deve essere scritto in un programma e per mantenere il nostro software ordinato.

La parola chiave def

Per definire una funzione utilizziamo la parola chiave **def** seguita dal nome che vogliamo dare a questa sezione del nostro programma, quindi a questa funzione. Come nel caso delle variabili, cercate di usare dei nomi che siano rappresentativi dello scopo della funzione.

Il nome viene seguito da una coppia di parentesi, e all'interno delle parentesi metteremo i parametri, ovvero delle variabili necessarie al funzionamento della funzione. Non tutte le funzioni avranno bisogno di parametri: in questo primo esempio lasceremo le parentesi vuote.

```
def say_my_name():  
    name = input("Come ti chiami? ")  
    print("Il tuo nome è: ", name)
```

Seguendo la logica di Python, il codice inserito all'interno della funzione deve essere indentato. All'interno della funzione potete aggiungere qualsiasi istruzione riteniate opportuna.

Per eseguire la funzione dobbiamo **richiamarla**, e per fare una chiamata alla funzione possiamo semplicemente utilizzare il suo nome seguito dalle parentesi

```
# definizione della funzione  
  
def say_my_name():  
    name = input("Come ti chiami? ")  
    print("Il tuo nome è: ", str(name))  
  
# chiamata della funzione  
say_my_name()
```

Facciamo ora un secondo esempio in cui utilizzeremo i parametri che abbiamo prima accennato. Definiamo una funzione addizione a cui passeremo due numeri e che ci restituirà la somma tra i due:

```
def addizione(a, b):  
    print("Questa è la funzione addizione.")  
    print("Fornisce la somma di due numeri passati come parametri.")  
    risultato = a + b  
    print("Il risultato dell'addizione è " + str(risultato))
```

Eseguiamo la funzione mediante la chiamata al suo nome e passiamoli due numeri come argomento tra parentesi.

```
addizione (15, 5)
```

L'argomento è il nome che viene dato al valore passato per essere depositato nella variabile parametro. Quindi i parametri in questo caso sono *a* e *b*, e gli argomenti dei parametri 15 e 5.

```
# output  
  
Questa è la funzione addizione.  
  
Fornisce la somma di due numeri passati come parametri.  
  
Il risultato dell'addizione è 20
```

Non ci sono particolari limiti per quanto riguarda i parametri che è possibile passare, ma badate di sceglierli sempre con cura e di dargli anche in questo caso dei nomi specifici.

Le docstring di Python

Una **docstring** è una stringa di testo che viene inserita all'inizio di una funzione, di una classe o di un modulo e che serve a documentare il codice. A differenza dei [commenti](#), le docstring sono considerate parte del codice e vengono utilizzate per generare la documentazione automaticamente: questo significa che possono essere lette da strumenti di documentazione come [Sphinx](#) e possono essere utilizzate per generare documentazione HTML e PDF. Le docstring si scrivono inserendo il testo all'interno di stringhe multiriga definite con tripli apici `""" """`:

```
def somma(a, b):  
    """  
    Questa funzione prende due argomenti, a e b, e restituisce la loro somma.  
    """  
    return a + b
```

In questo caso, la docstring descrive lo scopo della funzione e il valore che viene restituito. Possono essere utilizzate anche per documentare i parametri di una funzione, ad esempio:

```
def saluta(nome):  
    """  
    Questa funzione prende un argomento, nome, e restituisce una stringa di saluto.  
  
    :param nome: il nome della persona da salutare  
    :return: una stringa di saluto  
    """  
    return "Ciao, " + nome + "!"
```

In questo caso, la docstring descrive il parametro *nome* e il valore di ritorno della funzione.

Quando si utilizza una funzione che ha una docstring in **Visual Studio Code**, è possibile visualizzarla attraverso la funzionalità **IntelliSense** dell'editor: questo significa che quando si digita il nome della funzione, verrà mostrato un suggerimento che include la docstring della funzione, compresi i parametri e il valore che viene restituito. Vediamo un esempio:

```
def calcola_media(valori):  
    """  
    Calcola la media dei valori nella lista fornita.  
  
    Args:  
        valori (list): una lista di numeri
```

Returns:

float: la media dei numeri nella lista

"""

somma = 0

for valore in valori:

 somma += valore

media = somma / len(valori)

return media

Se scriviamo il nome della funzione e ci andiamo sopra con il mouse, vedremo la docstring:

```
(function) def calcola_media(valori: Any) -> (Any | float)
Calcola la media dei valori nella lista fornita.
Args:
  valori (list): una lista di numeri
Returns:
  float: la media dei numeri nella lista
calcola_media
```

Parametri opzionali e valori di default

Talvolta ci troveremo in una situazione in cui è necessario avere dei parametri opzionali per le nostre funzioni, ovvero fare in modo che queste funzionino con o senza l'aggiunta di valori in questi specifici parametri. Per rendere questo possibile, possiamo associare a questi parametri un valore di default.

Immaginate che stiate andando a comprare un laptop nuovo. Due caratteristiche da tenere sott'occhio sono sicuramente RAM e modello della CPU, che quindi sceglieremo. Come vi sarà capitato, vi viene chiesto se volete acquistare anche un pacchetto antivirus, e voi potete scegliere se lo volete oppure meno.

Proviamo a portare questo esempio in una funzione:

```
def laptop_nuovo(ram, cpu, antivirus=False):
    print("Il nuovo laptop avrà le seguenti caratteristiche:")
    print("ram: " + ram)
    print("cpu: " + cpu)
    if antivirus == True: # comparazione esplicita per fini didattici
        print("Hai comprato anche un antivirus!")
```

Quindi qualora non ci interessi acquistare l'antivirus possiamo semplicemente ignorare questo parametro a cui è stato dato il valore di default *False*:

```
>>> laptop_nuovo("16gb", "i7")
```

Il nuovo laptop avrà le seguenti caratteristiche:

ram: 16gb

cpu: i7

Diversamente se vogliamo l'antivirus ci basta passare *True* come valore per il parametro antivirus:

```
>>> laptop_nuovo("16gb", "i7", antivirus=True)
```

Il nuovo laptop avrà le seguenti caratteristiche:

ram: 16gb

cpu: i7

Hai comprato anche un antivirus!

L'istruzione **return** in Python

Prestate ora attenzione a un dettaglio.

Nella nostra funzione *addizione*, per fare in modo di mandare in output il risultato della somma abbiamo utilizzato una seconda funzione, ovvero la funzione **print()**, a cui abbiamo passato il risultato dell'operazione.

Proviamo a togliere tutti i **print()** e lasciare solo la variabile *risultato* e richiamiamo quindi la funzione:

```
def addizione(a, b):
```

```
    risultato = a + b
```

```
addizione(3, 3)
```

```
# output
```

Come immaginabile, la funzione non ci mostra nulla! Per fare in modo che la nostra funzione **restituisca** il risultato in maniera autonoma dobbiamo utilizzare il comando **return**.

Potremo quindi usare questo dato passandolo ad esempio a una variabile o usando la funzione **print()**:

```
def addizione(a, b):
```

```
    risultato = a + b
```

```
    return risultato
```

```
risultato = addizione(3, 6)
```

```
print(risultato)
```

```
# output
```

```
9
```

...ed ecco che ora otteniamo nuovamente il contenuto di risultato!

Il tipo nullo di Python: `NoneType`

Se non definiamo un valore da restituire tramite `return`, la funzione restituirà il tipo di dato nullo o *null* di Python, il **`None`** **Type**. Modifichiamo il codice scritto in precedenza e facciamo una prova

```
def addizione(a, b):
```

```
    risultato = a + b
```

```
risultato = addizione(3, 6)
```

```
print(risultato)
```

```
print(type(risultato))
```

```
# output
```

```
None
```

```
<class 'NoneType'>
```


12. Variabili Globali e Variabili Locali

In questa lezione approfondiremo il discorso su variabili e funzioni parlando di **variabili globali** e **variabili locali**.

Finora abbiamo detto, semplificando il tutto a fini didattici, che possiamo immaginare le variabili come delle scatole in cui depositiamo dei valori. Possiamo quindi richiamare e modificare questo valore utilizzando il nome della variabile.

```
x = 3
x += 1
x
4
```

Abbiamo inoltre parlato di funzioni, e abbiamo visto che anche qui possiamo e siamo incoraggiati ad utilizzare le variabili: si tratta di uno degli elementi **indispensabili** di Python e della programmazione in generale. Ma fino a dove ci possiamo spingere nell'utilizzo delle variabili? Fino a che punto è davvero possibile utilizzarle e modificarle nei nostri programmi?

Per rispondere a queste domande andremo ora ad approfondire il tema parlando di **Variabili Globali** e **Variabili Locali**. Partiamo da un esempio:

```
x = 15

def funzione_esempio():
    return x

print(funzione_esempio())
15
```

La nostra funzione esempio ci restituisce il valore assegnato alla variabile x.

Proviamo però ora a modificare leggermente la nostra funzione; diciamo che vogliamo aggiungere 2 al 15 presente in x.

```
x = 15

def funzione_esempio():
    x += 2
    return x

print(funzione_esempio())

# output in Python 3.11
UnboundLocalError: cannot access local variable 'x' where it is not associated with a value
```

in versioni precedenti di Python, il messaggio d'errore mostrato era

UnboundLocalError: local variable 'x' referenced before assignment

Come mai abbiamo ottenuto questo errore? Python ci informa del fatto che ci stiamo riferendo a una variabile prima di averla dichiarata e assegnata, eppure nell'esempio precedente tutto ha funzionato a meraviglia e la variabile `x` è proprio lì!

Local Scope e Global Scope

La chiave di lettura di questo comportamento apparentemente bizzarro è la seguente, prestate attenzione: in Python il codice e quindi anche le variabili, possono essere "salvati" in due ambienti diversi chiamati **local scope** e **global scope**, traducibili sostanzialmente come **ambito di visibilità locale** e **globale**

Potete pensare a questi due ambienti come a dei contenitori distinti in cui vengono definite e assegnate le variabili, un contenitore globale e un contenitore locale. Quando uno di questi contenitori viene distrutto, quindi quando uno di questi ambiti viene distrutto, lo stesso accade per i valori delle variabili in esso salvate che vengono quindi dimenticati.

Variabili Locali e Variabili Globali

Un **ambito locale** viene creato ogni volta che una funzione viene **chiamata**, e distrutto dopo che la funzione restituisce un valore con **return**.

È come se ogni volta che una funzione viene processata, Python le fornisse un contenitore e le dicesse: bene, metti le tue variabili e il tuo codice qua dentro. Possono quindi esistere tanti Local Scope quante funzioni abbiamo in esecuzione. Le variabili dichiarate all'interno di qualsiasi funzione, quindi dell'Ambito Locale della funzione, sono chiamate **variabili locali**.

Dall'altro lato invece, esiste e può esistere un unico Ambiente Globale, che viene creato automaticamente da Python all'avvio del programma e distrutto alla chiusura del programma. È l'ambito principale e tutte le variabili che vengono dichiarate qui, quindi all'esterno di una funzione, sono chiamate proprio **variabili globali**, e restano pertanto in vita dall'avvio del programma "principale" fino alla sua chiusura.

È possibile accedere alle variabili globali da qualsiasi parte del programma, mentre è possibile accedere alle variabili locali solamente dall'ambito locale della funzione in cui sono contenute.

Nel nostro esempio, `x` è proprio una variabile globale, e otteniamo un errore perché seppur è vero che da un local scope si può accedere alle variabili del global scope, il loro utilizzo all'interno della funzione è limitato.

L'istruzione **global**

Per poter modificare il valore di una variabile globale dall'interno di una funzione, come abbiamo provato a fare con la nostra `x`, dobbiamo prima dichiarare alla funzione le nostre intenzioni mediante l'istruzione **global**. Modifichiamo l'esempio precedente:

```
x = 15

def funzione_esempio():
    global x
    x += 2
    return x

print(funzione_esempio())

17
```

Ed ecco che ora ci è possibile modificare il valore della variabile `x`, perché Python sa che ci stiamo riferendo alla `x` presente nel global scope!

Un altro modo per poter utilizzare il valore di una variabile globale in maniera più complicata all'interno di una funzione è creando una variabile locale a cui assegnamo il valore della variabile globale. Potremo quindi poi modificare la nostra nuova variabile locale, aggirando così la limitazione.

```
x = 15

def funzione_esempio():
    y = x
    y += 2
    return y

print(funzione_esempio())

17
```

Come accennato inoltre, al contrario delle globali, è possibile accedere alle variabili locali solamente dall'ambito locale della funzione in cui sono contenute, quindi non è possibile accedervi dal global scope o dal local scope di un'altra funzione e otterremmo un errore qualora provassimo a forzare questa regola. Ad esempio:

```
def mia_funzione():
    val = 24
    print(val)
```

```
new_val = val + 6
```

```
# output
```

```
NameError: name 'val' is not defined
```

Potete utilizzare le variabili di una funzione solo se queste ci vengono passate dal return della funzione. Ad esempio, facciamo una piccola modifica:

```
def mia_funzione():
```

```
    val = 24
```

```
    return val
```

```
new_val = mia_funzione() + 6
```

```
print(new_val)
```

```
# output
```

```
30
```

Ed ecco che ora possiamo effettuare una somma con *spam* dall'ambito principale del programma.

Local e Global Scope: quanto sono importanti?

È bene imparare la distinzione tra local e global scope perché si tratta di concetti estremamente importanti. L'esistenza degli ambiti è utilissima ad esempio per ridurre il numero di bug e aumentare la robustezza dei programmi.

Pensate a che caos se in un programma complesso tutte le variabili fossero globali, e per colpa di un errore in una delle tante funzioni del programma i valori delle variabili fossero perennemente sballati, causando un crash sistematico. Inoltre restando in memoria fino alla fine del programma, in programmi complessi le variabili globali sono sinonimo di pesantezza e spreco di risorse.

L'utilizzo di Variabili Locali è quindi estremamente incoraggiato, e quello di Variabili Globali decisamente SCONSIGLIATO

13. Le Liste

In questa lezione parleremo di un nuovo tipo di dato di Python: le **liste**.

Le Liste in Python

Il tipo di dato Lista di Python, come il nome suggerisce, ci permette di raggruppare più elementi tra di loro.

Le liste in Python sono molto potenti, possono infatti contenere al loro interno diversi tipi di dato anche tutti assieme e i vari elementi vengono ordinati in base ad un **indice** proprio della lista, in modo da semplificarne l'accesso.

Creiamo ora una variabile *my_list* ed assegniamole una lista che contenga al suo interno dati di tipo *integer*, *float* e *string*. Per dichiarare una lista usiamo una coppia di parentesi quadre, e i vari elementi contenuti sono separati da una virgola

```
>>> my_list = [9.81, "pasta", 22, 44, 3.14]
>>> type(my_list)
```

output

```
<class 'list'>
```

Per ottenere in output il valore associato alla variabile *my_list*, in questo caso la nostra lista, possiamo usare anche stavolta il nome della variabile come visto finora con tutte le altre tipologie di dato:

```
>>> my_list
```

output

```
[9.81, 'pasta', 22, 44, 3.14]
```

Le liste sono così versatili che volendo possono contenere anche altre liste:

```
>>> new_list = ["asd", "poker", "luna", my_list]
```

```
>>> new_list
```

output

```
['asd', 'poker', 'luna', [9.81, 'pasta', 22, 44, 3.14]]
```

Gli indici del tipo di dato list in Python

Abbiamo detto che gli elementi sono ordinati in base ad un indice proprio di ciascuna Lista, e ciò significa che a ciascun elemento è associato un numero che ne rappresenta la posizione nell'elenco: comprendere ciò è fondamentale perché se vogliamo accedere ad un elemento specifico avremo bisogno dell'indice corrispondente nella lista.

NOTA FONDAMENTALE: le liste iniziano ad indice 0.

RIPETO: il primo indice delle liste è 0.

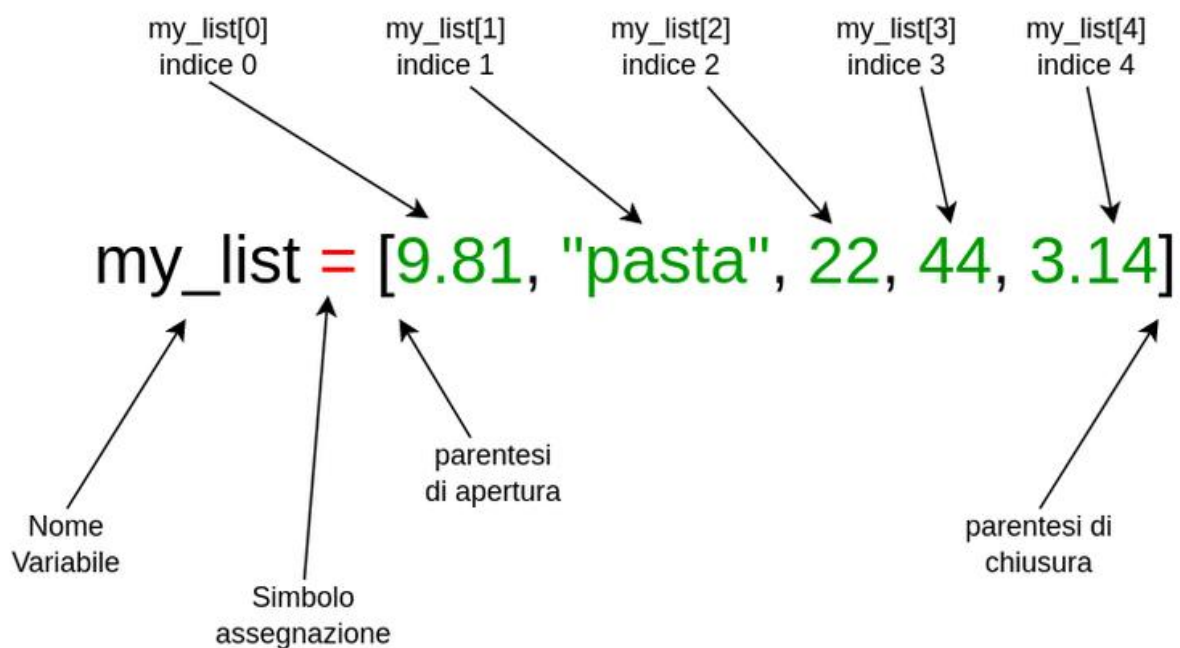
Per richiamare "pasta" dovremo quindi usare l'indice 1:

```
my_list[1]  
'pasta'
```

Mentre per ottenere 9.81 dovremo usare l'indice 0:

```
my_list[0]  
9.81
```

Per facilitare la comprensione da un punto di vista visivo ho preparato questo schema:



Talvolta può dimostrarsi utile accedere agli elementi della lista partendo dalla fine: in questo caso possiamo usare **indici negativi**, partendo da -1:

```
my_list[-1]  
3.14
```

Facciamo ora un esempio partendo da una lista di numeri primi.

Per ottenere un insieme di elementi contenuti in una lista, possiamo usare lo **slicing**, che si traduce letteralmente come **affettare**.

Questo ci permette di ottenere una "sottolista" di elementi a partire da due indici, e volendo possiamo assegnare questa nuova lista a una nuova variabile:

```
primi = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

```
primi[4:10]
```

```
# output
```

```
[11, 13, 17, 19, 23, 29]
```

```
fetta = primi[4:10]
```

```
fetta
```

```
# output
```

```
[11, 13, 17, 19, 23, 29]
```

Possiamo impostare questo indice variabile anche in modo da farlo iniziare, o terminare, con l'inizio oppure la fine della lista, utilizzando i due punti `:`.

Per ottenere tutti i numeri dal 5o all'ultimo nella nostra lista possiamo fare:

```
>>> primi[4:]
```

```
[11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

Ho utilizzato il quarto elemento perché partendo dall'indice 0, il quinto elemento nella lista *primi* sarà appunto quello con indice 4.

Parallelamente possiamo inoltre ottenere tutti gli elementi a partire dal primo fino ad una soglia da noi scelta in questo modo:

```
>>> primi[:5]
```

```
[2, 3, 5, 7, 11]
```

Liste e cicli for

Le liste sono davvero ovunque e vi capiterà di incontreremo spesso, forse anche più di altri tipi di dato! È quindi fondamentale tenere a mente che per navigare il contenuto di una lista possiamo usare ad esempio i cicli **for**:

```
primi = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

```
match = 11
```

```
for el in primi:
```

```
    if el == match:
```

```
        output = str(el) + " == " + str(match)
```

```
        print("Match! " + output)
```

```
    else:
```

```
        print(element)
```

```
# output
```

```
2
```

```
3
```

```
5
```

```
7
```

```
Match! 11 == 11
```

```
13
```

```
17
```

```
19
```

```
23
```

```
29
```

```
31
```

```
37
```

```
41
```

```
43
```

```
47
```

Torniamo ora all'esempio della variabile *new_list*, che al suo interno contiene anche la lista *my_list*. Volendo ottenere uno specifico elemento dalla lista *my_list* possiamo usare un doppio indice:


```
>>> my_list = [9.81, "pasta", 22, 44, 3.14]
>>> new_list = ["asd", "poker", "luna", my_list]
>>> new_list[-1][1]

# output
'pasta'
```

I Metodi delle Liste

Proviamo ora ad aggiungere un nuovo elemento alle nostre liste. Supponiamo di avere una lista della spesa e di voler aggiungere un nuovo elemento da acquistare.

```
>>> spesa = ["riso", "pollo", "verdura"]
>>> spesa += "sapone"
>>> spesa

# output
['riso', 'pollo', 'verdura', 's', 'a', 'p', 'o', 'n', 'e']
```

Python non ci dà errore, ma otteniamo un output decisamente controintuitivo! Questo perché *in un certo senso* le stringhe di Python sono delle liste di caratteri. Al di là di questo, otteniamo un errore provando ad aggiungere altre tipologie di dati.

```
>>> spesa = ["riso", "pollo", "verdura"]
>>> spesa += 3.14

# output
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'float' object is not iterable
```

Per lavorare con le liste di Python usiamo alcune specifiche istruzioni e delle funzioni "speciali" chiamate **metodi**. Senza scendere troppo nei dettagli avanzati, la particolarità dei metodi è che vengono richiamati **su** dati di un certo tipo.

Tipi di dato diversi avranno a disposizione metodi diversi.



Aggiungere elementi con `append()` ed `extend()`

Volendo aggiungere metodi singoli elementi ad una lista usiamo il metodo **`append()`**:

```
>>> spesa = ["riso", "pollo", "verdura"]
>>> spesa.append("sapone")
>>> spesa
```

output

```
['riso', 'pollo', 'verdura', 'sapone']
```

Mentre per aggiungere elementi ad una lista da una seconda lista usiamo il metodo **`extend()`**:

```
>>> spesa_extra = ["camicia", "copri volante"]
>>> spesa.extend(spesa_extra)
>>> spesa
```

output

```
['riso', 'pollo', 'verdura', 'sapone', 'camicia', 'copri volante']
```

Ordinare elementi tramite il metodo `sort()`

Il termine **`sort()`** è traducibile in Italiano come **ordinare**, o **classificare**.

Questo metodo ci consente di ordinare i valori all'interno di una lista mediante ordine alfabetico o numerico:

```
>>> alfabeto = ["z", "u", "a", "c", "v"]
```

```
>>> alfabeto.sort()
```

```
>>> alfabeto
```

```
# output
```

```
['a', 'c', 'u', 'v', 'z']
```

```
#####
```

```
>>> numeri = [32, 1, 15, 2, 22, 23, 56, 88]
```

```
>>> numeri.sort()
```

```
>>> numeri
```

```
# output
```

```
[1, 2, 15, 22, 23, 32, 56, 88]
```

```
#####
```

```
>>> numeri.sort(reverse=True)
```

```
>>> numeri
```

```
# output
```

```
[88, 56, 32, 23, 22, 15, 2, 1]
```

Il metodo `index()`: come ottenere l'indice di un elemento

Gli indici delle liste sono fondamentali perché identificano la posizione degli elementi al loro interno.

Per ottenere l'indice di uno specifico elemento possiamo usare il metodo **`index()`**.

```
>>> numeri = [32, 1, 15, 2, 22, 23, 56, 88]
```

```
>>> numeri.index(22)
```

```
# output
```

```
4
```

```
#####
```

```
>>> numeri.index(88)
```

```
# output
```

```
7
```

Sostituire ed eliminare elementi tramite indice

Possiamo sostituire un elemento passando un nuovo valore all'indice dell'elemento che intendiamo sostituire, oppure cancellarlo:

```
>>> spesa = ["riso", "pollo", "verdura"]
```

```
>>> spesa[1] = "tacchino"
```

```
>>> spesa
```

```
# output
```

```
['riso', 'tacchino', 'verdura', 'sapone']
```

```
#####
```

```
>>> del spesa[0]
```

```
>>> spesa
```

```
# output
```

```
['tacchino', 'verdura']
```

Rimuovere elementi con `pop()` e `remove()`

Infine, possiamo rimuovere elementi dalle liste usando anche i metodi **`pop`** e **`remove`**.

Con **`pop()`** rimuoviamo l'ultimo elemento della lista, mentre con **`remove()`** possiamo rimuovere elementi a partire dal loro valore:

```
>>> new_list = [True, None, "poker", 4.20, 1945]
```

```
>>> new_list
```

```
# output
```

```
[True, None, 'poker', 4.2, 1945]
```

```
#####
```

```
>>> new_list.pop()
```

```
1945
```

```
>>> new_list
```

```
# output
```

```
[True, None, 'poker', 4.2]
```

```
#####
```

```
>>> new_list.remove(None)
```

```
>>> new_list
```

```
[True, 'poker', 4.2]
```

L'elenco dei metodi delle liste non finisce qui, ma per ora direi che può bastare!

Parleremo ancora di liste più avanti nel corso.

14. Metodi delle Stringhe

In questa lezione approfondiremo la nostra conoscenza delle stringhe in Python introducendo alcuni metodi molto utili propri di questo tipo di dato e facendo alcune analogie col tipo di dato **lista**, scoprendo più di qualche punto in comune tra i due!

F-String o Formatted String Literal

Finora quando abbiamo dovuto concatenare più stringhe tra di loro, abbiamo utilizzato l'operatore **+**:

```
>>> nome = "Jack"
>>> "Ciao " + nome
'Ciao Jack'
```

Per poter unire dei valori numerici inoltre, abbiamo sempre dovuto utilizzare la funzione di supporto **str()**, che ci permette di ottenere la rappresentazione in stringa:

```
numero = 218
>>> "Ciao " + nome + " il tuo posto è il n" + str(numero)
'Ciao Jack il tuo posto è il n218'
```

Questo può portare a confusione o alla creazione di righe di codice particolarmente lunghe e difficili da leggere. In alternativa possiamo usare un sistema di formattazione avanzata chiamato **Formatted String Literal**, o **f-string**.

Con le **f-string** valori, variabili o espressioni possono essere passati all'interno di parentesi graffe.

```
>>> f"Ciao {nome}, il tuo posto è il n{numero}! Benvenuto!"
'Ciao Jack, il tuo posto è il n18! Benvenuto!'
```

Facciamo un altro esempio:

```
>>> z = 5
>>> f"Il quadrato di {z} è {z * z}"
'Il quadrato di 5 è 25'
```

Vi consiglio di tenere a mente questa tecnica di formattazione delle stringhe perché penso vi troverete ad usarla spessissimo.

Metodi per Stringhe

Come abbiamo detto nelle lezioni precedenti del corso, i metodi sono delle "funzioni speciali" proprie nel nostro caso dei tipi di dato. Sono proprietà di oggetti di una determinata tipologia.



Anche il tipo di dato stringa dispone di metodi propri molto comodi per permetterci di eseguire svariate comode azioni. I metodi delle stringhe sono davvero tanti e per un elenco completo vi consiglio di dare uno sguardo a [questa pagina](#).

In questa lezione analizzeremo alcuni dei metodi più comunemente utilizzati.

I metodi `startswith()` e `endswith()`

Con **`startswith()`** e **`endswith()`** possiamo controllare se una data stringa inizi o finisca con un determinato carattere o insieme di caratteri. Questi metodi restituiscono valori **booleani**, quindi ricordiamolo, *True* o *False*.

```
>>> messaggio = "Fate il vostro gioco"

>>> messaggio.startswith("Fate")
True

>>> messaggio.startswith("F")
True

>>> messaggio.starstwith("x")
False

>>> messaggio.endswith("gioco")
True

>>> messaggio.endswith("gioc")
False
```

I metodi `isalnum()`, `isalpha()`, `isdecimal()` e `isspace()`

Capiterà spesso che dobbiate verificare la tipologia di caratteri contenuti in una stringa, e i metodi **`isalnum()`**, **`isalpha()`**, **`isdecimal()`** e **`isspace()`** ci permettono di verificare proprio ciò per la gran parte dei casi più comuni, restituendoci *True* se la condizione è veritiera:

- **metodo `isalnum()`**: solo caratteri alfanumerici
- **metodo `isalpha()`**: solo caratteri alfabetici
- **metodo `isdecimal()`**: solo numeri
- **metodo `isspace()`**: tutti i caratteri sono spazi bianchi

```
spam = "asd123"
eggs = "999"
bacon = " "
monty = "poker "

>>> spam.isalnum()
True
>>> spam.isalpha()
False

>>> eggs.isdecimal()
True
>>> eggs.isalnum()
True

>>> monty.isspace()
False
>>> bacon.isspace()
True
```


I metodi `upper()` e `lower()`

I metodi **`upper()`** e **`lower()`** ci permettono di ottenere la versione maiuscola o minuscola di una stringa. Tenete però a mente che questi metodi non modificano il valore contenuto in una determinata variabile: per fare ciò sarà necessario assegnare un nuovo valore alla stessa variabile.

```
>>> name = "Alice"

>>> name.lower()
'alice'

>>> name.upper()
'ALICE'

# il valore di name non è cambiato
>>> name
'Alice'

>>> name = name.upper()

# il valore di name è ora stato cambiato
>>> name
'ALICE'
```

I metodi `split()` e `join()`

Il metodo **`join()`** è utile quando ci troviamo con una lista di stringhe e vogliamo unirle tra di loro per formare una nuova stringa risultante, come nel caso di una lista di parole. Questo metodo viene chiamato su una stringa che usiamo "a mo' di collante" tra le varie stringhe che vogliamo unire. Nel nostro caso questa stringa "collante" potrebbe essere una virgola seguita da uno spazio, oppure un carattere newline.

```
>>> to_do = ["portare il cane a passeggio", "finire di studiare", "fare la spesa", "lavare i panni"]

>>> ", ".join(to_do)
'portare il cane a passeggio, finire di studiare, fare la spesa, lavare i panni'

>>> da_fare = "oggi devo: " + ", ".join(to_do)
```

```
>>> da_fare

'oggi devo: portare il cane a passeggio, finire di studiare, fare la spesa, lavare i panni'

#####

>>> da_fare = "\n".join(to_do)

>>> print(da_fare)

portare il cane a passeggio
finire di studiare
fare la spesa
lavare i panni
```

Col metodo **split()** possiamo al contrario dividere una stringa in una lista di stringhe, e dovremo passare come parametro il carattere che intendiamo usare come separatore.

```
>>> citazione = "Nel mezzo del cammin di nostra vita..."

>>> citazione.split(" ")

['Nel', 'mezzo', 'del', 'cammin', 'di', 'nostra', 'vita...']
```

Il metodo **format()**

Per formattare le stringhe possiamo utilizzare anche il metodo **format()**:

```
stringa.format(valore1, valore2, ...)
```

Dove *stringa* è la stringa da formattare e *valore1*, *valore2*, ecc. sono i valori da inserire nella stringa. Il metodo **format()** sostituisce ogni posizione **{}** nella stringa con il valore corrispondente passato come parametro. Ad esempio:

```
>>> colore1 = "blu"

>>> colore2 = "verde"

>>> s = "I miei colori preferiti sono {} e {}".format(colore1, colore2)

>>> print(s)

# output

I miei colori preferiti sono blu e verde.
```

Questo metodo è utile quando si devono creare stringhe complesse o dinamiche, dove i valori possono variare a seconda della situazione ed era molto utilizzato prima della versione 3.6 di Python: nel [PEP 498](#) sono state proposte e poi aggiunte nel linguaggio le **f-string**, che risultano molto più compatte e rendono il codice più facilmente leggibile.

Similarità tra Liste e Stringhe

I tipi di dato di Python list e string sono per certi aspetti simili tra di loro, e per questo motivo ci sono alcune tipologie di azioni che possiamo fare su entrambi. Potete in questi esempi pensare quindi alle stringhe come a delle liste di caratteri.

La funzione len()

La funzione integrata di Python **len()** ci permette di ottenere la lunghezza di una lista o una stringa, e quindi rispettivamente il numero di elementi o di caratteri presenti in queste:

```
>>> my_str = "spam, eggs, bacon, spam"
>>> my_list = ["spam", "spam", "spam"]

>>> len(my_str)
23

>>> len(my_list)
3
```

Operatori in e not in

Possiamo usare gli operatori **in** e **not in** per verificare se un elemento o un carattere sia presente in una lista o stringa:

```
>>> my_list = ["spam", "spam", "spam"]
>>> new_str = "happiness"

>>> "bacon" in my_list
False

>>> "spam" in my_list
True

>>> "eggs" not in my_list
True
```

```
>>> "k" not in new_str
```

```
True
```

```
>>> "h" in new_str
```

```
True
```

String Slicing

In maniera analoga a quanto visto con le liste, possiamo accedere ai caratteri di una stringa tramite indice, in modo da ottenere una porzione della stessa. In questo esempio abbiamo una variabile chiamata *alfa* che contiene un insieme di caratteri alfabetici + tre punti di sospensione.

Volendo possiamo isolare la porzione puramente alfabetica o i tre punti usando lo **slicing** e usare questi valori per creare delle nuove variabili:

```
>>> alfa = "abcdefghijklm..."
```

```
>>> dots = alfa[-3:]
```

```
>>> dots
```

```
'...'
```

```
>>> alfa = alfa[:-3]
```

```
>>> alfa
```

```
'abcdefghijklm'
```

Stringhe e cicli for

Come per le liste, possiamo usare i cicli **for** assieme alle nostre stringhe:

```
random_alnum = "dj1oi3u4nuqnoru01u3m3mdasd"
```

```
counter = 0
```

```
match = "d"
```

```
for char in random_alnum:
```

```
    if char == match:
```

```
        counter += 1
```

```
output = f"Ho trovato {counter} caratteri '{match}' "
```

```
>>> output
```

```
"Ho trovato 3 caratteri 'd' "
```

15. Tuple e Set

In questa lezione parleremo di due nuovi tipi di dato: **tuple** e **set**.

Le tuple in Python

Come il tipo di dato lista visto nelle lezioni precedenti, anche il tipo **tuple** di Python rappresenta un insieme di elementi, definiti stavolta utilizzando una coppia di **parentesi tonde**, oppure senza alcuna parentesi ma separati da virgola.

```
>>> tupla = (2, 4, 9, 15, 23)
```

```
>>> type(tupla)
```

```
# output
```

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

```
#####
```

```
>>> tupla
```

```
# output
```

```
(2, 4, 9, 15, 23)
```

```
#####
```

```
>>> tupla_due = 7, 8, 9
```

```
>>> tupla_due
```

```
# output
```

```
(7, 8, 9)
```

Come per le liste, possiamo accedere agli elementi delle *tuple* utilizzando l'indice.

```
>>> tupla = (2, 4, 9, 15, 23)
```

```
>>> tupla[0]
```

output

2

Le tuple sono immutabili

La grande differenza tra i tipi di dato tuple e list è che le tuple sono immutabili. Per questo motivo il dato tuple **non dispone di un metodo append()** e non possiamo usare l'istruzione del per rimuovere elementi. Questo risulta molto comodo in certe situazioni: tenetelo a mente!

```
>>> tupla.append(999)
```

output

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

AttributeError: 'tuple' object has no attribute 'append'

#####

```
>>> del tupla[0]
```

output

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: 'tuple' object doesn't support item deletion

I set di Python

I **set** sono un altro tipo di dato di Python che permette di creare collezioni di elementi. I set vengono definiti con una coppia di **parentesi graffe**, e i valori sono separati da virgola

```
>>> my_set = {"asd", "qwerty", "cmd"}
```

```
>>> type(my_set)
```

output

```
<class 'set'>
```

La caratteristica fondamentale dei set è che non possono contenere duplicati al loro interno.

```
>>> new_set = {"asd", "asd", "qwerty", "cmd"}
>>> new_set

# output
{'qwerty', 'cmd', 'asd'}
```

Il metodo add() dei set

A differenza delle tuple possiamo però aggiungere nuovi elementi col metodo add.

Provare ad aggiungere elementi già presenti non funzionerà!

```
>>> new_set = {"asd", "asd", "qwerty", "cmd"}
>>> new_set.add("cmd")
>>> new_set

# output
{'qwerty', 'cmd', 'asd'}

#####

>>> new_set.add("nova")
>>> new_set

# output
{'qwerty', 'cmd', 'asd', 'nova'}
```

Approfondimento: Gli iterabili in Python

Finora abbiamo visto diversi tipi di dato in Python, come i numeri interi, i valori booleani, le liste e le stringhe.

C'è però una caratteristica che accomuna alcuni di questi tipi di dato: le **stringhe**, le **liste**, le **tuple**, i **set** e i **dizionari** (che vedremo in una lezione successiva), sono tutti oggetti che possono essere trattati come sequenze che contengono elementi accessibili per essere manipolati. Questo tipo di oggetti in Python sono detti **iterabili**. Gli altri tipi di dato, come ad esempio i booleani e i numeri int o float non sono iterabili perché rappresentano un unico valore o una singola entità e non contengono elementi.

Funzioni per iterabili

Python fornisce diverse funzioni per manipolare gli elementi degli **iterabili**, vediamo alcune delle più utilizzate:

La funzione `max()`

La funzione **`max()`** viene utilizzata per trovare il valore massimo di un oggetto iterabile.

```
max(iterable, key=None, default=None)
```

Dove *iterable* è la sequenza di cui si vuole trovare il valore massimo. Ad esempio possiamo usare **`max()`** per ottenere il numero maggiore in un set di numeri:

```
numeri = {3, 8, 1, 6, 9, 4}
valore_massimo = max(numeri)
print(valore_massimo)
```

output

9

L'argomento *default* specifica il valore da restituire se l'insieme di input è vuoto, mentre l'argomento *key* specifica un criterio di ordinamento personalizzato, ad esempio se gli passiamo **`len`** possiamo ottenere la parola più lunga in una tupla di parole:

```
parole = ('casa', 'auto', 'giardino', 'mestolo')
parola_più_lunga = max(parole, key=len)
print(parola_più_lunga)
```

output

giardino

La funzione `sum()`

La funzione **`sum()`** restituisce la somma degli elementi di un oggetto iterabile:

```
sum(iterable, start=0)
```

Dove *iterable* è l'oggetto iterabile che si desidera sommare e *start* è il valore di partenza opzionale della somma che per impostazione predefinita è uguale a zero, quindi se non viene specificato viene sommato il valore di tutti gli elementi. Ad esempio, sommiamo con **`sum()`** tutti gli elementi da una lista di numeri:

```
numeri = [33, 78, 5, 42, 23]
totale = sum(numeri)
```



```
print(totale)
```

```
# output
```

```
181
```

Vediamo cosa succede se impostiamo a 100 il valore di partenza:

```
totale = sum(neri, 100)
```

```
print(totale)
```

```
# output
```

```
281
```

La funzione `enumerate()`

La funzione **`enumerate()`** consente di iterare su una sequenza (liste, tuple, stringhe, ecc.) e restituire una tupla contenente il valore dell'elemento corrente e il suo indice:

```
enumerate(sequence, start=0)
```

Dove *sequence* è la sequenza da iterare e *start* è l'indice di partenza per la numerazione (il valore predefinito è 0). Ad esempio, se vogliamo stampare il nome di ogni frutto di una lista insieme al suo indice, possiamo utilizzare la funzione **`enumerate()`** in questo modo:

```
frutta = ["mela", "banana", "kiwi", "arancia"]
```

```
for indice, frutto in enumerate(frutta):
```

```
    print(indice, frutto)
```

```
# output
```

```
0 mela
```

```
1 banana
```

```
2 kiwi
```

```
3 arancia
```

In questo caso **`enumerate()`** restituisce una sequenza di tuple contenenti l'indice e il valore corrente della lista frutta. Queste tuple vengono assegnate alle variabili *indice* e *frutto* e vengono utilizzate all'interno del **ciclo for** per stampare il nome del frutto insieme al suo indice.

Se non c'è bisogno di utilizzare l'indice si può utilizzare l'underscore `_` al posto della variabile *indice*:

```
frutta = ["mela", "banana", "kiwi", "arancia"]
```

```
for _, frutto in enumerate(frutta):
```

```
    print(frutto)
```

```
# output
```

```
mela
```

```
banana
```

```
kiwi
```

```
arancia
```

La funzione `sorted()`

La funzione **`sorted()`** viene utilizzata per ordinare un iterabile. A differenza del metodo per liste `sort()`, che ha lo stesso scopo, può essere usato anche per le tuple e i set perché non modifica l'ordine degli elementi all'interno della sequenza originale ma la lascia invariata restituendo una nuova lista.

```
sorted(iterable, key=None, reverse=False)
```

Dove *iterable* è la sequenza di cui si vuole ordinare gli elementi. Come abbiamo visto per **`max()`**, anche in questo caso *key* consente di specificare un criterio di ordinamento personalizzato, mentre l'argomento *reverse*, se impostato su *True*, fa in modo che gli elementi siano in ordine decrescente.

Vediamo un esempio di utilizzo di **`sorted()`** a partire da una tupla:

```
numeri = (3, 8, 1, 6, 9, 4)
```

```
numeri_ordinati = sorted(numeri)
```

```
print(numeri_ordinati)
```

```
# output
```

```
[1, 3, 4, 6, 8, 9]
```

La funzione `filter()`

La funzione **`filter()`** prende in input una **funzione** e un **iterabile** e restituisce un nuovo iterabile contenente solo gli elementi dell'input per i quali la funzione restituisce *True*.

```
filter(funzione, iterabile)
```

Vediamo un esempio in cui controlliamo se un numero è pari o non lo è:

```
def numero_pari(n):
```

```
    return n % 2 == 0
```

```
numeri = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
numeri_pari = list(filter(numero_pari, numeri))

print(numeri_pari)

# output
[2, 4, 6, 8, 10]
```

La funzione **filter()** viene utilizzata per filtrare solo i numeri pari dall'elenco di numeri.

La funzione map()

La funzione **map()** permette di applicare una stessa funzione ad ogni elemento di un oggetto iterabile e restituisce un nuovo oggetto iterabile dello stesso tipo di quello di partenza contenente i risultati della funzione applicata a ciascun elemento.

```
map(function, iterable, ...)
```

È necessario specificare nell'argomento *function* la funzione che si desidera applicare a ciascun elemento dell'argomento *iterable*. Come indicano i puntini, la funzione può anche essere applicata a più oggetti iterabili contemporaneamente, basta passarli come argomento, e può essere utilizzata con qualsiasi funzione, anche definite dall'utente. Ad esempio, applichiamo a una lista di numeri una funzione che li triplica:

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

def triplica(n):
    return n * 3

numeri_triplicati = list(map(triplica, numeri))
print(numeri_triplicati)

# output
[3, 6, 9, 12, 15]
```

16. I Dizionari

In questa lezione parleremo di un nuovo tipo di dato di Python: **dict**, che significa **dictionary** (in italiano **dizionario**).

I dizionari di Python

I dizionari sono simili al tipo di dato lista di cui abbiamo già parlato nelle lezioni precedenti, ma a differenza di queste, in cui ad ogni elemento corrisponde un indice progressivo quindi da 0 in su, nei dizionari ad ogni elemento è associata una chiave **chiave**, che potrà essere usata per accedere al **valore** dell'elemento.

Siamo noi a scegliere che chiave usare, e possiamo usare qualsiasi tipo di dato eccetto liste e altri dizionari. Per ogni elemento all'interno del dizionario avremo quindi una **coppia chiave / valore**.

I dizionari di Python vengono definiti utilizzando una coppia di **parentesi graffe** che contenga al suo interno delle coppie chiave / valore separate da due punti:

```
>>> mio_dict = {"mia_chiave": "mio_valore", "age": 29, 3.14: "pi greco", "primi": [2, 3, 5, 7]}
>>> type(mio_dict)
```

output

```
<class 'dict'>
```

Quindi con questa azione abbiamo definito una struttura di dati contenente quattro coppie *chiave* : *valore*, associate al nome *mio_dizionario*. Come per tutte le variabili, possiamo accedere al valore in esse contenuto (nel nostro caso la struttura dati) tramite il nome stesso della variabile.

```
>>> mio_dict
```

output

```
{"mia_chiave": "mio_valore", "age": 29, 3.14: "pi greco", "primi": [2, 3, 5, 7]}
```

Dizionari e coppie chiave valore

Come vedete abbiamo utilizzato diversi tipi di valori, stringhe, interi e liste, associati ciascuno ad una chiave univoca. Le chiavi nei dizionari sono proprio il corrispettivo degli indici nelle Liste: usiamo le chiavi per richiamare i valori ad esse associati.

Per ottenere il valore associato alla chiave *mia_chiave_uno* facciamo in questo modo:

```
>>> mio_dict["mia_chiave"]
'mio_valore'
```

Stesso discorso vale per tutti gli altri valori presenti, richiamabili tramite le loro chiavi. Proviamo a richiamare il valore associato alla chiave 3.14:

```
>>> mio_dict[3.14]
```

```
'pi greco'
```

Per aggiungere un nuovo elemento al nostro dizionario facciamo in questa maniera:

```
>>> mio_dict["nuova_chiave"] = "nuovo_valore"
```

```
>>> mio_dict
```

```
{'mia_chiave': 'mio_valore', 'age': 24, 3.14: 'pi greco', 'primi': [2, 3, 5, 7], 'nuova_chiave': 'nuovo_valore'}
```

Allo stesso modo possiamo sostituire il valore associato ad una chiave già presente nel dizionario:

```
>>> mio_dict["age"] = 99
```

```
>>> mio_dict
```

```
{'mia_chiave': 'mio_valore', 'age': 99, 3.14: 'pi greco', 'primi': [2, 3, 5, 7], 'nuova_chiave': 'nuovo_valore'}
```

Come verificare la presenza di elementi nei dizionari

Per verificare se una chiave è presente o meno all'interno di un dizionario possiamo utilizzare gli operatori **in** e **not in**:

```
>>> "age" in mio_dict
```

```
True
```

```
>>> "zen" in mio_dict
```

```
False
```

Come rimuovere elementi da un dizionario

Per rimuovere una coppia chiave-valore possiamo utilizzare l'istruzione **del**:

```
>>> del mio_dict["mia_chiave"]
```

```
>>> mio_dict
```

```
{'age': 99, 3.14: 'pi greco', 'primi': [2, 3, 5, 7], 'nuova_chiave': 'nuovo_valore'}
```

I metodi dei Dizionari di Python

Come per altri tipi di dato, anche il tipo *dict* dispone di alcuni metodi propri: analizziamo i principali con un nuovo esempio.

Definiamo un nuovo dizionario associato a una variabile *ita_eng* e inseriamoci i valori tipici di un dizionario Italiano - Inglese

```
ita_eng = {"ciao": "hello", "arrivederci": "goodbye", "uova": "eggs", "gatto": "cat", "arancia": "orange"}
```

Esistono tre metodi associati ai dizionari che ci consentono di ottenere in output una lista di tutte le chiavi presenti, o di tutti i valori presenti, o di entrambi allo stesso tempo: **keys()**, **values()** ed **items()**.

Il metodo keys()

Keys è traducibile come "chiavi", ed è proprio il metodo che ci consente di ottenere una lista di tutte le chiavi presenti.

Quindi per ottenere una lista di tutte le chiavi presenti nel nostro dizionario *ita_eng* facciamo:

```
>>> ita_eng.keys()

# output

dict_keys(['ciao', 'arrivederci', 'uova', 'gatto', 'arancia'])
```

Metodo values()

Values significa valori, utilizziamo questo metodo per ottenere una lista di tutti i valori presenti:

```
>>> ita_eng.values()

# output

dict_values(['hello', 'goodbye', 'eggs', 'cat', 'orange'])
```

Il metodo items()

Infine utilizziamo il metodo **items()** per ottenere una lista di tutte le coppie chiavi-valore presenti.

```
>>> ita_eng.items()

# output

dict_items([('ciao', 'hello'), ('arrivederci', 'goodbye'), ('uova', 'eggs'), ('gatto', 'cat'), ('arancia', 'orange')])
```

Cosa restituiscono questi metodi?

Come possiamo interpretare dall'output, utilizzando questi metodi non ci vengono restituiti dei valori di tipi lista ma bensì, oggetti di tipo *dict_keys*, *dict_values* e *dict_items*.

Se abbiamo bisogno di una lista vera e propria possiamo utilizzare anche qui la nostra cara funzione **list()**:

```
>>> parole_eng = list(ita_eng.keys())

>>> parole_eng

['ciao', 'arrivederci', 'uova', 'gatto', 'arancia']
```

Dizionari e cicli for

Possiamo inoltre utilizzare questi metodi in combinazione con un ciclo **for**.

Ad esempio, per mandare in print tutte le chiavi del dizionario possiamo fare:

```
for chiave in ita_eng.keys():  
    print(chiave)
```

output

ciao

arrivederci

uova

gatto

arancia

Il metodo get()

Ma cosa succederebbe qualora provassimo a richiamare un valore associato a una chiave che non esiste nel nostro dizionario? Come è facile ipotizzare otterremmo un errore, e nello specifico un'eccezione del tipo *KeyError*, che se non gestita causerà il crash del programma.

```
>>> ita_eng["birra"]
```

output

Traceback (most recent call last):

File "stdin", line 1, in module

KeyError: 'birra'

Parleremo di gestione degli errori più avanti nel corso introducendo delle istruzioni dedicate.

Alcuni/e tra voi a questo punto staranno però pensando: be, abbiamo visto come usare le istruzioni if ed else, perché non usare queste due per evitare il crash del programma? Potremmo in effetti fare qualcosa di questo tipo:

```
if "birra" in ita_eng.keys():  
    print(ita_eng["birra"])  
else:  
    print("Chiave non trovata!")
```

output

Chiave non trovata!

Il metodo funziona ma richiede parecchie righe di codice che dobbiamo scrivere da noi! E ricordate come uno dei concetti più importanti di cui abbiamo discusso sia il riutilizzo del codice?

Fortunatamente, i dizionari di Python dispongono di un metodo specifico ideato proprio per situazioni come queste: il metodo **get()**. A questo passiamo due parametri: la chiave per il valore che stiamo cercando, e un valore di default come una sorta di "paracadute d'emergenza", qualora la chiave richiesta non esista.

Richiamiamo il metodo **get()** sul nostro dizionario *ita_eng* e passiamogli la chiave *birra*, e come valore di default la stringa "Chiave non trovata, mi spiace!"

```
>>> ita_eng.get("birra", "Chiave non trovata, mi spiace!")
```

output

```
'Chiave non trovata, mi spiace!'
```

Proviamo anche a chiamare il metodo passandoli una chiave stavolta esistente:

```
>>> ita_eng.get("ciao", "Chiave non trovata!!!")
```

output

```
'hello'
```

Il Metodo **setdefault()**

Parallelamente a quanto fatto col metodo **get()**, ci saranno dei momenti in cui a seconda della complessità di ciò che stiamo facendo, potremmo avere la necessità di creare una coppia chiave valore qualora una chiave non sia già presente e associata a un valore nel dizionario.

Restando sull'esempio precedente, supponiamo di voler verificare se sia presente la chiave *birra* nel nostro dizionario e di volerla aggiungere assieme ad un valore di default qualora non sia già presente

```
>>> ita_eng
```

```
{'ciao': 'hello', 'arrivederci': 'goodbye', 'uova': 'eggs', 'gatto': 'cat', 'arancia': 'orange'}
```

```
>>> ita_eng.setdefault("birra", "beer")
```

```
"beer"
```

```
>>> ita_eng
```

```
{'ciao': 'hello', 'arrivederci': 'goodbye', 'uova': 'eggs', 'gatto': 'cat', 'arancia': 'orange', 'birra': 'beer'}
```


Il metodo **setdefault()** ha anzitutto cercato la chiave *birra* all'interno del dizionario *ita_eng*, e visto che la chiave non esisteva ha creato una coppia chiave-valore col valore *beer* da noi passato!

17. Gestione degli Errori

In questa lezione impareremo a gestire eventuali errori che si possano manifestare nei nostri programmi, e per fare ciò parleremo delle istruzioni **try**, **except** e **finally**.

Eccezioni di Python e Messaggi di Errore

Parlando di errori mi riferisco in realtà a quelle **eccezioni** (in inglese **exception**) di Python che quando si manifestano, comportano spesso il crash del programma e la comparsa di un messaggio di errore. Otteniamo ad esempio un'eccezione del tipo **NameError**, provando a moltiplicare una variabile che non è stata definita.

```
>>> z * 5
```

```
NameError: name 'z' is not defined
```

Un'altra tipologia di errore ci viene mostrata se proviamo ad effettuare una divisione per zero, che restituisce **ZeroDivisionError**:

```
>>> 3/0
```

```
ZeroDivisionError: division by zero
```

Questi messaggi di errore sono molto comodi per noi sviluppatori, in quanto fornendoci dettagli relativi a un comportamento anomalo del nostro programma, ci permettono di **risolvere i vari bug** che inevitabilmente si presentano.

Ma non tutti gli errori sono bug.

Ad esempio è più che normale ottenere **ZeroDivisionError** provando a fare $3 / 0$.

In casi come questi è quindi importante gestire queste eccezioni, per rendere i nostri programmi più robusti e user friendly.

Le istruzioni try ed except in Python

Le istruzioni principali che vengono usate per questo scopo sono **try** ed **except**, che possono essere tradotte rispettivamente come **prova** e **eccetto** o **ad eccezione di**.

Volendo potete pensare a queste come a degli **if / else** ideati per la gestione delle eccezioni, in quanto anche in questo caso definiremo dei blocchi di codice distinti.

Il codice del blocco **try** verrà eseguito qualora tutto andasse liscio e senza errori, mentre il codice inserito nel blocco **except** verrà eseguito solamente qualora si dovesse manifestare l'eccezione ad esso associato.

Abbiamo qui di seguito una funzione moltiplicatore che richiede all'utente di inserire dei valori per le variabili *a* e *b*, ed effettua quindi una moltiplicazione.

Proviamo ad eseguire il codice e a forzarne il comportamento ipotizzato introducendo dei caratteri non numerici all'interno delle variabili *a* e *b*:

```
def moltiplicatore():  
    a = int(input('Inserisci il valore di a: '))  
    b = int(input('Inserisci il valore di b: '))  
    risultato = a * b  
    print(risultato)  
  
moltiplicatore()  
  
# output  
Inserisci il valore di a: eggs  
ValueError: invalid literal for int() with base 10: 'eggs'
```

Abbiamo ottenuto **ValueError**, ovvero errore di valore. Questo ci viene restituito perché la funzione **int()** non è in grado di convertire la stringa *eggs* in un numero intero.

Vediamo ora di usare le istruzioni **try** ed **except** per gestire questa eventualità, ed eseguiamo nuovamente il codice.

```
def moltiplicatore():  
    try:  
        a = int(input("Inserisci il valore di a: "))  
        b = int(input("Inserisci il valore di b: "))  
        risultato = a * b  
        print(risultato)  
    except ValueError:  
        print("Hey tu! solo caratteri numerici, grazie!")  
  
moltiplicatore()  
  
# output  
Inserisci il valore di a: bacon  
Hey amico! solo caratteri numerici, grazie!
```

Ed ecco che ora va decisamente meglio! Il nostro programma gestisce l'eccezione senza crashare, mostrando invece un messaggio informativo ai nostri utenti.

Quindi qualora siate in fase di progettazione di un programma, e notate che ci sono delle eventualità in cui, per quanto il programma sia ben strutturato, si arriva comunque a dei crash, come appunto in questi casi presentati, semplicemente copiate il nome dell'errore e usatelo assieme ad **except** per poter gestire questa eccezione. Così facendo il programma avrà tutto un altro livello di professionalità!

L'istruzione finally

Un'altra istruzione di Python utile in questi contesti è l'istruzione **finally**, traducibile in italiano come **alla fine** o **infine**. Come il nome stesso suggerisce, il codice definito nel blocco di **finally** verrà eseguito alla fine del programma qualsiasi cosa succeda, sia che si manifesti un errore oppure no.

Modifichiamo la nostra funzione usando questa istruzione:

```
def moltiplicatore():  
    try:  
        a = int(input("Inserisci il valore di a: "))  
        b = int(input("Inserisci il valore di b: "))  
        risultato = a * b  
        print(risultato)  
    except ValueError:  
        print("Hey tu! solo caratteri numerici, grazie!")  
    finally:  
        print("Sto chiudendo l'applicazione!")
```

moltiplicatore()

eseguiamo il codice cercando di innescare l'errore

Inserisci il valore di a: poker

Hey tu! solo caratteri numerici, grazie!

Sto chiudendo l'applicazione!

eseguiamo nuovamente il codice senza innescare l'errore

Inserisci il valore di a: 2

Inserisci il valore di b: 3

6

Sto chiudendo l'applicazione!

Come notate ora la stringa "Sto chiudendo l'applicazione!" viene mandata in print qualsiasi cosa succeda!

Nota: evitate di usare le istruzioni try ed except su tutto il codice di una funzione, ma cercate sempre di racchiuderci solamente il codice necessario

Come mostrare i messaggi di errore

Un'altra cosa che possiamo fare per migliorare il codice è mostrare in output il messaggio di errore.

Nel blocco dell'**except**, dopo il nome dell'eccezione che stiamo gestendo, possiamo aggiungere **as e**, e quindi mandare in output il valore di e tramite **print()**:

```
def moltiplicatore():  
    try:  
        a = int(input("Inserisci il valore di a: "))  
        b = int(input("Inserisci il valore di b: "))  
        risultato = a * b  
        print(risultato)  
    except ValueError as e:  
        print(f"Errore: {e}")  
        print("Hey tu! solo caratteri numerici, grazie!")  
    finally:  
        print("Sto chiudendo l'applicazione!")
```

moltiplicatore()

output

Inserisci il valore di a: spam

Errore: invalid literal for int() with base 10: 'spam'

Hey tu! solo caratteri numerici, grazie!

Sto chiudendo l'applicazione!

18. Ambienti Virtuali in Python con venv

Durante questo corso abbiamo imparato a usare varie funzioni integrate di Python e a importare i moduli della Standard Library. In questa lezione vedremo come sia possibile installare moduli esterni creati da altri developer, sia su **Windows** che su **Linux** o **Mac OS**, e nel fare ciò parleremo di **ambienti virtuali** e **PIP**.

Il Python Package Index

Il [Python Package Index](#) è un deposito di software scritto per Python.

Si tratta di una risorsa molto importante perché contiene al suo interno centinaia di migliaia di progetti scritti da varie organizzazioni o singoli developer, e non parlo solo di semplici progetti sviluppati in un fine settimana ma di librerie complesse e strutturate utilizzate da migliaia di aziende e altre attività, nel mondo reale. Da questo repository è possibile scaricare queste librerie gratuitamente e iniziare a creare software anche molto avanzato, senza dover reinventare la ruota.

Il comando pip list

I moduli del Python Package Index vengono installati da terminale, tramite un programma chiamato **PIP**, che dovrete già avere nel vostro computer in quanto è presente, tipicamente, nel pacchetto di installazione di Python. I vari moduli possono essere installati in due aree: a livello di sistema o in un ambiente virtuale.

Partiamo dall'analisi di quanto presente a livello di sistema: aprite quindi un terminale e date il comando **pip list**.

Nota: su sistemi Linux e Mac OS capita spesso di avere due versioni di Python installate (2.X e 3.X) e per questo motivo, a seconda della vostra personale configurazione potreste dover usare il comando **pip3** invece di **pip** per installare moduli utilizzabili con Python 3.

```
pip list
```

Se il comando ha funzionato a dovere, dovrete ora vedere un elenco di pacchetti installati a livello di sistema come ad esempio:

```
pip list
```

```
# output
```

```
Package          Version
-----
pip              22.0.4
setuptools       58.1.0
```

Attenzione: qualora non otteniate alcun elenco, potreste invece dover installare **pip**.

Su Windows la procedura più immediata prevede tipicamente la reinstallazione di Python secondo quanto mostrato nella [prima lezione](#). Nello specifico, fate attenzione alle varie spunte che selezionate!

Su Linux o WSL, il terminale dovrebbe restituirvi il comando opportuno da dare per installer pip a seconda del vostro sistema operativo. I comandi sono tipicamente i seguenti:

```
# comando tipico su Ubuntu e derivate
```

```
sudo apt install python3-pip
```

```
# comando tipico su Arch e derivate
```

```
sudo pacman -S python-pip
```

Installare package a livello di sistema è sconsigliato per vari motivi: uno di questi è il fatto che progetti diversi richiedono spesso versioni diverse degli stessi package e questo può causare conflitti, inoltre su alcuni sistemi sono richiesti permessi speciali per poter installare del nuovo software.

Ed è qui che entrano in gioco gli ambienti virtuali.

Cosa sono gli Ambienti Virtuali di Python?

Gli **Ambienti Virtuali** o **Virtual Environments** di Python sono uno strumento che permette di creare degli spazi indipendenti dal resto del sistema in cui è possibile testare e lavorare con Python e **pip**.

Concretamente avremo una cartella che conterrà vari file necessari al funzionamento dell'ambiente e una copia dei binary di Python, e dentro questa potremo poi installare tutti i moduli con la versione che vogliamo in base al progetto su cui stiamo lavorando, ma che richiederanno l'attivazione di questo ambiente per essere utilizzati.

Esistono vari strumenti per creare ambienti virtuali, e noi useremo uno chiamato **venv** che è presente di default con l'installazione di Python.

Creare Ambienti Virtuali Python su Windows, Linux e Mac OS

È importante sapere quale comando utilizzare per poter richiamare una shell di Python da terminale. Dovrete in sostanza usare lo stesso comando usato finora nel corso per eseguire gli script o per avviare una shell Python.

In generale, questo comando può essere: "python", "python3" o "py".

Oltre a questo comando, dovrete usare alcune istruzioni aggiuntive che informino Python che volete creare un ambiente virtuale, così come per eseguire uno script dobbiamo passargli il percorso che porta al file.py.

Spostatevi in una cartella dove avete permessi di scrittura (per testare va benissimo anche la cartella Documenti o Scrivania/Desktop) e date il comando:

ricorda: usa il comando usato finora per avviare una shell / eseguire uno script: python, python3 o py

```
python -m venv venv
```

Così facendo abbiamo detto a Python (comando: **python** o **python3** o **py**) che intendiamo usare il modulo **venv** (**-m venv**) per creare un ambiente virtuale in una nuova cartella chiamata **venv**. Avremmo potuto quindi cambiare la parte finale del comando per creare un ambiente in una cartella chiamata in qualsiasi modo.

Creare Ambienti Virtuali con Conda

Un altro sistema di gestione di pacchetti e di ambienti virtuali molto utilizzato è [Conda](#), che rispetto a **venv** offre diverse funzionalità aggiuntive, come la possibilità di creare ambienti virtuali con diverse versioni di Python o una gestione delle dipendenze avanzata. Per creare un nuovo ambiente virtuale tramite **Conda**, su Anaconda/Miniconda Prompt digitate:

```
conda create --name myenv
```

Sostituite *myenv* con un nome a vostra scelta. Se necessario potete indicare quale versione di Python specifica installare alla fine del comando:

```
conda create --name myenv python=3.11
```

Come attivare un Ambiente Virtuale di Python

Per poter usare un ambiente virtuale dobbiamo anzitutto attivarlo: il comando di attivazione è leggermente diverso su Windows rispetto a Linux e Mac OS.

Dal terminale di sistema, nel percorso contenente anche la cartella appena creata, date il comando:

```
# su Linux e Mac OS
```

```
source ./nome_cartella_ambiente/bin/activate
```

```
# su Windows, da PowerShell
```

```
.\nome_cartella_ambiente\Scripts\Activate.ps1
```

```
# su Windows, da cmd
```

```
nome_cartella_ambiente\Scripts\activate
```

Invece per Conda date il comando:

```
# Windows
```

```
conda activate myenv
```

```
# Linux
```



```
source activate myenv
```

Tenete conto che per usare un ambiente virtuale dovreste attivarlo ogni volta che intendete installare nuovi package o eseguire uno script che utilizzi i package in esso installati.

Vi ricordo che nel video allegato a questa lezione troverete tutte le istruzioni - passo passo.

Saprete che un ambiente virtuale è stato attivato correttamente quando vedrete la scritta **(venv)** all'inizio dell'output del vostro terminale.

Windows e l'errore relativo all'attivazione degli ambienti virtuali

Su Windows potrebbe capitare di ricevere un errore relativo all'esecuzione di Script sul sistema. Ci sono diverse soluzioni al riguardo, e tutte prevedono l'esecuzione di un comando specifico per dire a Windows che intendete eseguire comunque questo script.

Vi lascio [questo link dove vengono vagliate diverse opzioni](#), potendo scegliere tra queste una che più si addica alle vostre esigenze.

Come usare gli Ambienti Virtuali

A questo punto, se l'ambiente virtuale è stato attivato correttamente, dovreste essere pronti ad installarci dei nuovi package! Per fare ciò useremo **pip**.

Provate anzitutto a verificare che tutto funzioni correttamente dando nuovamente il comando **pip list**, dovreste ottenere un output di questo tipo:

```
(venv) utente@sistema:~/Desktop$ pip list
```

```
Package Version
```

```
-----
```

```
pip 22.0.2
```

```
setuptools 59.5.0
```

Come installare i package in un Ambiente Virtuale

Siamo ora pronti finalmente per installare dei package nel nostro sistema!

Per installare [Django, uno dei web framework Python più utilizzati](#) di cui parlo anche nei miei [corsi completi](#), date il comando:

```
pip install django
```

Volendo potete ora ridare il comando **pip list**, che dovrebbe mostrarvi stavolta tra i package installati, anche Django ed alcuni altri package.

```
pip list
```

```
# output
```

```
Package Version
```

```
-----
```

```
asgiref 3.5.2
```

```
Django 4.0.5
```

```
pip 22.0.2
```

```
setuptools 59.5.0
```

```
sqlparse 0.4.2
```

```
tzdata 2022.1
```

Ora che il package **Django** è stato installato, possiamo finalmente utilizzarlo, e potremo accederci da una shell interattiva interna all'ambiente virtuale, in maniera analoga a quanto facciamo con i moduli della Standard Library.

Ad ambiente attivo, avviate quindi una shell interattiva col comando **python**, e date i seguenti comandi che dovrebbero darvi un output simile a quello qui mostrato:

```
>>> import django
```

```
>>> django.VERSION
```

```
# output
```

```
(4, 0, 5, 'final', 0)
```

Come installare una versione specifica di un package Python

Potrà capitarvi di dover installare una versione specifica di un package, magari per questioni di compatibilità con altri elementi software utilizzati nello stesso progetto.

Utilizzando il comando visto sopra verrà però installata l'ultima versione disponibile.

Come fare per installare una versione specifica di un package in Python?

Supponiamo di voler installare la versione 0.25.1 di pandas.

Possiamo fare:

```
pip install pandas==0.25.1
```

E se ora diamo il comando `pip list`, ecco che notiamo come la versione effettivamente installata sia la 0.25.1, ovvero la versione specifica del package di cui abbiamo bisogno.

Tuttavia è molto importante precisare che sarà possibile installare solamente versioni di package compatibili con la versione di Python utilizzata per creare l'ambiente virtuale!

Ad esempio, nel nostro caso, volendo usare pandas 0.25.1 non sarà possibile usare versioni di Python recenti come la 3.12.

In tal caso potete sempre installare una versione compatibile di Python tramite il [py launcher](#) su Windows, oppure [pyenv](#) su Linux e Mac OS.

Come disattivare un Ambiente Virtuale

Così come vengono attivati, gli ambienti virtuali possono essere anche disattivati, ad esempio per riaccedere all'ambiente di sistema discusso in precedenza.

Per fare ciò è sufficiente dare il comando:

```
deactivate
```

A questo punto, la scritta **(venv)** dovrebbe essere sparita dall'output.

Nota: l'ambiente virtuale verrà inoltre disattivato alla chiusura del terminale di sistema, e dovrete quindi riattivarlo ogni volta.

Windows Subsystem for Linux (Python da WSL)

Potrebbe capitare per varie ragioni di avere la necessità di sviluppare un software su Linux anche se si utilizza Windows, soprattutto in caso si debba testare la compatibilità di un programma progettato per essere multi-piattaforma o per poter utilizzare librerie di terze parti che non supportano Windows.

Se state usando una versione di Windows uguale o superiore a 10 potete installare nel vostro computer una o più distribuzioni Linux a vostra scelta senza dover utilizzare necessariamente una macchina virtuale: installate la funzionalità **Windows Subsystem for Linux (WSL)** con PowerShell o con il prompt dei comandi di Windows digitando il seguente comando come amministratori:

```
wsl --install
```

Questo comando abiliterà le funzionalità necessarie per eseguire **WSL** e installare la distribuzione Ubuntu di Linux, che potrete poi utilizzare direttamente dal menù di PowerShell e sul terminale di **Visual Studio Code**.

Per ulteriori dettagli sull'installazione di **WSL** potete seguire la [guida ufficiale di Microsoft](#).

La procedura per l'uso di Python e venv sotto WSL è la stessa di Linux, e potete quindi usare i comandi relativi mostrati sopra.

Errori Comuni e Come Risolverli

Precisiamo che anzitutto, è fondamentale che leggiate sempre nel dettaglio i messaggi di errore che vi vengono mostrati, perché molte più volte che non, questi conterranno al loro interno anche i comandi necessari a poterli risolvere.

Tipicamente potreste ricevere dei messaggi di errore in queste situazioni:

- **Problema provando ad utilizzare pip.** In questo caso, potreste dover installare pip nel sistema. Come precisato sopra, su Windows potrebbe risultare più comodo re installare Python da zero seguendo la procedura mostrata nella prima lezione, mentre su Linux dovreste poter risolvere con l'installazione di un package extra da riga di comando
- **Problema in fase di creazione col comando "`python -m venv nome_ambiente`".** Questo errore è più comune su Linux che su Windows. Anche in questo caso la soluzione passa per l'installazione di un package extra e dovreste ottenere il comando preciso per il vostro sistema direttamente dal messaggio d'errore stesso. Di norma il comando, su distro Ubuntu e derivate (WSL incluso) è "`sudo apt install python3-venv`".