



FREE eBook

Le Basi di Python

Lorenzogovoni.com



Introduzione	4
1 Che cos'è Python?	5
1.1 A chi può servire	5
1.2 Installare Python	5
2 Tipi di dati in Python	11
2.1 Interi	11
2.2 Stringhe.....	11
2.3 Float.....	11
2.4 Tupla	12
2.5 Lista.....	14
2.6 Dizionari	21
2.7 Set.....	24
3 Operatori	29
3.1 Operatori aritmetici.....	29
3.2 Operatori di confronto	29
3.3 Operatori di assegnazione	30
3.4 Operatori binari.....	31
3.5 Operatori logici.....	32
3.6 Operatori di appartenenza	32
3.7 Operatori di identità	33
4 Funzioni	35
4.1 Sintassi	35
4.2 Built-in	37
4.3 User-defined	39
4.4 Funzioni anonime o lambda.....	40
4.5 Moduli.....	41
4.6 Pacchetti	45
5 Condizione if - else statement	46
6 Loops	51
6.1 Ciclo For	51
6.2 Ciclo While	54
6.3 Istruzioni Break e Continue	57
7 Oggetti e classi	61

7.1	Classe.....	62
7.2	Oggetto	63
7.3	Ereditarietà Python	66
8	Errori ed eccezioni.....	71
8.1	Errori sintattici.....	71
8.2	Errori logici	71
	Conclusione	79

Introduzione

Mi chiamo Lorenzo Govoni, autore del blog **lorenzogovoni.com**.

Già da qualche anno sto utilizzando Python come linguaggio di programmazione nella scrittura degli articoli e risoluzione dei problemi di machine learning.

Per questo motivo, ho creato questo ebook: per dare al lettore la possibilità di imparare ad usare Python, nel caso in cui voglia addentrarsi nel mondo del machine learning.

Sto strutturando articoli dando per scontato che tutti conoscano questo linguaggio: non è così, anche se esistono molte risorse online (gratuite). Pertanto ci tenevo a fare una guida di base e renderla disponibile a chiunque sia interessato ad apprendere il funzionamento.

In questo ebook verranno elencate le principali caratteristiche di questo linguaggio di programmazione. È adatto per chi è alle prime armi, o chi comunque vuole tenere ripassato questo valido strumento.

Tengo a precisare che puoi condividere questo PDF con chiunque ritieni possa trarne beneficio. Dopotutto, si tratta di un ebook non ufficiale e gratuito di Python creato per scopi didattici.

L'ebook sarà strutturato nel seguente modo. Nel primo capitolo verrà spiegato cos'è Python, per chi è adatto questo linguaggio di programmazione e come installarlo velocemente.

Nel secondo iniziamo ad addentrarci nei tipi di dati più comuni che vengono utilizzati da chi programma in Python.

Nel terzo capitolo sono trattati i vari operatori disponibili a cui si può far riferimento.

Nel quarto capitolo ci addentriamo nel mondo delle funzioni, per creare questi blocchi di codice organizzati e riutilizzabili.

Nel quinto capitolo viene trattato un altro argomento di interesse: le condizioni dette if-else, per far seguire ad un programma un'istruzione anziché un'altra.

Nel sesto capitolo si avrà a che fare con i cicli (Loops), per iterare un determinato numero di volte una o più condizioni.

Nel settimo capitolo è introdotta la programmazione ad oggetti, in particolare vengono definite le classi e gli oggetti e come poterle creare.

L'ebook viene concluso con l'ottavo capitolo, dove è enunciato come gestire i due tipi di errori comuni con cui si ha a che fare: gli errori di sintassi e di logica.

NB. Insieme all'ebook troverai i vari script utilizzati nel corso del documento. Sono stati eseguiti tramite Spyder, un potente ambiente scientifico scritto in Python, per Python, e progettato da e per scienziati, ingegneri e analisti di dati. Possono comunque essere eseguiti anche tramite l'ambiente Python standard, o l'idle che viene installato durante l'installazione di Python.

In aggiunta, troverai anche gli script in formato ipynb, utilizzabili nel notebook Jupyter, un altro strumento incredibilmente potente per lo sviluppo interattivo e la presentazione di progetti di scienza dei dati.

1 Che cos'è Python?

Sviluppato per la prima volta alla fine degli anni '80 da **Guido van Rossum**, Python è un linguaggio di codifica generico, il che significa che, a differenza di HTML, CSS e JavaScript, può essere utilizzato per altri tipi di programmazione e sviluppo software oltre che allo sviluppo web.

A causa della sua ubiquità e capacità di funzionare su quasi ogni architettura di sistema, Python è un linguaggio universale che si trova in una varietà di applicazioni diverse.

In linea generale, Python può essere usato per attività di:

- Sviluppo di app Web e mobile back-end (o lato server);
- Sviluppo di app desktop e software;
- Elaborazione di big data ed esecuzione di calcoli matematici;
- Scrittura di script di sistema (creazione di istruzioni che dicono a un sistema informatico di "fare" qualcosa).

Ma non lasciare che la vasta gamma di versatilità di Python ti spaventi. Proprio come quei linguaggi più familiari, Python è un linguaggio di programmazione su richiesta facile da imparare.

1.1 A chi può servire

Python è una stella nascente nel mondo della programmazione per due motivi principali: la vasta gamma di attività che può gestire, unita al fatto che in realtà è un linguaggio adatto a chi è alle prime armi con la programmazione. La sintassi del codice Python utilizza parole chiave inglesi e questo rende facile per chiunque capire e iniziare con la lingua.

Tuttavia, per quanto semplice sia la sintassi di Python, viene utilizzato per progetti dal suono complicato come l'intelligenza artificiale e l'apprendimento automatico. Ciò significa che Python è perfetto per una vasta gamma di utenti, tra cui:

- Programmatori principianti;
- Sviluppatori di app Web e mobile;
- Ingegneri del software;
- Scienziati dei dati;
- E chiunque altro lavori con o apprenda sulla programmazione del computer!

1.2 Installare Python

Python è un linguaggio di programmazione gratuito, open-source e multiplatforma, il che significa che può essere eseguito su più piattaforme come Windows, macOS, Linux ed è stato persino portato su macchine virtuali Java e .NET.

Anche se la maggior parte degli odierni Linux e Mac hanno preinstallato Python, la versione potrebbe non essere aggiornata. Quindi, è sempre una buona idea installare la versione più recente.

Per farlo:

1. Scarica l'ultima versione di Python al sito [www.Python.org](https://www.python.org) nella sezione downloads.
2. Esegui il file di installazione e segui i passaggi per installare Python.

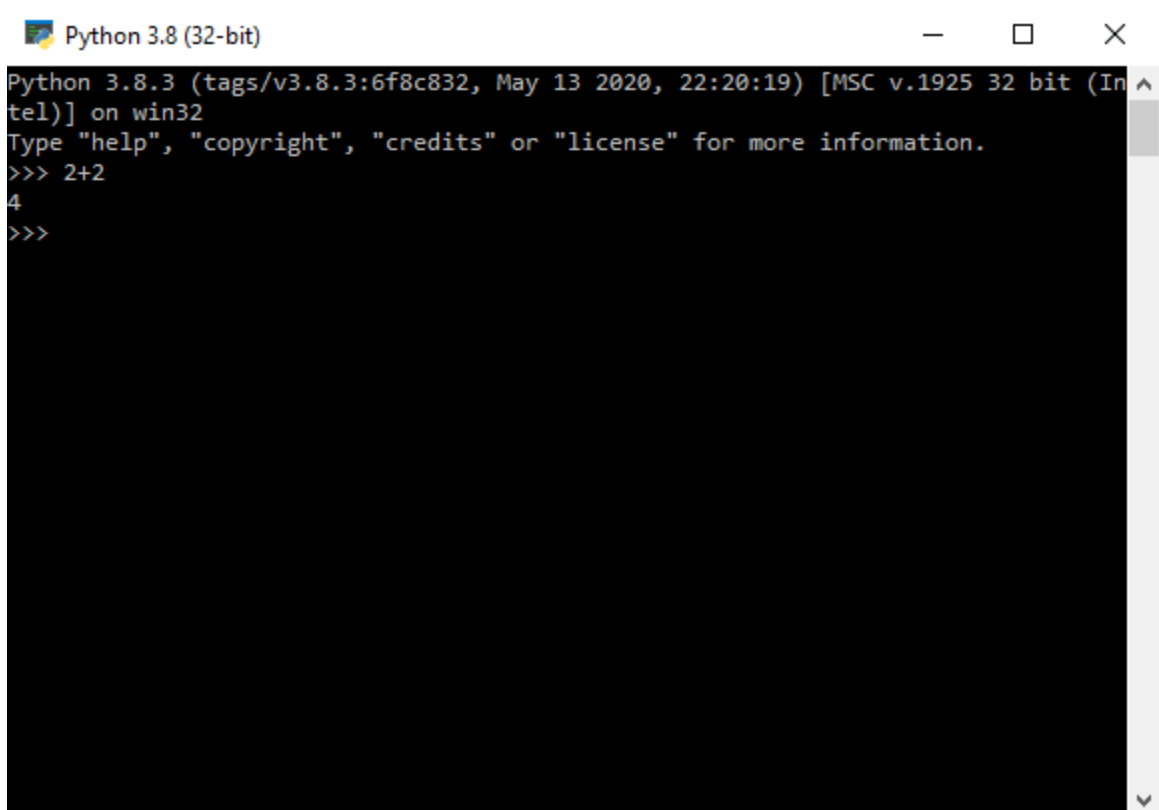
Durante il processo di installazione, seleziona Aggiungi Python alle variabili di ambiente: in questo modo potrai eseguire Python da qualsiasi parte del computer.

Gli esempi successivi, nonché gli script, saranno svolti su Windows.

1.2.1 Esegui Python in modalità immediata

Una volta terminato il processo di installazione, digitando Python nella barra di ricerca verrà richiamato l'interprete in modalità immediata. Possiamo digitare direttamente il codice Python e premere Invio per ottenere l'output. In alto possiamo vedere la versione installata (nell'esempio la 3.8.3).

Prova a digitare $2 + 2$ e premi Invio: otteniamo 4 come output. Questo prompt può essere utilizzato come una calcolatrice.



```
Python 3.8 (32-bit)
Python 3.8.3 (tags/v3.8.3:6f8c832, May 13 2020, 22:20:19) [MSC v.1925 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> 2+2
4
>>>
```

Figura 1: esempio operazione in Python per Windows

Per uscire da questa modalità, digitare **exit()** e premere Invio.

1.2.2 Esegui Python Integrated DeveLopment Environment (IDLE)

Possiamo usare qualsiasi software di modifica del testo per scrivere un file di script Python.

Dobbiamo solo salvarlo con l'**estensione .py**. Ma l'uso di un IDE può semplificarci molto la vita. IDE è un software che fornisce al programmatore funzioni utili come suggerimenti sul codice, evidenziazione e verifica della sintassi, esploratori di file, ecc. per lo sviluppo di applicazioni.

A proposito, quando installi Python, viene installato anche un IDE chiamato IDLE. Puoi usarlo per eseguire Python sul tuo computer. Quando apri IDLE dalla barra di ricerca, viene aperta una shell Python interattiva.

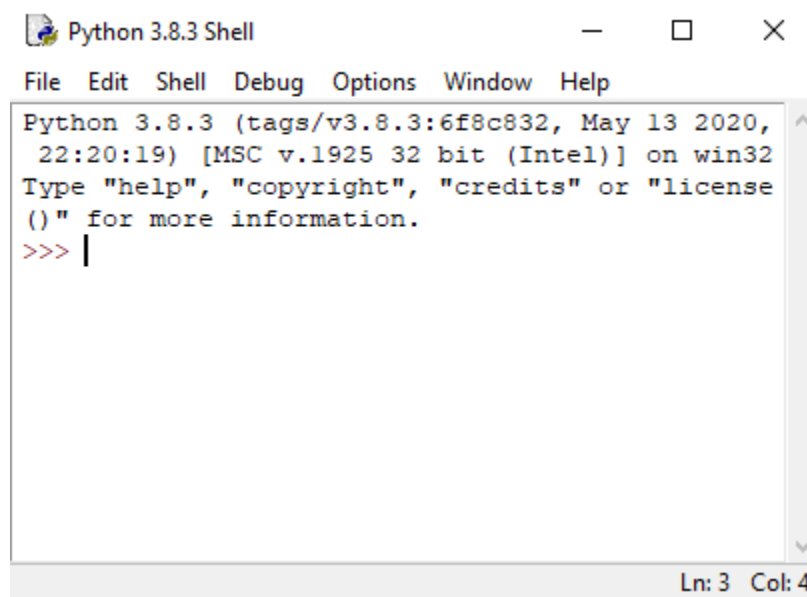


Figura 2: Python shell

Ora puoi creare un nuovo file e salvarlo con l'estensione **.py**. Ad esempio, **test1.py**. Per farlo clicca su file e poi su New File.

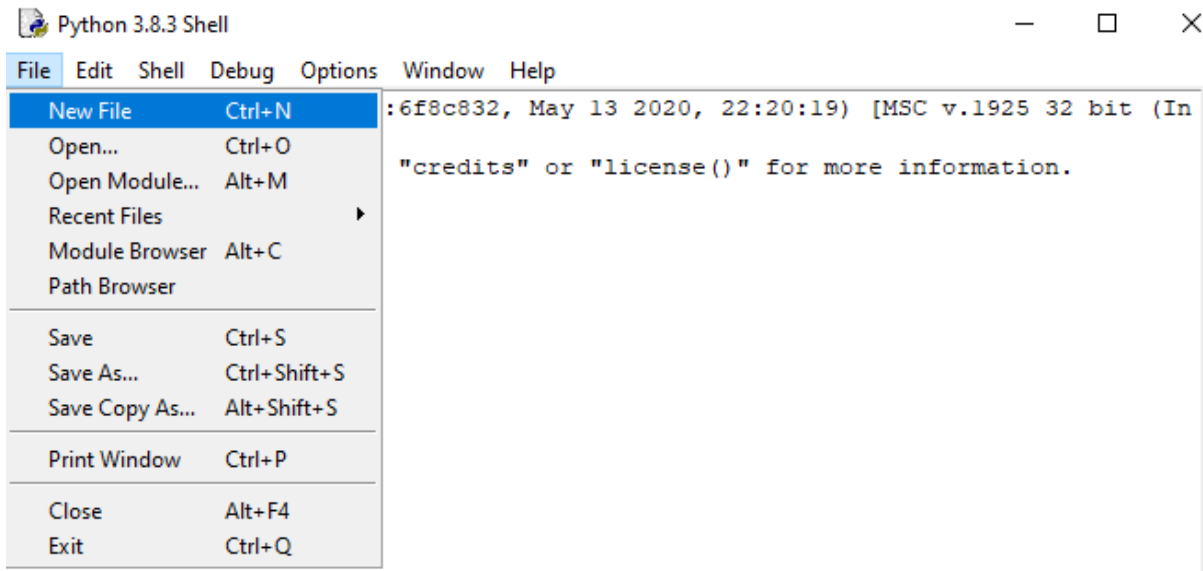


Figura 3: Creare il primo file Python

Scrivi il codice Python nel file e salvalo.

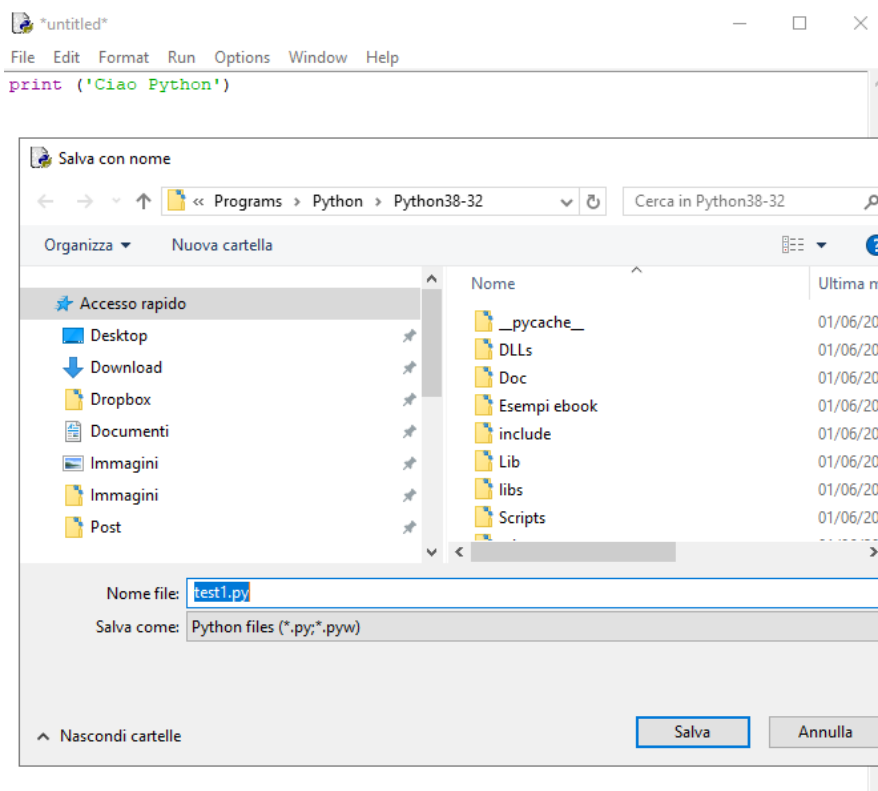


Figura 4: Salvare il file test1.py

Per eseguire il file vai su **Run** -> **Run module** o fai semplicemente clic su **F5**.

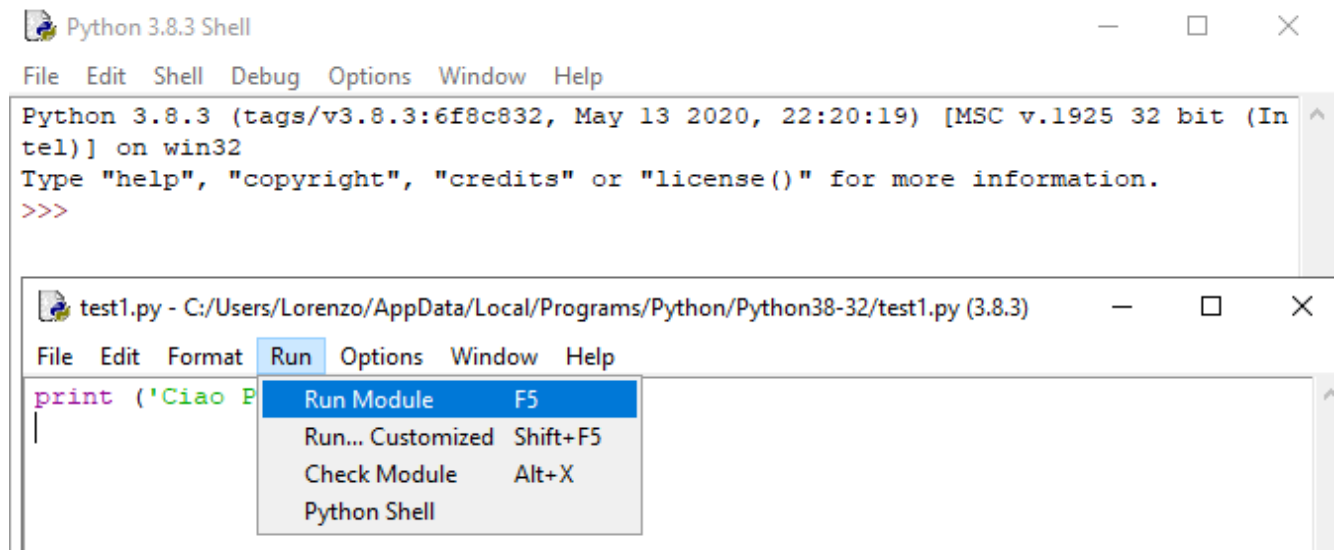


Figura 5: Eseguire il primo comando Python

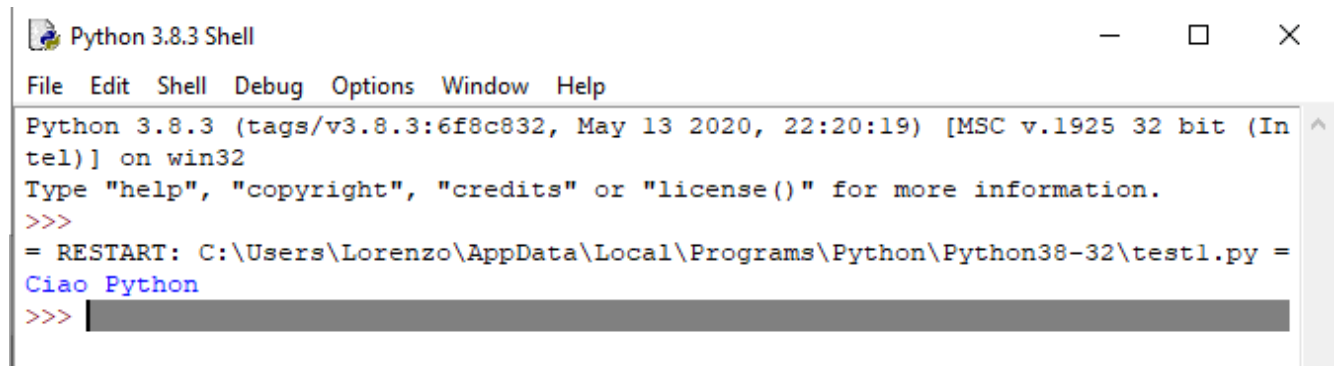
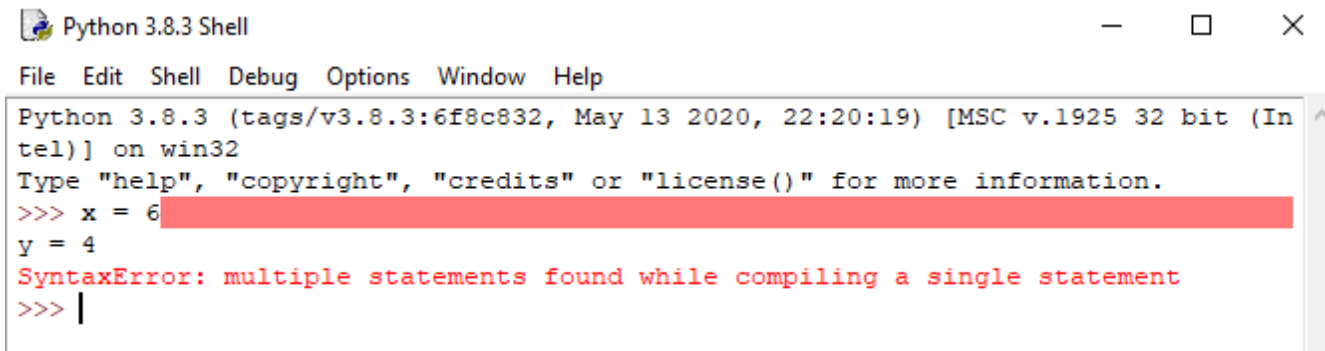


Figura 6: Risultato test1.py

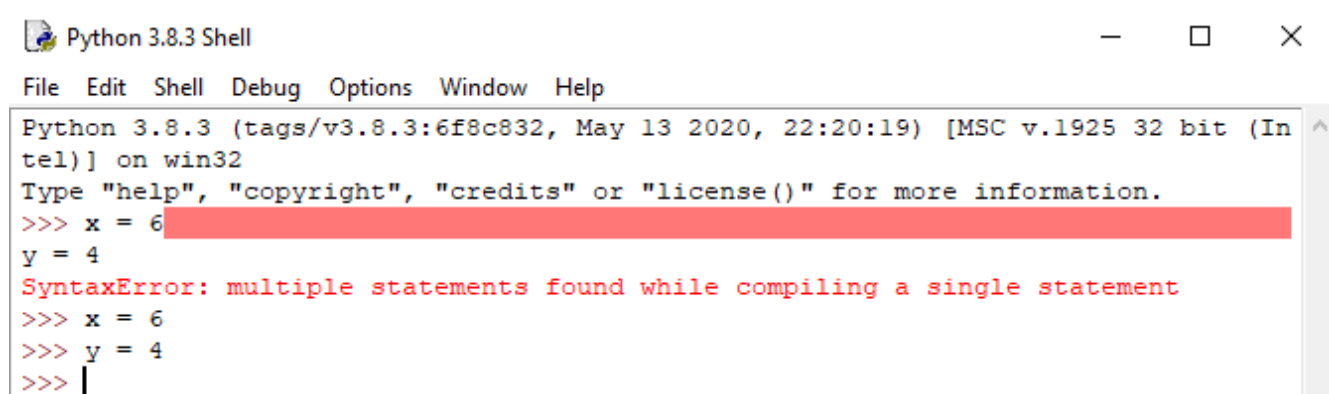
È bene notare che si può eseguire codice anche dal Python Shell, però un'operazione alla volta, altrimenti si otterrà errore.

A screenshot of a Python 3.8.3 Shell window. The window has a title bar with the text 'Python 3.8.3 Shell' and standard window controls. Below the title bar is a menu bar with 'File', 'Edit', 'Shell', 'Debug', 'Options', 'Window', and 'Help'. The main text area shows the following content: 'Python 3.8.3 (tags/v3.8.3:6f8c832, May 13 2020, 22:20:19) [MSC v.1925 32 bit (Intel)] on win32', 'Type "help", "copyright", "credits" or "license()" for more information.', and a prompt '>>>' followed by 'x = 6' on one line and 'y = 4' on the next line. A red error message 'SyntaxError: multiple statements found while compiling a single statement' is displayed below the input. The prompt '>>>' is followed by a vertical bar cursor.

```
Python 3.8.3 (tags/v3.8.3:6f8c832, May 13 2020, 22:20:19) [MSC v.1925 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> x = 6
y = 4
SyntaxError: multiple statements found while compiling a single statement
>>> |
```

Figura 7: Errore Python Shell con più di un comando

Si deve eseguire un comando alla volta, quindi x = 6 digitare invio, poi y = 4 e ridigitare invio, per continuare senza errori:

A screenshot of a Python 3.8.3 Shell window, similar to the one in Figure 7. The main text area shows the same header and prompt. The input sequence is now: '>>> x = 6' followed by a new line, and '>>> y = 4' followed by a new line. The prompt '>>>' is followed by a vertical bar cursor. No error message is present.

```
Python 3.8.3 (tags/v3.8.3:6f8c832, May 13 2020, 22:20:19) [MSC v.1925 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> x = 6
>>> y = 4
>>> |
```

Figura 8: Python Shell, eseguendo un'istruzione per volta

Bene, dopo questa breve introduzione, passiamo a vedere i tipi di dati con cui si avrà a che fare iniziando a programmare in Python.

2 Tipi di dati in Python

In Python ci si troverà a che fare con diverse tipologie di dati, vediamo le più comuni.

Puoi trovare i seguenti esempi negli script `Tipi_dati1.py` o `Tipi_dati1.ipynb`.

2.1 Interi

I numeri interi corrispondono all'insieme ottenuto unendo i numeri naturali (0, 1, 2, ...).

In Python si può associare una variabile ad un numero intero tramite un'uguaglianza. Ad esempio:

```
Z = 7
```

Z sarà una variabile di tipo intero.

2.2 Stringhe

Una stringa in informatica è una sequenza di caratteri con un ordine prestabilito. Per intenderci qualsiasi valore letterario o alfanumerico, come Xj45, 2j3b, test, gatto1, ecc.

È importante sottolineare che in Python questi tipi di caratteri vengono rappresentati con degli apici, o doppi apici: " o """. E pertanto indifferente scrivere la variabile x come:

```
x = "prova"
```

```
x = 'prova'
```

Entrambe saranno considerate Stringhe.

2.3 Float

Anche il termine float è molto utilizzato. Rappresenta un numero in virgola mobile (in inglese floating point) e in analisi numerica indica il metodo di rappresentazione approssimata dei numeri reali e di elaborazione dei dati usati dai processori per compiere operazioni matematiche.

Assegnare alla variabile y il valore 5,6 si può fare sempre con uguaglianza:

```
y = 5.6
```

La virgola però viene rappresentata con un punto, come da sistema di separazione decimale anglosassone e americano.

Pertanto scrivendo `y = 5,6` anziché `5.6`, avremmo creato una tupla.

2.4 Tupla

Una tupla è una **sequenza immutabile di oggetti Python**. Le tuple sono sequenze, proprio come gli elenchi. Creare una tupla è semplice, in quanto è sufficiente inserire diversi valori separati da virgola.

```
Tupla1 = 1,2,3,4,5,6
```

Si possono creare tuple in Python anche indicando i termini al suo interno da parentesi tonde, in questo modo:

```
Tupla1 = (1,2,3,4,5,6)
```

Oppure, da elementi tra apici, sempre separati da virgola:

```
Tupla2 = "a", "b", "c", "d".
```

Le tuple possono contenere sia valori numerici, che lettere, che alfanumerici. Le seguenti sono tutti esempi di tuple:

```
tupla3 = ('fisica', 'chimica', 2017, 2020);
```

```
tupla4 = ('1a', "2b", '3c', 4, 'e4x');
```

```
tupla5 = "a", "b", "c", "d";
```

Per poter creare una tupla con un elemento, avere un elemento solo tra parentesi non è sufficiente. Avremo bisogno di una virgola finale per indicare che in realtà è una tupla.

```
tupla6 = ('ciao',)
```

2.4.1 Indicizzazione tuple

Possiamo usare l'operatore index [] per accedere a un elemento in una tupla, dove l'indice inizia da 0.

Quindi, una tupla con 6 elementi avrà indici da 0 a 5. Cercare di accedere a un indice al di fuori dell'intervallo dell'indice di tupla (6,7, ... in questo esempio) genererà un **IndexError**.

L'indice deve essere un numero intero, quindi non possiamo usare float o altri tipi. Ciò comporterà un errore chiamato **TypeError**.

```
tupla3[2]
```

```
# Output: 2017
```

```
tupla3[4]
```

```
# Output: Tuple index out of range
```

Allo stesso modo, alle tuple nidificate si accede utilizzando l'indicizzazione nidificata, come mostrato nell'esempio seguente:

```
tupla7 = (('parole', 'studio'), (1,2,4), [3,6,19])
```

digitando `tupla7[0]` ottengo come risultato la prima tupla ('parole', 'studio').

Al contrario se volessi ottenere solo un elemento della singola tupla posso digitare ad esempio `tupla7[0][1]` per ottenere della prima tupla il secondo valore ('studio').

2.4.2 Indicizzazione negativa

Python consente l'indicizzazione negativa per le sue sequenze.

L'indice di -1 si riferisce all'ultimo elemento, -2 al penultimo elemento e così via.

`Tupla3[-2]` restituisce sempre 2017.

2.4.3 Slicing

Possiamo accedere a una gamma di elementi in una tupla usando i **due punti** dell'operatore slicing.

`Tupla1[1:4]` -> risultato (2,3,4)

#Restituisce dal secondo al 4 valore della tupla

`Tupla1[:]` -> risultato (1,2,3,4,5,6)

#Restituisce tutti i valori della tupla

`Tupla1[2:]` -> risultato (3,4,5,6)

#Restituisce dal secondo valore fino all'ultimo della tupla

2.4.4 Eliminare una tupla

Non è possibile eliminare un elemento della tupla (a meno che non rientri tra parentesi quadre), ma solamente una tupla per intero con il **comando del** seguito dal nome della tupla da eliminare. Esempio:

`del tupla2`

2.5 Lista

Una lista (detta anche elenco) è una struttura dati in Python che è una **sequenza di elementi ordinabile mutabile o modificabile**. Ogni valore all'interno di una lista viene chiamato *elemento*. Proprio come le stringhe sono definite come caratteri tra virgolette, le liste sono definite con valori tra parentesi quadre []. I numeri all'interno della lista possono essere scritti senza virgolette.

Un esempio di lista è la seguente:

```
listadiprova = ['a','b','c',12,3,5]
```

Le differenze tra tuple ed elenchi sono che le tuple non possono essere cambiate a differenza degli elenchi e le tuple usano parentesi tonde, mentre gli elenchi usano parentesi quadre.

Poiché le tuple sono abbastanza simili alle liste, entrambe vengono utilizzate in situazioni simili. Tuttavia, ci sono alcuni vantaggi nell'implementare una tupla rispetto ad un elenco.

Di seguito sono elencati alcuni dei principali vantaggi:

- Generalmente utilizziamo le tuple per tipi di dati eterogenei (diversi) e le liste per tipi di dati omogenei (simili).
- Poiché le tuple sono immutabili, l'iterazione attraverso una tupla è più rapida rispetto alla lista. Quindi c'è un leggero aumento delle prestazioni.
- Le tuple che contengono elementi immutabili possono essere utilizzate come chiave per un dizionario. Con le liste, questo non è possibile.
- Se disponi di dati che non cambiano, implementarli come tupla garantirà che rimangano protetti da scrittura.

Le liste possono contenere anche delle sottoliste, come di seguito:

```
lista2 = [['ab','2d'], [3,9], [7, 'c']]
```

La lista2 contiene 3 sottoliste: ['ab','2d'], [3,9] e [7,'c'].

Le operazioni di indicizzazione, indicizzazione negativa e slicing sono valide anche per le liste.

Come abbiamo poi accennato, le liste sono mutabili, il che significa che i loro elementi possono essere cambiati a differenza delle tuple.

2.5.1 Aggiungere elementi ad una lista

Possiamo utilizzare l'operatore di assegnazione (=) per modificare un elemento o un intervallo di elementi.

```
listadiprova[1] = 'paperino'
```

```
print(listadiprova)
```

Output: ['a', 'paperino', 'c', 12, 3, 5]

```
listadiprova[2:4] = [11,15]
```

```
print (listadiprova)
```

```
# Output: ['a', 'paperino', 11, 15, 3, 5]
```

È possibile aggiungere un elemento a un elenco utilizzando il **metodo append()** o aggiungere più elementi utilizzando il **metodo extend()**:

```
listadiprova.append(25)
```

```
print (listadiprova)
```

```
# Risultato: ['a', 'paperino', 11, 15, 3, 5, 25]
```

```
listadiprova.extend([1,47,78])
```

```
print (listadiprova)
```

```
# Risultato: ['a', 'paperino', 11, 15, 3, 5, 25, 1, 47, 78]
```

Possiamo anche usare l'operatore + per combinare due elenchi. Questo è anche chiamato concatenazione.

Ad esempio:

```
lis = [1,2]
```

```
print(lis + [3,4])
```

```
# Risultato: [1,2,3,4]
```

L'operatore * ripete un elenco per il numero di volte indicato. Ad esempio se ripetiamo 3 volte la lista lis si può usare il simbolo del per:

```
print(lis * 3)
```

```
# Risultato: [1, 2, 1, 2, 1, 2]
```

Inoltre, possiamo inserire un elemento nella posizione desiderata usando il **metodo insert()** o inserire più elementi in una sezione vuota di un elenco. Il primo parametro indica la posizione dove inserire il valore, mentre quest'ultimo è indicato dal secondo parametro.

Ad esempio con la seguente riga vado ad inserire il valore 5 nella seconda posizione, in quanto si parte da 0:

```
lis.insert(1,5)

print(lis)

# Risultato: [1,5,2]
```

lo stesso risultato lo si può ottenere anche nel seguente modo:

```
lis = [1,2]

lis[1:1] = [5]

print(lis)

# Output: [1, 5, 2]
```

È possibile anche inserire più di un valore indicando la posizione di inserimento:

```
lis[2:2] = [4,3,7]

print(lis)

# Risultato: [1,5,4,3,7,2]
```

La posizione 2, non era l'ultima (perché c'era il valore 2) quindi i valori vengono inseriti a partire da quella posizione e il 2 viene spostato alla posizione 5 (6° valore).

2.5.2 Eliminare il valore di una lista

Possiamo eliminare uno o più elementi da un elenco utilizzando la parola chiave `del`. Può persino eliminare completamente l'elenco.

Ad esempio dalla lista `lis` di prima `[1,5,4,3,7,2]` cancelliamo il 4 elemento:

```
del lis[3]

print(lis)

# Output: [1, 5, 4, 7, 2]
```

Possiamo cancellare anche più di un valore contemporaneamente:

```
del lis[2:4]

print(lis)
```



```
# Output: [1, 5, 2]
```

Per cancellare interamente la lista basta digitare del seguito dal nome della lista:

```
del lis
```

Possiamo usare il **metodo remove()** per rimuovere l'oggetto dato, ad esempio: `lis2 = ["a", 0,14, "abc", "gatto", 678, 95, "1gh"]`

```
lis2.remove (0)
```

```
Print (lis2)
```

```
# Output: ['a', 14, 'abc', 'gatto', 678, 95, '1gh']
```

Il **metodo pop()** invece può essere utilizzato per rimuovere un oggetto nell'indice dato. Il metodo `pop()` rimuove e restituisce l'ultimo elemento se non viene fornito l'indice. Questo ci aiuta a implementare gli elenchi come stack (struttura dei dati first in, last out).

```
print(lis2.pop(0))
```

```
# Output: a
```

```
print(lis2.pop())
```

```
# Output: 1gh
```

Infine, possiamo anche usare il **metodo clear()** per svuotare un elenco.

```
Lis2.clear()
```

```
print(lis2)
```

```
# Output: []
```

2.5.3 Metodi più comuni per le liste

La tabella sotto elenca alcuni metodi comuni in Python per le liste.

Metodo	Descrizione
append()	Aggiunge un elemento alla fine dell'elenco
extend()	Aggiunge tutti gli elementi di un elenco a un altro elenco
insert()	Inserisce un elemento nell'indice definito
remove()	Rimuove un elemento dall'elenco
pop()	Rimuove e restituisce un elemento in corrispondenza dell'indice specificato
clear()	Rimuove tutti gli elementi dall'elenco
index()	Restituisce l'indice del primo elemento corrispondente
count()	Restituisce il conteggio del numero di elementi passati come argomento
sort()	Ordina gli elementi in un elenco in ordine crescente
reverse()	Inverte l'ordine degli elementi nell'elenco
copy()	Restituisce una copia dell'elenco

Vediamo ora altri metodi utilizzati.

2.5.4 Type()

Python ha un metodo integrato chiamato **type** che generalmente è utile per capire il tipo di variabile utilizzata nel programma in fase di esecuzione.

Se un singolo argomento (oggetto) viene passato a type() incorporato, restituisce il tipo dell'oggetto specificato.

```
x = 23
```

```
type(x)
```

```
# Output: Int
```

```
x1 = 'prova'
```

```
type(x1)
```

```
# Output: Str
```

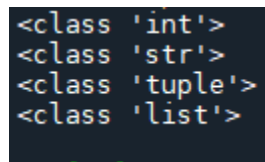
Il risultato nel primo caso darà un intero, mentre nel secondo caso una stringa.

2.5.5 Print()

La funzione **print()** stampa il messaggio specificato sullo schermo o su un altro dispositivo di output standard. Il messaggio può essere una stringa o qualsiasi altro oggetto, l'oggetto verrà convertito in una stringa prima di essere scritto sullo schermo. Ad esempio:

```
x = 4
s = "Ciao"
d = (2,4,'b')
y = [1,'c',3]
print(type(x))
print(type(s))
print(type(d))
print(type(y))
```

Il risultato sarà il seguente, dove ci dice che la variabile x è un intero, la variabile s una di tipo stringa, d una tupla e y una lista.



```
<class 'int'>
<class 'str'>
<class 'tuple'>
<class 'list'>
```

Figura 9:Esempio metodo Type()

2.5.6 Dir()

Dir() è una potente funzione integrata in Python3, che restituisce un elenco degli attributi e dei metodi di qualsiasi oggetto (ad esempio funzioni, moduli, stringhe, elenchi, dizionari ecc.)

Ad esempio incorporando una tupla all'interno di dir, otterremo tutte le funzioni che si possono utilizzare per la tupla.

Richiamando la tupla2, creata sopra, avremo (per risparmiare spazio i parametri sono stati elencati uno di fianco all'altro):

```
dir(tupla2)
```

Risultato:

```
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
 '__getattr__', '__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__', '__init_subclass__',
```

```
'__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',  
'__repr__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'count', 'index']
```

Ad esempio il metodo `__add__()` permette di aggiungere un nuovo valore ad una tupla.

```
Tupla1.__add__((85,))  
(1, 2, 3, 4, 5, 6, 85)
```

Figura 10: Esempio metodo `__add__()`

È possibile contare quante volte compaiono i valori in una tupla col metodo `count()`. Ad esempio nella Tupla1 se contiamo il valore 4 (come qualsiasi altro otteniamo 1 come risultato):

```
Tupla1.count(4)  
1
```

Figura 11: Esempio metodo `count()`

Col metodo `index()` viene restituita la posizione del valore richiamato della tupla. Allo stesso modo se cerchiamo l'indice del valore 4 troviamo 3 perché collocato in tale posizione nella Tupla1:

```
Tupla1.index(4)  
3
```

Figura 12: Esempio metodo `index()`

Possiamo anche richiamare una lista con la funzione `dir`:

```
Dir(listadiprova)
```

Risultato:

```
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__', '__eq__',  
'__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__', '__iadd__', '__imul__',  
'__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__',  
'__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__',  
'__sizeof__', '__str__', '__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop',  
'remove', 'reverse', 'sort']
```

Ad esempio `__delitem__()` cancella il valore dalla posizione indicata dall'argomento:

```
listadiprova = ['a','b','c',12,3,5]
```

```
listadiprova.__delitem__(2)
```

Figura 13: Esempio metodo `__delitem__()`

Da lista di prova viene cancellato il secondo valore ('c'), come risultato si ottiene:

```
listadiprova  
['a', 'b', 12, 3, 5]
```

Figura 14: Risultato metodo `__delitem__()`

2.6 Dizionari

Puoi trovare i seguenti esempi negli script `Tipi_dati2.py` o `Tipi_dati2.ipynb`.

Il dizionario Python è una **raccolta di oggetti non ordinata**. Mentre altri tipi di dati composti hanno valore solo come elemento, un dizionario viene definito da una (o più) **coppia chiave - valore**. La chiave è rappresentata dall'elemento che precede i due punti, mentre il valore corrispondente è dislocato dopo i due punti.

Creare un dizionario è semplice, perché occorre posizionare gli oggetti tra parentesi graffe {} separati da virgola.

Mentre i valori possono essere di qualsiasi tipo dati e possono essere ripetuti, le chiavi devono essere di tipo immutabile (stringa, numero o tupla con elementi immutabili) e devono essere univoci.

Ad esempio un semplice dizionario può essere:

```
diz = {1: 'mela', 2: 'arancia', 3: 'pera'}
```

1,2 e 3 sono le chiavi (di tipo intero), mentre mela, arancia e pera sono i valori.

I dizionari possono avere anche valori misti, eccone un esempio:

```
diz2 = {'Nome': 'Giacomo', 1: [2, 4, 3]}
```

Possiamo anche creare un dizionario usando la funzione integrata `dict()`:

```
dict1 = dict({'a': 'test1', 'b': 'test2', 'c': 'test3'})
```

Per accedere ai valori, il dizionario utilizza le chiavi. La chiave può essere utilizzata tra parentesi quadre o con il metodo `get()`.

Ad esempio se volessimo visualizzare il valore della chiave a, potremmo digitare:

```
print(dict1['a'])  
  
# Risultato: test1
```

```
print(dict1.get('b'))  
  
# Risultato: test2
```

La differenza durante l'utilizzo di `get()` è che restituisce `None` anziché `KeyError`, se la chiave non viene trovata.

```
print(dict1.get('d'))  
  
# Risultato: None
```

2.6.1 Aggiungere/modificare un elemento di un dizionario

I dizionari sono mutabili. È possibile aggiungere nuovi articoli o modificare il valore degli articoli esistenti utilizzando l'operatore di assegnazione. Se la chiave è già presente, il valore viene aggiornato, altrimenti viene inserita una nuova chiave: la coppia valore viene aggiunta al dizionario.

```
dict1['d'] = 'giocare a palla'
```

Stampando il risultato vediamo la chiave e il valore aggiunti al dizionario:

```
print(dict1)  
  
# Risultato: {'a': 'test1', 'b': 'test2', 'c': 'test3', 'd': 'giocare a palla'}
```

Per modificare il valore basta aggiornare il valore della chiave già esistente nel seguente modo:

```
dict1['b'] = 'nuotare in piscina'  
  
print (dict1)  
  
#Risultato {'a': 'test1', 'b': 'nuotare in piscina', 'c': 'test3', 'd': 'giocare a palla'}
```

2.6.2 Eliminare o rimuovere elementi da un dizionario

Possiamo rimuovere un particolare elemento in un dizionario usando il **metodo `pop()`**. Questo metodo rimuove il valore con la chiave fornita.

```
dict1.pop('c')
```

Eliminiamo così il 3° elemento del dizionario:

```
print(dict1)
```

```
# Risultato: {'a': 'test1', 'b': 'nuotare in piscina', 'd': 'giocare a palla'}
```

Il **metodo popitem()** può essere utilizzato per rimuovere e restituire un elemento arbitrario (chiave, valore) dal dizionario.

```
dict1.popitem()
```

Vediamo cosa abbiamo cancellato:

```
print(dict1)
```

```
# Risultato: {'a': 'test1', 'b': 'nuotare in piscina'}
```

Tutti gli elementi possono essere rimossi contemporaneamente usando il **metodo clear()**.

```
Dict1.clear()
```

```
Print(dict1)
```

```
# Risultato: {}
```

Possiamo anche usare la parola chiave **del** per rimuovere singoli elementi o l'intero dizionario stesso.

```
del(diz1[2])
```

```
print(diz1)
```

```
# Risultato: {1: 'mela', 3: 'pera'}
```

2.6.3 Controllare i valori in un dizionario

Possiamo verificare se una chiave si trova in un dizionario tramite la parola chiave **in**. Si noti che il test di appartenenza è solo per le chiavi, non per i valori.

Quindi ad esempio digitando:

```
print(1 in diz1) -> otteniamo True
```

```
print(2 in diz1) -> otteniamo False (perché cancellato con l'istruzione del precedente).
```

2.6.4 Controllare la lunghezza di un dizionario

Si può utilizzare il **metodo len()** per ottenere la grandezza di un dizionario.

```
Print(len(diz2))
```

Risultato: 2

2.6.5 Metodi comuni per i dizionari

Vediamo ora alcuni metodi comuni per i dizionari:

Metodo	Descrizione
<code>clear()</code>	Rimuove tutti gli elementi dal dizionario.
<code>copy()</code>	Restituisce una copia del dizionario.
<code>get(key [, d])</code>	Restituisce il valore della chiave. Se la chiave non esce, restituisce il valore di default d (che è settato a None).
<code>items()</code>	Restituisce un nuovo oggetto con gli elementi del dizionario in formato (chiave, valore).
<code>keys()</code>	Restituisce un nuovo oggetto con le chiavi del dizionario.
<code>pop(key [, d])</code>	Rimuove l'elemento con chiave e restituisce il suo valore o d se la chiave non viene trovata. Se il valore di default d non viene fornito e la chiave non viene trovata, genera <code>KeyError</code> .
<code>popitem()</code>	Rimuove e restituisce un oggetto arbitrario (chiave, valore). Genera <code>KeyError</code> se il dizionario è vuoto.
<code>setdefault(key [, d])</code>	Se key è nel dizionario, restituisce il suo valore. In caso contrario, inserisce la chiave con un valore di default d e lo restituisce (il valore predefinito è None).
<code>update([other])</code>	Il metodo <code>update()</code> aggiunge elementi al dizionario se la chiave non è nel dizionario. Se la chiave si trova nel dizionario, aggiorna la chiave con il nuovo valore.
<code>values()</code>	Restituisce un nuovo oggetto con i valori del dizionario

2.7 Set

Un set è una **raccolta non ordinata di elementi**. Ogni elemento è unico (non sono presenti quindi duplicati) e deve essere immutabile (ossia non modificabile).

Tuttavia, il set stesso è modificabile. Possiamo aggiungere o rimuovere elementi da un set.

I set possono essere utilizzati per eseguire operazioni matematiche come unione, intersezione, differenza simmetrica ecc.

Un set viene creato posizionando tutti gli elementi all'interno di parentesi graffe {}, separati da una virgola o utilizzando la **funzione set()** incorporata.

Può avere un numero qualsiasi di elementi e possono essere di diversi tipi (intero, float, tupla, stringa ecc.). Ma un set non può avere elementi mutabili come liste, altri set al suo interno o dizionari come suoi elementi.

Ad esempio:

```
set1 = {1, 2, 9}
print(set1)
# Risultato: {1, 2, 9}
```

```
set2 = {7.4, "Ciao", (8, 9, 5)}
print(set2)
# Risultato: {'Ciao', (8, 9, 5), 7.4}
```

Come anticipato i set non hanno duplicati, quindi se ho il set3 composto da:

```
set3 = {1,1,2,2,3,3}
print(set3)
# Risultato: {1,2,3}
```

In aggiunta dentro un oggetto set non si può aggiungere una lista, se no si otterrà un errore:

```
set4= {5, 9, ["a",4]}
print(set4)
# Risultato: unhashable type: 'list'
```

Creare un set vuoto è un po' complicato. Le parentesi graffe vuote {} creeranno un dizionario vuoto in Python. Per creare un set senza alcun elemento, utilizziamo la funzione set() senza alcun argomento.

```
set5 = {}
print(type(set5))
# Risultato: class 'dict'
```

```
set6 = set()

print(type(set6))

# Risultato: class 'set'
```

2.7.1 Modifica di un set in Python

Abbiamo visto che i set sono mutabili. Tuttavia, poiché non sono ordinati, l'indicizzazione non ha alcun significato.

Non è possibile accedere o modificare un elemento di un set utilizzando l'indicizzazione o lo slicing. Impostare il tipo di dati non lo supporta.

Possiamo aggiungere un singolo elemento usando il **metodo add()** e più elementi usando il **metodo update()**. Il metodo update() può prendere tuple, liste, stringhe o altri insiemi come argomento. In tutti i casi, i duplicati vengono evitati.

```
set1.add(3)

print(set1)

# Risultato: {1,2,3,9}
```

```
set1.update([2, 3, 4])

print(set1)

# Risultato: {1,2,3,4,9}
```

2.7.2 Rimozione di elementi da un set

Un elemento particolare può essere rimosso da un set usando i **metodi discard()** e **remove()**.

L'unica differenza tra i due è che la funzione discard() lascia invariato un set se l'elemento non è presente nel set. D'altra parte, la funzione remove() genererà un errore in tale condizione (se l'elemento non è presente nel set).

L'esempio seguente lo illustrerà.

```
set7 = {1, 2, 3, 4, 6, 9}

print(set7)

# Risultato: {1, 2, 3, 4, 6, 9}
```

```
set7.discard(3)

print(set7)
```

```
# Risultato: {1, 2, 4, 6, 9}
```

```
set7.remove(6)
```

```
print(set7)
```

```
# Risultato: {1, 2, 4, 9}
```

se l'elemento non è presente ottengo un errore con il metodo remove, ma con discard otterrò il set iniziale

```
set7.discard(5)
```

```
print(set7)
```

```
# Risultato: {1, 2, 4, 9}
```

```
set7.remove(5)
```

```
# Risultato: KeyError: 5
```

Allo stesso modo, possiamo rimuovere e restituire un oggetto usando il **metodo pop()**.

Poiché set è un tipo di dati non ordinato, non è possibile determinare quale elemento verrà visualizzato. È completamente arbitrario. Di conseguenza, potresti ottenere output a questi script diversi da quelli qua definiti.

```
set8 = set("Ciao")
```

```
print(set8)
```

```
# Risultato: {'a', 'o', 'C', 'i'}
```

```
print(set8.pop())
```

```
# Risultato: a
```

```
set8.pop()
```

```
print(set8)
```

```
#Risultato: {'C', 'i'}
```

Possiamo anche rimuovere tutti gli elementi da un set usando il **metodo clear()**.

```
set8.clear()
```

```
print(set8)
```

```
# Risultato: set()
```

2.7.3 Altre operazioni set

Vediamo ora una panoramica dei metodi più diffusi utilizzabili con l'oggetto set:

Metodo	Descrizione
add()	Aggiunge un elemento al set
clear()	Rimuove tutti gli elementi dal set
copy()	Restituisce una copia del set
difference()	Restituisce la differenza di due o più set come nuovo set
difference_update()	Rimuove tutti gli elementi di un altro set da questo set
discard()	Rimuove un elemento dal set se è un membro. (Non fa nulla se l'elemento non è nel set)
intersection()	Restituisce l'intersezione di due set come nuovo set
intersection_update()	Aggiorna il set con l'intersezione tra sé stesso e un altro
isdisjoint()	Restituisce True se due set hanno un'intersezione nulla
issubset()	Restituisce True se un altro set contiene questo set
issuperset()	Restituisce True se questo set contiene un altro set
pop()	Rimuove e restituisce un elemento set arbitrario. Genera KeyError se il set è vuoto
remove()	Rimuove un elemento dal set. Se l'elemento non è un membro, genera un KeyError
union()	Restituisce l'unione dei set in un nuovo set
update()	Aggiorna il set con l'unione di sé stesso e degli altri

3 Operatori

Dopo una panoramica generale sui tipi di dati, vediamo i principali operatori che il linguaggio Python supporta:

- Operatori aritmetici;
- Operatori di confronto (relazionali);
- Operatori di assegnazione;
- Operatori binari;
- Operatori logici;
- Operatori di appartenenza;
- Operatori di identità.

Diamo un'occhiata a questi operatori più da vicino.

Puoi trovare gli esempi nelle tabelle successive agli script
`Operatori_python.py` o `Operatori_python.ipynb`.

3.1 Operatori aritmetici

Ipotizziamo di avere due variabili: x pari a 2 e k uguale a 4. La seguente tabella mostra alcuni esempi di come possono essere utilizzati gli operatori matematici:

Operatore	Descrizione operatore	Esempio
+	Simbolo dell'addizione	$x + k = 6$
-	Simbolo della sottrazione	$x - k = -2$
*	Moltiplicazione	$x * k = 8$
/	Divisione	$x / k = 0.5$
%	Restituisce il resto della divisione (modulo)	$x \% k = 2$ $k \% x = 0$
**	Elevamento a potenza	$x ** k = 16$
//	Restituisce il valore intero della divisione	$x // k = 0$, $k // x = 2$, $-5.0 // 3 = -2.0$

3.2 Operatori di confronto

Questi operatori confrontano i valori su entrambi i lati e decidono la relazione tra loro. Sono anche chiamati **operatori relazionali**.

Supponiamo che la variabile z contenga 5, la variabile y contenga 7, e la variabile r sia pari a 5. Avremo:

Operatore	Descrizione	Esempio
==	Se i valori di due operandi sono uguali, la condizione diventa vera, altrimenti falsa	$r == z \rightarrow$ output True $z == y \rightarrow$ output False
!=	Se i valori di due operandi non sono uguali, allora la condizione diventa vera, viceversa falsa	$r != z \rightarrow$ output False $z != y \rightarrow$ output True
>	Se il valore dell'operando di sinistra è maggiore del valore dell'operando di destra, allora la condizione diventa vera, al contrario falsa.	$r > z \rightarrow$ output False $y > z \rightarrow$ output True
<	Se il valore dell'operando di sinistra è inferiore al valore dell'operando di destra, allora la condizione diventa vera, al contrario falsa. È l'operatore opposto al precedente.	$r < z \rightarrow$ output False $y < z \rightarrow$ output False $r < y \rightarrow$ output True
>=	Se il valore dell'operando di sinistra è maggiore o uguale al valore dell'operando di destra, allora la condizione diventa vera, al contrario falsa	$r >= z \rightarrow$ output True $y >= z \rightarrow$ output True $r >= y \rightarrow$ output False
<=	Se il valore dell'operando di sinistra è inferiore o uguale al valore dell'operando di destra, allora la condizione diventa vera, al contrario falsa	$r <= z \rightarrow$ output True $y <= z \rightarrow$ output False $r <= y \rightarrow$ output True

3.3 Operatori di assegnazione

Gli operatori di assegnazione permettono di assegnare un valore a una variabile. Supponiamo che la variabile g contenga 12 e la variabile h contenga 16, avremo:

Operatore	Descrizione	Esempio
=	Assegna valori dagli operandi del lato destro all'operando del lato sinistro	$f = g + 5 \rightarrow$ Output 17
+=	Aggiunge l'operando di destra all'operando di sinistra e assegna il risultato all'operando di sinistra	$f += 2 \rightarrow$ Output 19 è uguale a $f = f + 2$

-=	Sottrae l'operando di destra dall'operando di sinistra e assegna il risultato all'operando di sinistra	$f -= 9 \rightarrow$ Output 10 è uguale a $f = f - 9$
*=	Moltiplica l'operando di destra con l'operando di sinistra e assegna il risultato all'operando di sinistra	$f *= 2 \rightarrow$ Output 20 è uguale a $f = f * 2$
/=	Divide l'operando di sinistra con l'operando di destra e assegna il risultato all'operando di sinistra	$f /= 4 \rightarrow$ Output 5.0 è uguale a $f = f / 4$
%=	Prende il modulo utilizzando i due operandi e assegna il risultato all'operando di sinistra	$f \% = 2 \rightarrow$ Output 1 (f era pari a 5.0) è uguale a $f = f \% 2$
**=	Esegue il calcolo esponenziale (potenza) sugli operatori e assegna valore all'operando di sinistra	$g ** = 2 \rightarrow$ Output 144 è uguale a $g = g ** 2$
//=	Divide l'operando di sinistra con l'operando di destra e assegna il valore intero della divisione all'operando di sinistra	$h //= 5 \rightarrow$ Output 3 è uguale a $h = h // 5$

3.4 Operatori binari

Esistono poi gli operatori binari (o *bitwise*) che permettono di lavorare al livello dei singoli bit e sono utili in particolari circostanze, come far eseguire dei calcoli ad un computer.

Un numero in binario viene rappresentato solamente con sequenze di 0 o 1, partendo da destra verso sinistra. Per determinare il valore di un numero in questa rappresentazione, si utilizzano le potenze di base 2 che vengono moltiplicate per uno o zero tante volte quanto necessario alla rappresentazione.

$$\dots 2^{11} 2^{10} 2^9 2^8 2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0$$

Ad esempio, ipotizziamo che a sia pari a 20 e b pari a 5.

In termini decimali 20 si può scrivere come $2 * 10^1$

In termini binari 20 è pari a: 10100, ossia:

$$2^0 * 0 + 2^1 * 0 + 2^2 * 1 + 2^3 * 0 + 2^4 * 1 = 4 + 16 = 20$$

L'esponente partendo da 0 indica la posizione del bit nella rappresentazione. 0 è la prima posizione, 1 la seconda e così via.

Allo stesso modo, in termini decimali 5 si può scrivere come $5 * 10^0 = 5 * 1 = 5$

5 invece può essere espresso così: $101 \rightarrow 2^0 * 1 + 2^1 * 0 + 2^2 * 1 = 1 + 4 = 5$

Gli operatori bitwise confrontano bit di numeri e restituiscono un intero. Tra i più diffusi abbiamo:

Operatore	Descrizione	Esempio
& (and)	Imposta ogni bit su 1 se entrambi i bit sono 1, altrimenti lascia 0	$a \& b = 4$ (in binario 0000 0100)
(or)	Imposta ogni bit su 1 se uno dei due è 1, o tutte e 2 sono pari a 1. Lascia 0 solo se entrambi sono uguali a 0	$a b = 21$ (in binario 0001 0101)
^ (XOR)	Imposta ogni bit su 1 se solo uno dei due bit è 1, altrimenti lascia 0	$a \wedge b = 17$ (in binario 0001 0001)
~	Inverte tutti i bit (il valore che restituisce è pari a -n -1, con n il valore seguito dalla tilde)	$\sim a = -21$ (in binario 1110 1011) $\sim b = -6$ (in binario 1010)
<<	Il valore degli operandi di sinistra viene spostato a sinistra dal numero di bit specificato dall'operando di destra	$a << b = 640$ (in binario 0010 1000 0000)
>>	Il valore degli operandi di sinistra viene spostato a destra dal numero di bit specificato dall'operando di destra	$a >> b = 0$ (in binario 0000 0000)

3.5 Operatori logici

Esistono i seguenti operatori logici supportati dal linguaggio Python. Supponiamo che la variabile m sia uguale a True e che la variabile n contenga False:

Operatore	Descrizione	Esempio
AND	Se entrambi gli operandi sono veri, allora la condizione diventa vera	$m \text{ and } n \rightarrow \text{Output False}$
OR	Se uno dei due operandi è diverso da zero, la condizione diventa vera	$m \text{ or } n \rightarrow \text{Output True}$
NOT	utilizzato per invertire lo stato logico del suo operando	$\text{not } n \rightarrow \text{Output True}$ $\text{not } m \rightarrow \text{Output False}$

3.6 Operatori di appartenenza

Tra gli operatori di appartenenza in Python abbiamo in e not in. Vengono utilizzati per verificare se un valore o una variabile vengono trovati in una sequenza (stringa, elenco, tupla, set e dizionario).

In un dizionario possiamo solo verificare la presenza della chiave, non il valore.

Se c ad esempio è pari a 'pluto', abbiamo:

Operatore	Descrizione	Esempio
in	Restituisce vero se la variabile viene trovata nella sequenza, altrimenti False	'l' in c -> Output True 'a' in c -> Output False
not in	Restituisce vero se la variabile non viene trovata nella sequenza, altrimenti False	'f' not in c -> Output True 't' not in c -> Output False

Qui, 'l' è presente nella variabile c ma 'Pluto' non è presente in c (ricorda, Python fa distinzione tra maiuscole e minuscole, è case sensitive).

3.7 Operatori di identità

Gli operatori di identità confrontano le posizioni di memoria di due oggetti. Di seguito sono illustrati due operatori Identity:

Operatore	Descrizione	Esempio
is	Valuta se le variabili su entrambi i lati dell'operatore puntano allo stesso oggetto e in caso positivo restituisce True, False in caso contrario	r is z -> Output True r is y -> Output False
is not	Valuta se le variabili su entrambi i lati dell'operatore puntano ad oggetti differenti e in caso positivo restituisce True, in caso contrario False	r is not z -> Output False r is not y -> Output True

Tieni presente che is è diverso dall'operatore == che abbiamo visto in precedenza. Mentre quest'ultimo operatore confronta i valori di entrambi gli operandi e verifica l'uguaglianza dei valori, l'operatore is controlla se entrambi gli operandi si riferiscono allo stesso oggetto oppure no.

Per vedere l'oggetto di una variabile si può usare la **funzione id()**: ad esempio, ecco il motivo per cui r is z è True (hanno lo stesso oggetto) e r is y è False (hanno oggetti diversi):

```
id(r)
140712300618256

id(z)
140712300618256

id(y)
140712300618320
```

Figura 15: Esempio funzione id()

L'output di questa funzione varia con differenti esecuzioni: molto probabilmente eseguendo lo stesso script per questa funzione troverai valori differenti.

4 Funzioni

In Python, una funzione è un **gruppo di istruzioni correlate che esegue un'attività specifica**.

Le funzioni aiutano a suddividere il programma in blocchi più piccoli e modulari. Man mano che il programma diventa sempre più grande, le funzioni lo rendono più organizzato e gestibile.

Inoltre, evitano la ripetizione e rendono riutilizzabile il codice.

Puoi trovare i seguenti esempi negli script `Funzioni.py` o `Funzioni.ipynb`.

4.1 Sintassi

La sintassi di una funzione è così composta:

```
def nome_funzione(parametri):  
    """docstring"""  
    statement(s)
```

Quindi una funzione è composta da:

1. **Parola chiave def** che segna l'inizio dell'intestazione della funzione.
2. Un nome di funzione per identificare in modo univoco la funzione. La denominazione delle funzioni segue le stesse regole di scrittura degli identificatori in Python.
3. **Parametri** (argomenti) attraverso i quali passiamo i valori a una funzione. Sono opzionali e non obbligatori.
4. I due punti (:) per segnare la fine dell'intestazione della funzione.
5. Stringa di documentazione facoltativa (docstring) per descrivere cosa fa la funzione.
6. Una o più istruzioni Python valide che compongono il corpo della funzione (statement). Le dichiarazioni devono avere lo stesso livello di rientro (di solito 4 spazi).
7. Un'istruzione di restituzione facoltativa (return) per restituire un valore dalla funzione.

Ad esempio:

```
def saluta (nome):  
    """ Questa funzione saluta la persona che viene passata come parametro """  
    stampa ("Ciao, " + nome + ". Buongiorno!")
```

Per richiamare una funzione creata basta scrivere il nome della funzione e tra parentesi passare il parametro impostato alla creazione della funzione. Ad esempio:

`saluta ('Giampiero')` -> Output: Ciao, Giampiero. Buongiorno!

4.1.1 Spiegare il funzionamento di una funzione

La prima stringa dopo l'intestazione della funzione si chiama docstring ed è l'abbreviazione di stringa per la documentazione. Viene brevemente usata per spiegare cosa fa una funzione. Sebbene facoltativa, la documentazione è una buona pratica di programmazione.

È consigliato documentare il codice di una funzione, nei casi in cui verrà utilizzata anche da altri utenti, oppure se dovrai riutilizzarla in futuro.

Nell'esempio sopra, abbiamo una spiegazione immediatamente sotto l'intestazione della funzione. Generalmente utilizziamo le virgolette triple in modo che il docstring possa estendersi fino a più righe. Questa stringa è disponibile per noi come attributo `__doc__` della funzione.

`print(saluta.__doc__)`

Output -> Questa funzione saluta la persona che viene passata come parametro

4.1.2 Return Statement

L'istruzione **return** viene utilizzata per uscire da una funzione e tornare al punto in cui è stata chiamata.

Questa istruzione può contenere un'espressione che viene valutata e il valore viene restituito. Se non è presente alcuna espressione nell'istruzione o l'istruzione return stessa non è presente all'interno di una funzione, la funzione restituirà l'oggetto None (nessuno).

`print(saluta("Giacomo"))` -> Output: Ciao Giacomo. Buongiorno! None

Qui, None è il valore restituito poiché `saluta()` stampa direttamente il nome e non viene utilizzata alcuna istruzione return.

Esempio Return statement:

`def esponenziale(num, exp):`

```
    """Questa funzione restituisce l'esponenziale del numero passato con la potenza definita dalla variabile exp"""
    return num**exp
```

`print(esponenziale(2,4))` -> Risultato 16

`print(esponenziale(-5,3))` -> Risultato -125

In Python possiamo avere due tipi di funzioni definite:

- **Built-in**: sono quelle già presenti ed esistenti in Python;
- **User-defined**: sono quelle definite dall'utente.

4.2 Built-in

Vediamo le principali funzioni di questo tipo che sono presenti in Python:

abs(): restituisce il valore assoluto di un numero

all(): restituisce True quando tutti gli elementi del parametro iterable (lista, tupla, dizionario) sono True

any(): verifica se qualsiasi elemento di un Iterable è True

ascii(): restituisce una stringa contenente una rappresentazione stampabile

bin(): converte il numero intero in stringa binaria

bool(): converte un valore in booleano

bytearray(): restituisce un array con una dimensione in byte specificata

callable(): verifica se l'oggetto è richiamabile

chr(): restituisce un carattere (una stringa) da un numero intero

classmethod(): restituisce il metodo di classe per una determinata funzione

compile(): restituisce un oggetto codice Python

complex(): crea un numero complesso

delattr(): elimina l'attributo dall'oggetto

dict(): crea un dizionario

dir(): cerca di restituire gli attributi dell'oggetto

divmod(): restituisce una tupla con quoziente di due numeri e resto

enumerate(): restituisce un oggetto enumerato

eval(): esegue il codice Python all'interno del programma

exec(): esegue il programma creato dinamicamente

filter(): costruisce un iteratore da elementi che sono veri

float(): restituisce un numero in virgola mobile

format(): restituisce una rappresentazione formattata di un valore

frozenset(): restituisce un oggetto frozenset immutabile, ossia la versione di un oggetto set non modificabile

getattr(): restituisce il valore dell'attributo denominato di un oggetto

globals(): restituisce il dizionario della tabella corrente dei simboli globale. Una tabella dei simboli è una struttura dati gestita da un compilatore che contiene tutte le informazioni necessarie sul programma.

hasattr(): restituisce se l'oggetto ha un attributo denominato

hash(): restituisce il valore hash di un oggetto

help(): richiama il sistema di aiuto integrato

hex(): converte un intero in esadecimale

id(): restituisce l'identificazione di un oggetto

input(): legge e restituisce una riga di stringa

int(): restituisce un numero intero da un numero o una stringa

isinstance(): verifica se un oggetto è un'istanza di classe

issubclass(): verifica se un oggetto è una sottoclasse di una classe

iter(): restituisce un iteratore

list(): crea un elenco/lista in Python

len(): restituisce la lunghezza di un oggetto

locals(): restituisce il dizionario di una tabella corrente dei simboli locale. Questa tabella dei simboli memorizza tutte le informazioni necessarie all'ambito locale del programma.

map(): applica la funzione e restituisce un elenco

max(): restituisce l'articolo più grande

memoryview(): restituisce la vista in memoria di un argomento

min(): restituisce il valore più piccolo

next(): recupera l'elemento successivo dall'iteratore

object(): crea un oggetto privo di caratteristiche

oct(): restituisce la rappresentazione ottale di un numero intero

open(): restituisce un oggetto file

ord(): restituisce un numero intero che rappresenta il carattere Unicode passato come parametro

pow(): restituisce la potenza di un numero

print(): stampa l'oggetto indicato

property(): restituisce l'attributo della proprietà

range(): restituisce la sequenza di numeri interi tra start e stop

repr(): restituisce una rappresentazione stampabile dell'oggetto

reversed(): restituisce l'iteratore inverso di una sequenza

round(): arrotonda un numero ai decimali specificati

set(): costruisce e restituisce un set

setattr(): imposta il valore di un attributo di un oggetto

slice(): restituisce un oggetto "slice" che può essere utilizzato per tagliare stringhe, elenchi, tuple, ecc.

sorted(): restituisce un elenco ordinato dall'iterabile indicato

staticmethod(): trasforma un metodo in un metodo statico

str(): restituisce la versione stringa dell'oggetto

sum(): aggiunge elementi di Iterable (lista, tupla, dizionario, ecc.) e restituisce la somma.

super(): Restituisce un oggetto proxy della classe base

tuple(): restituisce una tupla

type(): restituisce il tipo di oggetto

vars(): restituisce l'attributo `__dict__`

zip(): restituisce un iteratore di tuple

4.3 User-defined

Le funzioni che definiamo per svolgere determinate attività specifiche vengono definite funzioni definite dall'utente (**user-defined**).

Tutte le altre funzioni che scriviamo da sole, che non sono già state scritte da altri, rientrano nelle funzioni definite dall'utente. Quindi, la nostra funzione definita dall'utente potrebbe essere riutilizzabile per qualcun altro.

Questo tipo di funzioni portano ad una serie di vantaggi:

- Le funzioni definite dall'utente aiutano a scomporre un programma di grandi dimensioni in piccoli segmenti, facilitando la comprensione, la manutenzione e il debug del programma.
- Se si verifica un codice ripetuto in un programma. La funzione può essere utilizzata per includere quei codici ed eseguire quando necessario chiamando tale funzione.
- I programmatori che lavorano su progetti di grandi dimensioni possono dividere il carico di lavoro creando diverse funzioni.

Gli esempi mostrati nel paragrafo 4.1 delle funzioni, sono semplici esempi di funzioni definite dall'utente.

4.4 Funzioni anonime o lambda

In Python, una **funzione anonima** è una funzione definita senza nome. Mentre le normali funzioni sono definite usando la parola chiave `def`, le funzioni anonime sono definite usando la parola chiave **lambda**. Quindi, le funzioni anonime sono anche chiamate funzioni lambda.

Sintassi dell'istruzione:

lambda argomento: espressione

Le funzioni Lambda possono avere un numero qualsiasi di argomenti ma solo un'espressione. L'espressione viene valutata e restituita. Le funzioni Lambda possono essere utilizzate ovunque siano richiesti oggetti funzione.

Un esempio molto semplice di una funzione lambda può essere così definito:

```
triplo = lambda x: x * 3
```

```
print (triplo(16))
```

Risultato: 48

In pratica con la precedente funzione viene stampato il triplo di un numero (il triplo di 16, nell'esempio).

Nel programma sopra, `lambda x: x * 3` è la funzione lambda. Qui `x` è l'argomento e `x * 3` è l'espressione che viene valutata e restituita.

Questa funzione non ha nome. Restituisce un oggetto funzione che è il triplo di quello definito.

Solitamente, le funzioni lambda vengono utilizzate quando richiediamo una funzione senza nome per un breve periodo di tempo.

In Python, generalmente viene utilizzata come argomento per una funzione di ordine superiore (una funzione che accetta altre funzioni come argomenti). Le funzioni Lambda vengono utilizzate insieme a funzioni integrate come `filter()`, `map()` ecc.

Vediamo due esempi per maggiore comprensione.

4.4.1 Funzione lambda con funzione filter

La **funzione filter()** in Python accetta una funzione e un elenco come argomenti.

La funzione viene chiamata con tutti gli elementi nell'elenco e viene restituito un nuovo elenco che contiene elementi per i quali la funzione restituisce True. Ecco un esempio di utilizzo della funzione filter() per filtrare solo i numeri dispari da un elenco.

```
lista = [12,75,31,24,17,8,6,93,100]
```

```
lista_dispari = list(filter(lambda x: (x%2 == 1) , lista))
```

```
print(lista_dispari)
```

Risultato: [75, 31, 17, 93]

4.4.2 Funzione lambda con funzione map

La **funzione map()** in Python accetta una funzione e una lista.

La funzione viene chiamata con tutti gli elementi nella lista e ne viene restituita una nuova che contiene gli elementi restituiti da quella funzione per ciascun elemento.

Ecco un esempio di utilizzo della funzione map() per triplicare tutti gli elementi in un elenco.

```
lista2 = [11,4,6,17,13,22,7]
```

```
lista_triplicata = list(map(lambda x: x * 3 , lista2))
```

```
print(lista_triplicata)
```

Risultato -> [33, 12, 18, 51, 39, 66, 21]

4.5 Moduli

I moduli si riferiscono a un file contenente istruzioni e definizioni Python.

Un file contenente codice Python, ad esempio Modulotest.py, è chiamato modulo.

Puoi trovare i seguenti esempi negli script Modulotest.py o Modulotest.ipynb.

Utilizziamo i moduli per suddividere i programmi di grandi dimensioni in piccoli file gestibili e organizzati. Inoltre, i moduli forniscono riusabilità del codice.

Possiamo definire le nostre funzioni più utilizzate in un modulo e importarlo, invece di copiarne le definizioni in diversi programmi.

Creiamo un modulo. Digita quanto segue e salvalo come Modulotest.py.

```
def prod(x,y):
```

```
risultato = x*y  
  
return (risultato)
```

Qui, abbiamo definito una **funzione prod()** all'interno di un modulo chiamato Modulotest. La funzione accetta due numeri e restituisce il loro prodotto.

4.5.1 Importare un modulo

Per importare un modulo all'interno di un altro modulo o nell'interprete Python, occorre utilizzare il comando `import` seguito dal nome del modulo che si vuole importare:

```
import Modulotest
```

Attenzione ai caratteri, perché Python è case sensitive, ossia i caratteri in maiuscolo o in minuscolo fanno la differenza. Pertanto Modulotest non è uguale a modulotest.

Per utilizzare la funzione `prod()` poi è possibile dichiarare `nome modulo.funzione`. Nel nostro esempio avremo:

```
a= Modulotest.prod(5,24)  
  
print (a)  
  
#Risultato: 120
```

In questo modo eseguiamo il prodotto tra 5 e 24, ottenendo 120 come risultato.

4.5.2 Percorso di ricerca del modulo Python

Se otteniamo errore dall'importazione di un modulo molto probabilmente è perché il file dove è salvato il modulo non è incluso all'interno del percorso di ricerca del modulo Python (`sys`).

Digitando il comando **`import sys`** si vedono tutti i percorsi in cui è possibile salvare i moduli. Se il file non è salvato in nessun percorso otteniamo un errore in fase di importazione.

ModuleNotFoundError: No module named 'Modulotest'

```
import sys  
  
sys.path
```

Risultato:

```
['C:\\Users\\Lorenzo',  
  
'C:\\Users\\Lorenzo\\Anaconda3\\Python37.zip',
```

```
'C:\\Users\\Lorenzo\\Anaconda3\\DLLs',  
'C:\\Users\\Lorenzo\\Anaconda3\\lib',  
'C:\\Users\\Lorenzo\\Anaconda3',  
,  
'C:\\Users\\Lorenzo\\Anaconda3\\lib\\site-packages',  
'C:\\Users\\Lorenzo\\Anaconda3\\lib\\site-packages\\win32',  
'C:\\Users\\Lorenzo\\Anaconda3\\lib\\site-packages\\win32\\lib',  
'C:\\Users\\Lorenzo\\Anaconda3\\lib\\site-packages\\Pythonwin',  
'C:\\Users\\Lorenzo\\Anaconda3\\lib\\site-packages\\IPython\\extensions',  
'C:\\Users\\Lorenzo\\.iPython',  
'C:\\Users\\Lorenzo\\Desktop\\Python\\File Python']
```

Per aggiungere un percorso a sys, basta digitare il comando **sys.path.append('indicare percorso di salvataggio del modulo da importare')**. È importante notare che il percorso è separato da due barre laterali \\.

Allo stesso modo se si vuole rimuovere un percorso, il comando da utilizzare è:

sys.path.remove('percorso da rimuovere')

4.5.3 Importare un modulo e rinominarlo

Un'altra modalità per importare un modulo è quella che permette di associandogli un cosiddetto alias, in modo da richiamare quest'ultimo anziché il modulo per intero. Ad esempio:

```
import math as m
```

```
print("Il valore di e è pari a", m.e)
```

Risultato: il valore di e è pari a 2.718281828459045

Si noti che il nome math non è riconosciuto nel nostro ambito. Quindi, math.e non è valido e solo m.e è l'implementazione corretta.

4.5.4 Importare un modulo con la clausola from

Possiamo importare nomi specifici da un modulo senza importare il modulo nel suo insieme. Ecco un esempio:

```
from math import sqrt, floor
```

```
print(sqrt(4))
```

Output -> 2.0

```
print(floor(3.4))
```

Output -> 3

4.5.5 Reimportare un modulo

Per reimportare un modulo si può utilizzare la **funzione reload()**, all'interno del **modulo imp** per ricaricare un modulo.

Questa funzione è utile soprattutto dallo Shell Python in cui è possibile importare solamente un modulo alla volta.

Può quindi succedere che una volta importato il modulo, esso venga modificato. Anziché chiudere e riaprire lo shell è possibile digitare:

```
Import imp
```

```
Import Modulotest # primo import
```

Se andassi a modificare Modulotest avendo aperto lo shell, aggiungendo ad esempio print ("Ciao"):

```
def prod(x,y):
```

```
    risultato = x*y
```

```
    return (risultato)
```

```
print ('ciao')
```

Ritornando nello shell di Python posso importare nuovamente il modulo Modulotest modificato:

```
imp.reload(Modulotest) # secondo import
```

Output -> Ciao

```
# <module 'Modulotest' from '.\\Modulotest.py'>
```

4.6 Pacchetti

Di solito non archiviamo tutti i nostri file sul nostro computer nella stessa posizione. Usiamo una gerarchia di directory ben organizzata per un accesso più facile.

File simili sono conservati nella stessa directory, ad esempio, potremmo tenere tutti i giochi nella directory "videogame". Analogamente a questo, Python ha pacchetti per directory e moduli per file.

Man mano che il programma applicativo aumenta di dimensioni con molti moduli, inseriamo moduli simili in un pacchetto e moduli diversi in pacchetti diversi. Ciò rende un programma di una certa dimensione facile da gestire e concettualmente chiaro.

Allo stesso modo, poiché una directory può contenere sottodirectory e file, un pacchetto Python può avere sottopacchetti e moduli.

Una directory deve contenere un file chiamato `__init__.py` affinché Python lo consideri come un pacchetto. Questo file può essere lasciato vuoto ma generalmente inseriamo il codice di inizializzazione per quel pacchetto in questo file.

Quindi mentre un modulo è un singolo file Python, un pacchetto è una **directory di moduli Python** che contiene un file `__init__.py` aggiuntivo, per distinguere un pacchetto da una directory che contiene solo un gruppo di Python script.

Per importare un pacchetto occorre indicare il nome del pacchetto seguito dal percorso dei sottopacchetti da importare. Ipotizzando di avere il pacchetto "videogame" al percorso `videogame\livello\inizio`, si può importare il pacchetto nel seguente modo:

```
import videogame.livello.inizio
```

Durante l'importazione dei pacchetti, Python appare nell'elenco delle directory definite in `sys.path`, in modo simile al percorso di ricerca del modulo. Se non lo trova darà errore.

5 Condizione if - else statement

Quando vogliamo far seguire un certo tipo di percorso al programma piuttosto che un altro si possono utilizzare gli **if -else statements**.

Questa dicitura è adatta in quelle casistiche in cui si voglia eseguire del codice solo se una determinata condizione è soddisfatta.

Sintassi dell'istruzione

if espressione di test:

statement (s)

Qui, il programma valuta l'espressione del test ed eseguirà le istruzioni solo se l'espressione del testo è True, ossia vera.

Se l'espressione di testo è falsa (False), le istruzioni non vengono eseguite. In Python, il corpo dell'istruzione if è indicato dal rientro, mentre la prima riga non indentata segna la fine.

Di default, Python interpreta i valori diversi da zero come Vero. None e 0 vengono interpretati come False.

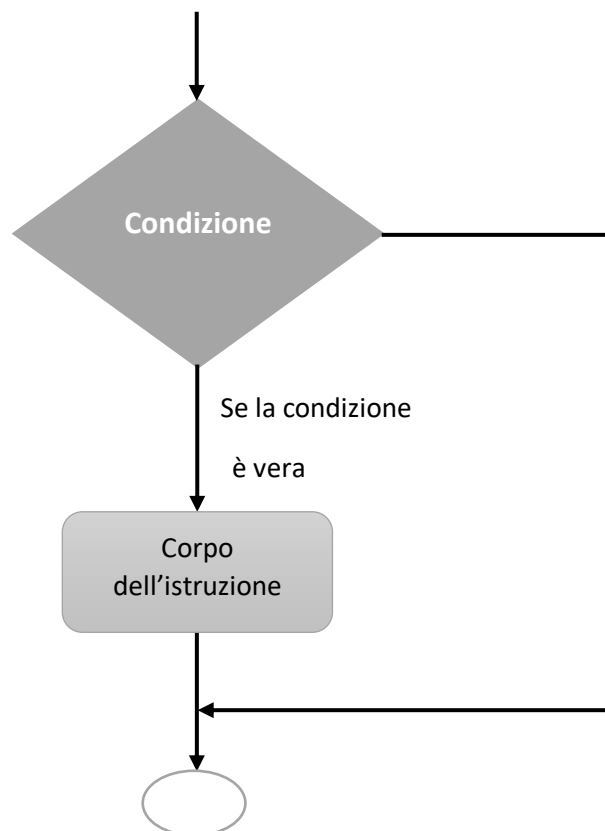


Figura 16: Flusso if Statement

Puoi trovare i seguenti esempi negli script `If-else_statement.py`
o `If-else_statement.ipynb`.

Esempio:

```
numero = 5
```

```
if numero > 0:
```

```
    print ("il numero positivo è pari a: " + str(numero))
```

Risultato: il numero positivo è pari a: 5

Questa condizione, essendo vera, fa sì che il codice del comando `print` venga eseguito. E come risultato otteniamo il valore della variabile `numero`.

```
numero1 = -4
```

```
if numero1 > 0:
```

```
    print ("il numero positivo è pari a: " + str(numero1))
```

La condizione precedente invece non è vera, pertanto non verrà visualizzato nessun risultato. Infatti qualora l'espressione di test risulti falsa, le istruzioni all'interno del corpo `if` vengono ignorate.

È possibile però anche far eseguire al programma istruzioni quando la condizione risulta falsa. Per farlo dobbiamo aggiungere il comando `else`.

Continuando con l'esempio precedente possiamo vedere che, se aggiungiamo l'istruzione `else` seguente, viene stampato il valore del numero negativo.

```
numero1 = -4
```

```
if numero1 > 0:
```

```
    print ("il numero positivo è pari a: " + str(numero1))
```

```
else:
```

```
    print ("il numero negativo è pari a: " + str(numero1))
```

Risultato: il numero negativo è pari a: -4

Ovviamente se la variabile `numero1` fosse uguale a un numero positivo verrebbe stampata la prima istruzione (contenente nell'`if` statement).

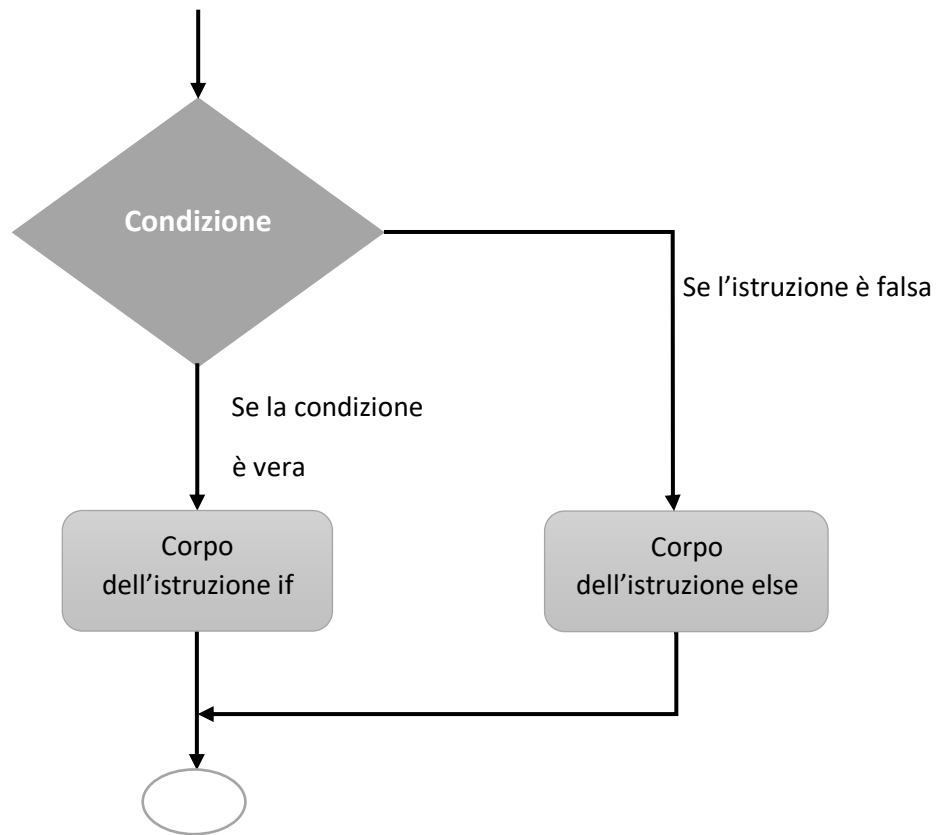


Figura 17: Flusso `if-else` statement

Si possono successivamente annidare condizioni tra di loro, utilizzando il **comando `elif`**. Ad esempio se volessi associare alla variabile `numero1` il valore di 0, per non farlo risultare nei valori negativi potrei aggiungere la successiva condizione:

```
numero1 = 0
```

```
if numero1 > 0:
```

```
    print ("il numero positivo è pari a: " + str(numero1))
```

```
elif numero1 == 0:
```

```
    print ("il numero è pari a: " + str(numero1))
```

```
else:
```

```
    print ("il numero negativo è pari a: " + str(numero1))
```

Risultato: il numero è pari a: 0

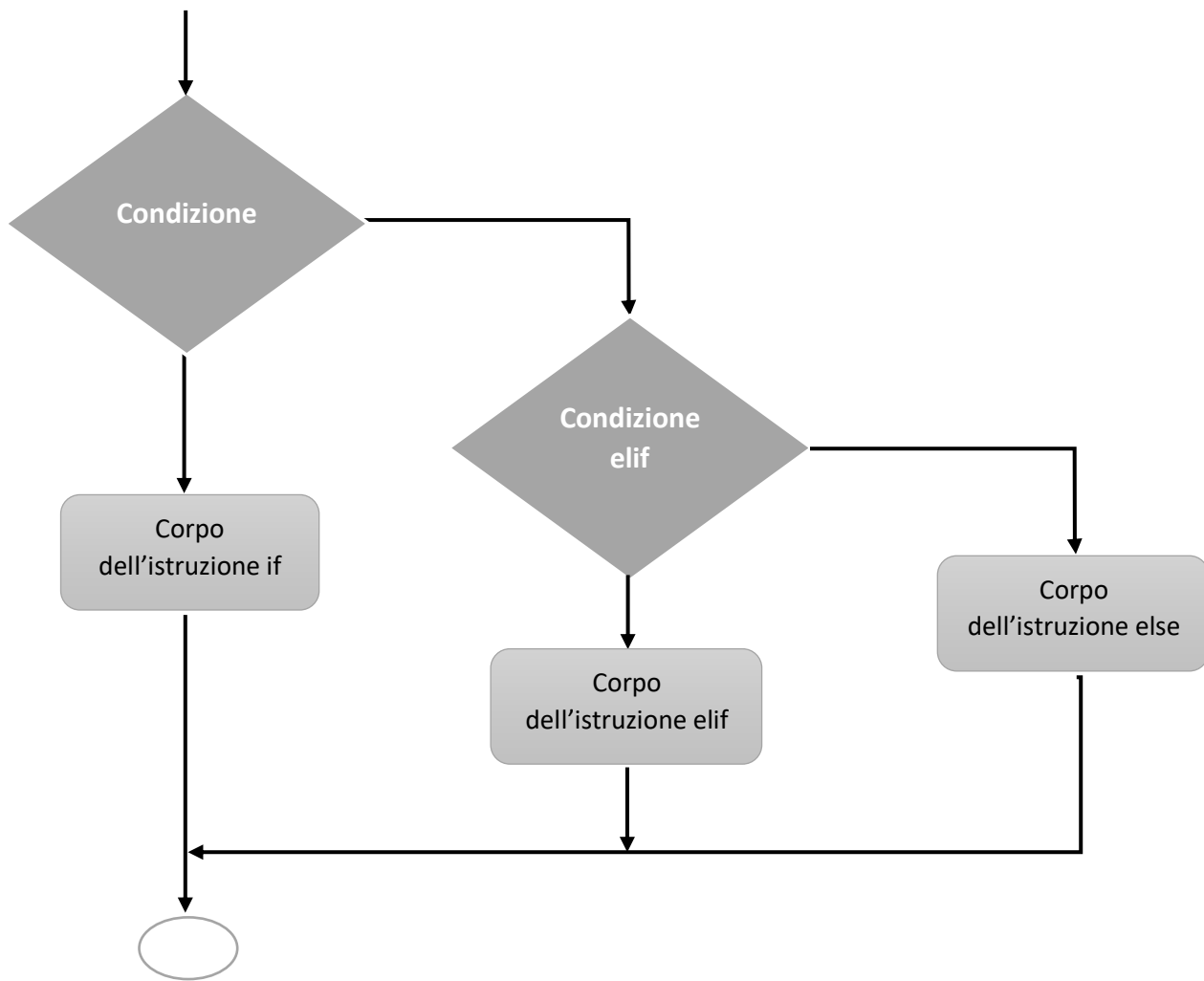


Figura 18: Flusso If-elif-else statement

Infine, possiamo avere un'istruzione if ... elif ... else all'interno di un'altra istruzione if ... elif ... else. Questa si chiama **condizione annidata if-else**.

Qualsiasi numero di queste affermazioni può essere nidificato l'una nell'altra e il rientro è l'unico modo per capire il livello di annidamento. Possono diventare confusi, quindi devono essere evitati se non necessario.

Un esempio di ciclo annidato può essere il seguente:

```
numero1 = 4
if type(numero1) == int:
    if numero1 > 0:
        print ("il numero è un intero positivo, pari a: " + str(numero1))
    elif numero1 < 0:
```

```
    print ("il numero relativo è pari a: " + str(numero1))
else:
    print ("il numero è pari a: " + str(numero1))
else:
    if type(numero1) == float:
        print ("il numero è reale e pari a: " + str(numero1))
```

Risultato: il numero è un intero positivo, pari a: 4

6 Loops

Il ciclo, detto anche loop, è un'istruzione che si ripete fino al raggiungimento di una condizione specificata.

In Python possiamo avere:

- Ciclo for;
- Ciclo while.

Vediamoli brevemente.

6.1 Ciclo For

Il ciclo for viene usato per scorrere su una sequenza (elenco, tupla, stringa) o altri oggetti iterabili.

La sintassi del ciclo è la seguente:

for val in sequence:

Corpo del for

Qui, val è la variabile che accetta il valore dell'elemento all'interno della sequenza su ogni iterazione. Il ciclo continua fino a quando non raggiungiamo l'ultimo elemento della sequenza.

Il corpo del ciclo for è separato dal resto del codice usando il rientro.

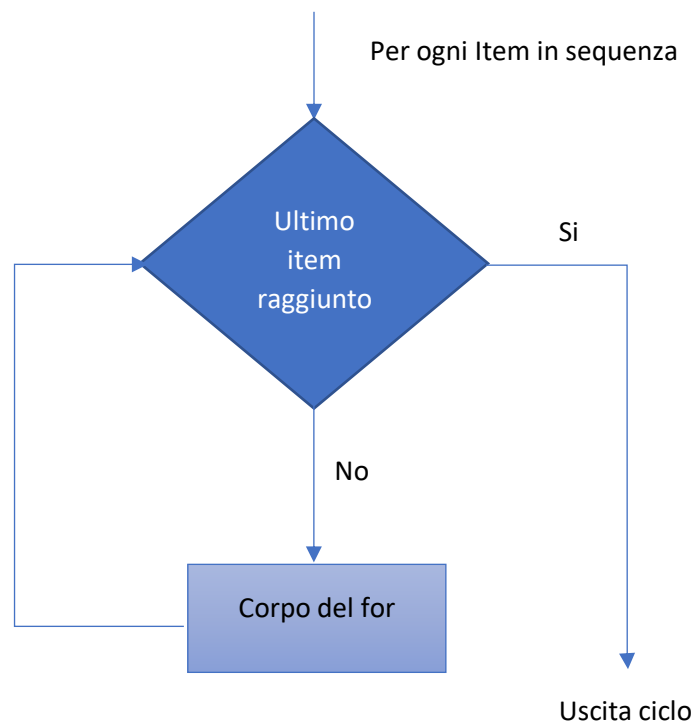


Figura 19: Diagramma di flusso ciclo for

Puoi trovare i seguenti esempi negli script `Loops.py` o `Loops.ipynb`.

```
x = [2,3,4,5]
prod = 1
for val in x:
    x = prod
    prod = prod * val
    print ("Moltiplicando "+str(val)+ " per " +str(x) +" otteniamo " +str(prod))
```

Come risultato otteniamo:

Moltiplicando 2 per 1 otteniamo 2

Moltiplicando 3 per 2 otteniamo 6

Moltiplicando 4 per 6 otteniamo 24

Moltiplicando 5 per 24 otteniamo 120

6.1.1 La funzione range ()

Possiamo generare una sequenza di numeri usando la **funzione range()**. Ad esempio, `range(5)` genererà numeri da 0 a 4 (5 numeri). Viene usata insieme alla funzione `list` per stampare questi numeri.

```
list(range(5))
```

Risultato: [0, 1, 2, 3, 4]

Possiamo anche definire l'inizio, l'arresto e la dimensione del passo come intervallo (`start`, `stop`, `step_size`). Quest'ultimo parametro ha come valore predefinito 1, se non fornito.

```
list(range(7,15))
```

Risultato: [7, 8, 9, 10, 11, 12, 13, 14]

```
List(range(7,15,2))
```

Risultato: [7, 9, 11, 13]

La funzione **range()** può essere utilizzata nei loop per iterare una sequenza di numeri. In aggiunta è possibile combinarla con la **funzione len()** per scorrere una sequenza usando l'indicizzazione. Ecco un esempio:

```
item = ['banana', 'mela', 'arancia']  
  
for val in range(len(item)):  
    print ('Vorrei mangiare una',item[val])
```

Risultato:

Vorrei mangiare una banana

Vorrei mangiare una mela

Vorrei mangiare una arancia

6.1.2 Loop con else

È possibile utilizzare il loop con l'else se la condizione del loop non è verificata. Ad esempio:

```
num = 1  
  
for val in range(num,25,3):  
    print(val)  
  
else:  
    print('Nessun elemento rimanente')
```

Risultato:

1

4

7

10

13

16

19

22

Nessun elemento rimanente

Oppure è possibile utilizzare la funzione **break**, qualora ci si trovi all'interno di una condizione **if**, all'interno di un ciclo, per uscire e non rimanere bloccati.

```

età = 35
dict1 = {25: 'ragazzo', 42: 'donna', 7: 'bimbo', 57: 'signore'}
for val in dict1:
    if val == età:
        print (dict1[età])
        break
else:
    print("Età della persona non presente nel dizionario")

```

Risultato -> Età della persona non presente nel dizionario

6.2 Ciclo While

Oltre al ciclo for, possiamo avere il ciclo while. Questo ciclo in Python viene utilizzato per scorrere su un blocco di codice purché l'espressione di test (condizione) sia vera e nei casi in cui non si conosca il numero di volte per iterare in anticipo.

La sintassi viene così definita:

while espressione di test:

Corpo del while

Nel ciclo while, l'espressione di test viene dapprima controllata.

Il corpo del loop viene inserito solo se l'espressione di test restituisce un risultato True. Dopo una iterazione, e ad ognuna di esse, l'espressione di test viene ricontrollata. Questo processo continua fino a quando tale valore viene valutato come False.

Anche in questo ciclo, il corpo viene determinato tramite rientro e la prima riga non indentata segna la fine.

Python interpreta qualsiasi valore diverso da zero come Vero. None e 0 vengono interpretati come False.

Esempio ciclo while:

```

a = (5,7,98,24,47)
i = 0
somma = 0
while i < 5:
    print(a[i])
    somma = somma + a[i]

```

```
i = i + 1
```

```
print ('la somma della tupla è pari a: '+str(somma))
```

Risultato:

5

7

98

24

47

la somma della tupla è pari a: 181

Le precedenti righe di codice stampano ad ogni iterazione l'elemento della tupla corrispondente. E infine una volta usciti dal ciclo visualizziamo la somma dei numeri all'interno della tupla.

L'espressione di test risulta True fintanto che la variabile contatore *i* è minore o uguale a *n* (5 nell'esempio).

È fondamentale aumentare il valore della variabile contatore nel corpo del loop (*i*). Questo è molto importante (e per lo più dimenticato). In caso contrario, si otterrà un ciclo infinito.

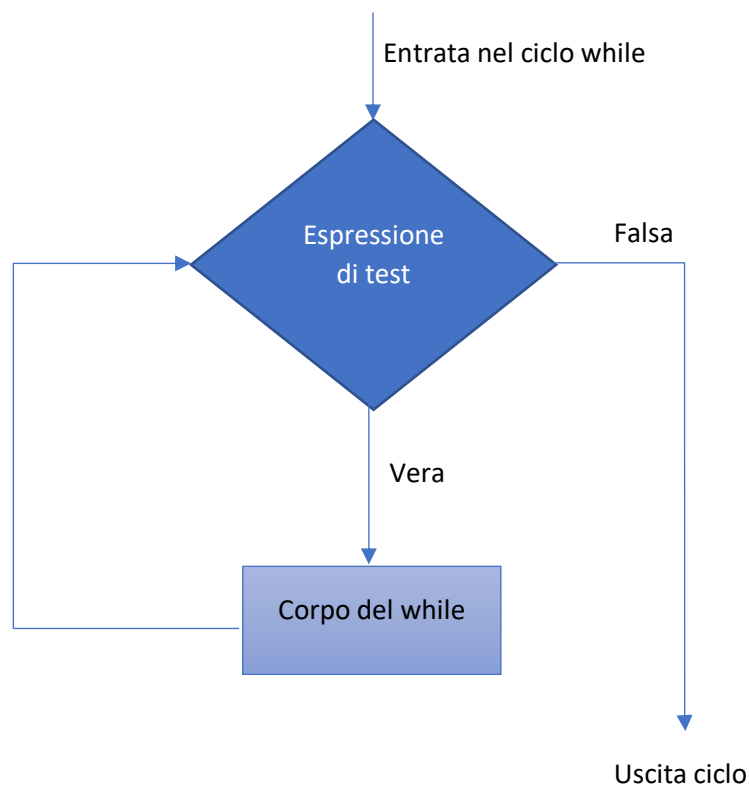


Figura 20: Diagramma di flusso ciclo while

6.2.1 Ciclo while con else

Come per i cicli for, anche i cicli while possono avere un **blocco else opzionale**, che viene eseguito se la condizione nel ciclo restituisce False.

In aggiunta, il ciclo while può essere terminato con un'istruzione break. In tali casi, la parte else viene ignorata. Quindi, la parte else viene eseguita se non si verifica alcuna interruzione e non appena la condizione diventa falsa.

Ad esempio la seguente istruzione permette di valutare se entro 8 iterazioni riusciamo ad ottenere un valore pari a guess, numero random che va da 1 a 8. Utilizziamo la **funzione random.randint()** (dopo aver importato random) per calcolare tale variabile ad ogni iterazione.

```
import random

count = 1

while count < 8:

    guess = random.randint(1,8)

    if count >= 1:

        count +=1

        if guess == count:

            print ("il numero random e count sono pari a "+str(guess))

            print ("Hai vinto!")

            break

        elif count == 8:

            print ("Hai perso")

            print ("dopo 8 iterazioni count è diverso da guess")

else:

    print ("Prova ancora")
```

Se non viene trovato il valore pari a quello random entro le 8 iterazioni, si verifica la condizione else del ciclo while:

Hai perso

dopo 8 iterazioni count è diverso da guess

Prova ancora

6.3 Istruzioni Break e Continue

In Python, le istruzioni break e continue possono alterare il flusso di un normale ciclo. I cicli ripetono su un blocco di codice fino a quando l'espressione di test è falsa, ma a volte desideriamo terminare l'iterazione corrente o persino l'intero ciclo senza controllare l'espressione di test. Le istruzioni break e continue sono utilizzate in questi casi.

6.3.1 Break

L'istruzione break, anticipata poco fa, termina il ciclo che lo contiene. Il controllo del programma scorre all'istruzione immediatamente dopo il corpo del loop.

Se l'istruzione break si trova all'interno di un loop nidificato (loop all'interno di un altro loop), l'istruzione break interromperà il loop più interno.

Questa istruzione è utilizzabile all'interno di entrambi i cicli for e while. Utilizzando l'esempio visto per il ciclo for, possiamo uscire dal ciclo una volta che arriviamo ad un determinato valore (20):

```
x = [2,3,4,5]
prod = 1
for val in x:
    x = prod
    prod = prod * val
    if prod > 20:
        print ('X è pari a '+str(x) +' mentre prod è uguale a '+str(prod))
        break
```

#Risultato -> X è pari a 6 mentre prod è uguale a 24

Per il ciclo while ipotizzando di voler restituire la somma alla quarta iterazione:

```
a = (5,7,98,24,47)
i = 0
somma = 0
while i < 5:
    somma = somma + a[i]
    i = i + 1
    if i == 4:
```

```
print ('Raggiunta la quarta iterazione del ciclo!'+  
      ' La somma della tupla è pari a: '+str(somma))  
  
Break
```

Risultato: Raggiunta la quarta iterazione del ciclo! La somma della tupla è pari a: 134

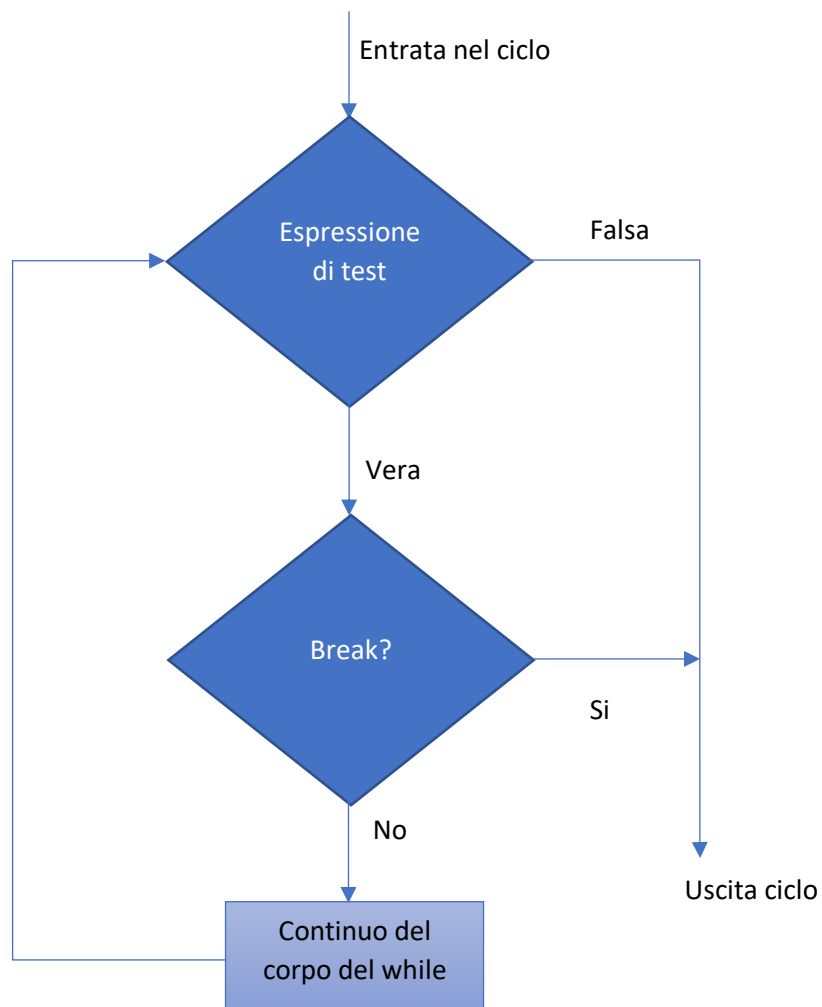


Figura 21: Flusso istruzione break

6.3.2 Continue

L'istruzione continue viene utilizzata per saltare il resto del codice all'interno di un ciclo solo per l'iterazione corrente. Il ciclo non termina ma continua con l'iterazione successiva.

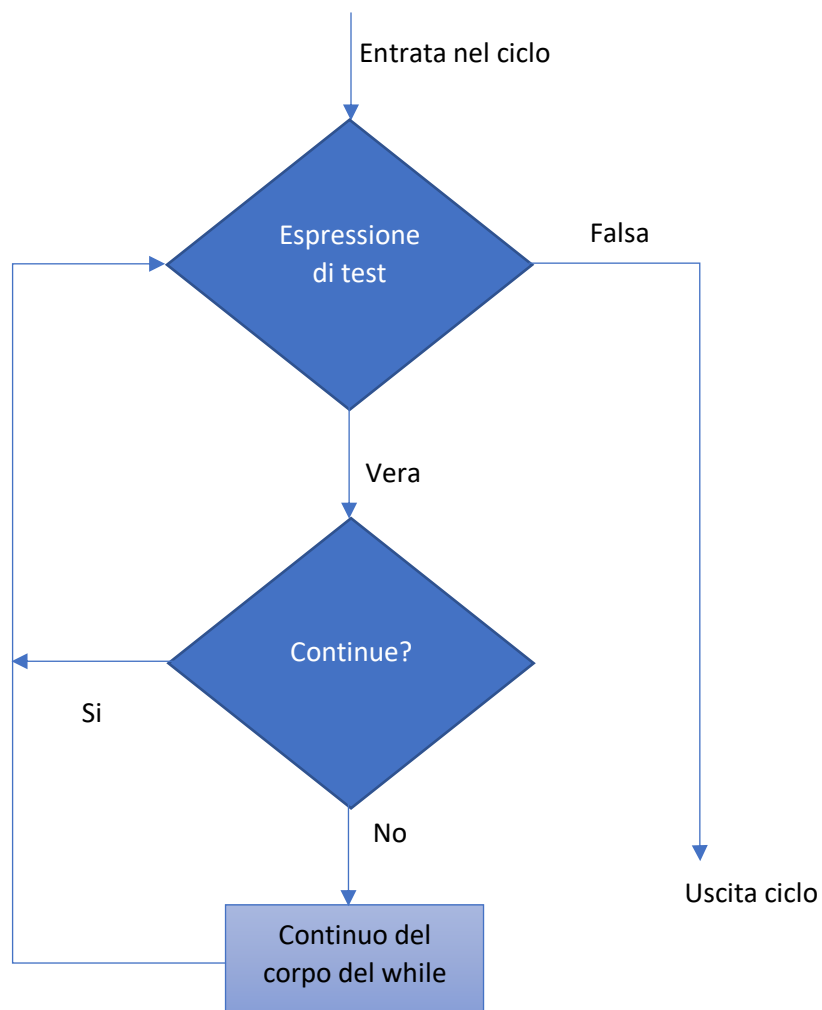


Figura 22: Flusso istruzione continue

Un esempio di utilizzo di istruzione continue per il ciclo for può essere la seguente:

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

```
for i in vals:
```

```
    if i == 2:
```

```
        continue
```

```
    print(i**2)
```

Risultato: 1, 9, 16, 25

In pratica quando si avvera l'istruzione if all'interno del ciclo e si richiama continue, si procede oltre senza fare nulla, ed è il motivo per cui non viene stampato 4 come valore.

Per il ciclo while il risultato non cambia:

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

```
i = 1
```

```
while i <= 5:
```

```
    if i == 2:
```

```
        i=i+1
```

```
        continue
```

```
    print(i**2)
```

```
    i += 1
```

Risultato: 1,9,16,25

La cosa da fare attenzione nel ciclo while è di inserire prima del *continue*, l'incremento del contatore, altrimenti si rientrerà in un loop infinito.

7 Oggetti e classi

Python è un linguaggio di programmazione multi-paradigma, che supporta e si avvale della **programmazione orientata agli oggetti** (in inglese Object Oriented Programming, abbreviato a OOP). Essa rappresenta un modello che organizza la progettazione di software attorno a dati e a oggetti, piuttosto che a funzioni e a logica.

Un oggetto può essere definito come una collezione di dati con attributi e comportamenti univoci.

Per comprendere meglio questo concetto pensiamo a un cane: per Python rappresenta un oggetto.

Il suo nome, la sua età, e il colore del pelo sono attributi, mentre abbaiare, latrare sono due comportamenti dell'oggetto cane.

L'OOP in Python si concentra sulla creazione di codice riutilizzabile, anche noto come DRY (Don't Repeat Yourself), e si basa sulle seguenti proprietà:

- **Ereditarietà:** Capita che gli oggetti siano spesso molto simili. Se creo due oggetti cane, ad esempio, ognuno di essi può avere attributi e comportamenti in comune. Condividono di fatto una logica comune, anche se non sono del tutto uguali.

Quindi, come riutilizzare la logica comune tra le due classi ed estrarre la logica unica in una classe separata? Un modo per raggiungere questo obiettivo è l'ereditarietà.

Significa che crei una classe (figlio) derivando da un'altra classe (genitore). In questo modo, formiamo una gerarchia. Magari nel nostro esempio possiamo creare una classe genitore Cane, che comprenderà all'interno le classi figlio Labrador, Pastore Tedesco, Siberian Husky, ecc. rientranti nella classe genitore.

La classe figlio riutilizza tutti i campi e i metodi della classe genitore (parte comune) e può implementare la propria (parte unica).

L'ereditarietà altro non è che un processo di utilizzo dei dettagli di una nuova classe senza modificare la classe esistente.

- **Incapsulamento:** Supponiamo di avere un programma, che ha alcuni oggetti logicamente diversi che comunicano tra loro, secondo delle regole definite.

L'incapsulamento si ottiene quando ogni oggetto mantiene il suo stato privato, all'interno di una classe. Altri oggetti non hanno accesso diretto a questo stato. Al contrario, possono solo chiamare un elenco di funzioni pubbliche, chiamate metodi.

Quindi, l'oggetto gestisce il proprio stato tramite metodi, e nessun'altra classe può toccarlo se non esplicitamente consentito. Se si desidera comunicare con l'oggetto, è necessario utilizzare i metodi forniti. Ma (per impostazione predefinita), non puoi cambiare lo stato.

In pratica con l'incapsulamento si può nascondere i dettagli privati di una classe da altri oggetti.

- **Polimorfismo:** Supponiamo che abbiamo una classe genitore e alcune classi secondarie che ereditano da essa. A volte vogliamo usare una raccolta, (ad esempio una lista) che contiene un mix di tutte queste

classi. Oppure abbiamo implementato un metodo per la classe genitore, ma vorremmo usarlo anche per i figli.

Questo può essere risolto usando il polimorfismo.

In poche parole, il polimorfismo offre un modo per usare una classe esattamente come il suo genitore, lasciando invariate le classi figlio, le quali mantengono i propri metodi così come sono.

Ciò in genere accade definendo un'interfaccia (padre) da riutilizzare. Quindi, ogni classe figlio implementa la propria versione di questi metodi.

Ogni volta che una raccolta (come una lista) o un metodo prevede un'istanza del genitore (in cui sono descritti metodi comuni), la lingua si occupa di valutare la corretta implementazione del metodo comune, indipendentemente da quale figlio viene passato.

Un concetto di utilizzo dell'operazione comune in diversi modi per l'inserimento di dati diversi.

Puoi trovare i seguenti esempi negli script
`Classe_e_oggetti.py` o `Classe_e_oggetti.ipynb`.

7.1 Classe

Come le definizioni delle funzioni iniziano con la parola chiave `def`, in Python definiamo una classe usando la parola chiave **`class`**. La prima stringa si chiama docstring, che anche se non obbligatoria, rappresenta una breve descrizione della classe. Ecco una semplice definizione di classe:

```
class primaclasse:
```

```
    """Ho creato la prima classe"""
```

```
    Pass
```

La parola chiave `class` è seguita dal nome della classe, in questo esempio, `primaclasse`. Successivamente viene usato il `pass`, un'istruzione molto spesso usata come un segnaposto dove alla fine andrà il codice e che ci consente di eseguire questo codice senza generare un errore.

Quando si crea una classe viene creato uno spazio dei nomi locale in cui sono definiti tutti i suoi attributi, che possono essere dati o funzioni.

Ci sono anche degli attributi speciali che iniziano con il doppio trattino basso (`__`). Ad esempio, `__doc__` ci fornisce la documentazione di quella classe.

Non appena definiamo una classe, viene creato un nuovo oggetto classe con lo stesso nome, che ci permette di accedere ai diversi attributi e di creare un'istanza di nuovi oggetti di quella classe.

```
class classe2:
```

```
    "Questa è la mia seconda classe"
```

```
    x = 24
```

```
    def fun(self):
```

```
        print('Prova')
```

```
print(classe2.x)
```

```
# Output -> 24
```

```
print(classe2.fun)
```

```
# Output -> <function classe2.fun at 0x0000027A9456B4C8>
```

```
print(classe2.__doc__)
```

```
# Output -> Questa è la mia seconda classe'
```

7.2 Oggetto

Quando viene creata una classe vengono poi definiti gli oggetti per tale classe, chiamati anche **istanze**.

Mentre la classe è il modello, un'istanza è una copia della classe con valori effettivi, letteralmente un oggetto appartenente a una classe specifica.

Detto in altro modo, una *classe* è come un *modulo* o un *questionario*, che definisce le informazioni necessarie. Dopo aver compilato il modulo, la tua *copia specifica* è un'istanza della classe; contiene informazioni reali rilevanti per te.

È possibile compilare più copie per creare molte istanze diverse, ma senza il modulo come guida, ci si perderebbe, non sapendo quali informazioni sono richieste. Pertanto, prima di poter creare singole istanze di un oggetto, viene definita una classe.

Quando si ha una classe, la procedura per creare un oggetto è simile a una chiamata di funzione:

```
obj = classe2()
```

La riga precedente crea un nuovo oggetto di istanza chiamato `obj` per la `classe2` creata in precedenza. Possiamo accedere agli attributi degli oggetti usando il prefisso del nome oggetto. Gli attributi possono essere dati o metodi.

Il metodo di un oggetto è l'insieme delle funzioni corrispondenti di quella classe. Qualsiasi oggetto funzione che è un attributo di classe definisce un oggetto metodo di quella classe.

Ciò significa che, poiché Classe2.fun è un oggetto funzione (attributo della classe), obj.fun sarà un oggetto metodo.

obj.fun()

Output -> Prova

Potresti aver notato il parametro self nella definizione della funzione all'interno della classe, ma abbiamo chiamato il metodo semplicemente come obj.fun() senza alcun argomento.

Ciò è dovuto al fatto che, ogni volta che un oggetto chiama il suo metodo, l'oggetto stesso viene passato come primo argomento. Quindi, obj.fun() si traduce in classe2.fun(obj).

In generale, chiamare un metodo con un elenco di n argomenti equivale a chiamare la funzione corrispondente con un elenco di argomenti inserendo l'oggetto del metodo prima del primo argomento.

Per questi motivi, il primo argomento della funzione in classe deve essere l'oggetto stesso, e convenzionalmente è chiamato self. Può essere chiamato diversamente, ma si consiglia vivamente di seguire la convenzione.

7.2.1 Costruttori in Python

Le funzioni di classe che iniziano con il doppio trattino basso (__) sono chiamate funzioni speciali in quanto hanno un significato speciale.

Di particolare interesse è la funzione __init__ (), che viene chiamata ogni volta che viene istanziato un nuovo oggetto di quella classe.

Questo tipo di funzione è anche chiamata costruttore nella programmazione orientata agli oggetti. Normalmente la si usa per inizializzare tutte le variabili.

class cane:

#attributo

specie = 'mammifero'

def __init__(self, nome, età=0):

self.nome = nome

self.età = età

def getData(self):

print("{} ha {} anni ed è {}".format(self.nome, self.età, self.specie))


```
Pluto = cane('Pluto',8)
```

```
Pluto.getData()
```

Risultato -> Pluto ha 8 anni ed è mammifero

Nel caso della classe Cane(), ogni cane ha un nome e un'età specifica, che è ovviamente importante sapere quando inizi a creare effettivamente oggetti “cani” diversi.

Ricorda: la classe serve solo a definire il Cane, non creando in realtà istanze di singoli cani con nomi ed età specifici. Allo stesso modo, la variabile self è anche un'istanza della classe. Poiché le istanze di una classe hanno valori diversi, possiamo indicare cane.nome = nome anziché self.nome = nome.

Ma poiché non tutti i cani condividono lo stesso nome, dobbiamo essere in grado di assegnare valori diversi a istanze diverse. Da qui la necessità della variabile self, che aiuterà a tenere traccia delle singole istanze di ogni classe.

Nell'esempio sopra, definiamo una nuova classe per rappresentare cani. Ha due funzioni, __init__() per inizializzare le variabili e getData() per visualizzare correttamente tali valori.

Adesso creiamo una nuova istanza della classe cane:

```
Pippo = cane('Pippo')
```

```
Pippo.attr = '5 mesi'
```

```
print((Pippo.nome, Pippo.età, Pippo.attr))
```

Risultato -> ('Pippo', 0, '5 mesi')

```
print(Pluto.attr)
```

Risultato -> 'cane' object has no attribute 'attr'

Una cosa interessante da notare nel passaggio precedente è che gli attributi di un oggetto possono essere creati al volo. Abbiamo creato un nuovo attributo attr per l'oggetto Pippo e lo leggiamo anche. Ma questo non ha creato quell'attributo per l'oggetto Pluto (questo il motivo dell'errore precedente).

Possiamo assegnare ora un comportamento (un metodo dell'istanza o dell'oggetto) alla classe Cane, nel seguente modo:

```
class cane:
```

```
    #attributo
```

```
    specie = 'mammifero'
```

```
def __init__(self, nome, età=0):
    self.nome = nome
    self.età = età

def getData(self):
    print("{} ha {} anni ed è {}".format(self.nome, self.età, self.specie))

def speak(self, sound):
    return "{} fa {}".format(self.nome, sound)
```

```
Pluto = cane('Pluto', 8)
```

```
Print(Pluto.speak("Bau Bau"))
```

Risultato: Pluto fa Bau Bau

7.2.2 Cancellare attributi e oggetti

Qualsiasi attributo di un oggetto può essere eliminato in qualsiasi momento, usando l'istruzione `del`. Prova quanto segue sulla shell Python per vedere l'output:

```
del Pippo
```

Sul comando `del Pippo`, questa associazione viene rimossa e il nome Pippo viene eliminato dallo spazio dei nomi corrispondenti. L'oggetto tuttavia continua a esistere in memoria e se nessun altro nome è associato ad esso, viene successivamente distrutto in automatico.

Questa distruzione automatica di oggetti senza riferimento in Python è anche chiamata **garbage collection**.

7.3 Ereditarietà Python

L'ereditarietà è una funzionalità potente nella programmazione orientata agli oggetti. Si riferisce alla definizione di una nuova classe con modifiche minime o nulle a una classe esistente. La nuova classe è chiamata classe derivata (o figlio) e quella da cui eredita è chiamata classe base (o genitore).

Sintassi:

class genitore:

corpo della classe genitore

```
class figlio(genitore):
```

corpo della classe figlio derivata

La classe derivata eredita le funzionalità dalla classe base, aggiungendovi nuove funzionalità. Ciò si traduce in una riutilizzabilità del codice.

7.3.1 Esempio di ereditarietà in Python

Facciamo finta di essere in un parco per cani. Esistono più oggetti Cane coinvolti in comportamenti Cane, ciascuno con attributi diversi.

Ciò significa che alcuni cani corrono, mentre alcuni si allungano e altri guardano solo altri cani. Inoltre, ogni cane ha un nome per il proprio proprietario e un'età. Un altro modo per differenziare un cane da un altro, è quello della razza, come precedentemente anticipato.

Ipotizzando di utilizzare la classe cane precedentemente creata, creiamo due classi figlie (con due razze differenti):

Classe Figlia (eredita dalla classe Cane)

```
class Labrador(cane):
```

```
    def run(self, velocità):
```

```
        return "{} corre {}".format(self.nome, velocità)
```

Classe Figlia (eredita dalla classe Cane)

```
class PastoreTedesco(cane):
```

```
    def run(self, velocità):
```

```
        return "{} corre {}".format(self.nome, velocità)
```

Le classi figlie ereditano attributi e comportamenti dalla classe genitore

```
Tom = PastoreTedesco("Tom", 4)
```

```
Tom.getData()
```

Output -> Tom ha 4 anni ed è mammifero

Le classi figlie hanno attributi e comportamenti specifici

```
print(Tom.run("velocemente"))
```

Output -> Tom corre velocemente

Per semplicità, non abbiamo aggiunto alcun attributo o metodo speciale per differenziare un Labrador da un Pastore Tedesco.

Per verificare se una classe figlia ha un padre si può usare la **funzione isinstance()**:

```
# E' Tom un'istanza di cane()?
```

```
print(isinstance(Tom, cane))
```

Output -> True

```
# E' Jamie un'istanza di cane()?
```

```
Jamie = cane("Jamie", 100)
```

```
print(isinstance(Jamie, cane))
```

Output -> True

```
# E' Jonny un'istanza di PastoreTedesco()?
```

```
Jonny = Labrador("Jonny", 6)
```

```
print(isinstance(Jonny, PastoreTedesco))
```

Output -> False

```
# È Jamie un'istanza di Tom?
```

```
print(isinstance(Jamie, Tom))
```

Output -> TypeError: isinstance() arg 2 must be a type or tuple of types

Sia Tom che Jamie sono istanze della classe cane(), mentre Jonny non è un'istanza della classe PastoreTedesco(). Quindi, come controllo di integrità, abbiamo testato se Jamie è un'istanza di Tom, il che è impossibile poiché Tom è un'istanza di una classe anziché una classe stessa, da cui la ragione del TypeError.

Allo stesso modo, **issubclass()** viene utilizzato per verificare l'ereditarietà delle classi.

```
issubclass(cane, Labrador)
```

Output -> False

```
issubclass(Labrador,cane)
```

```
# Output -> True
```

Cane non è una sottoclasse di Labrador, ma lo è il contrario.

Un'ultima precisazione: può succedere che il metodo `__init__` venga inizializzato sia nella classe genitore, che nella classe figlia. In questa situazione, succede che il metodo nella classe derivata sovrascrive quello nella classe base.

Questo concetto viene definito **Override**.

7.3.2 Metodo Override

L'override è una parte molto importante della programmazione orientata agli oggetti, poiché fa sì che l'ereditarietà utilizzi tutta la sua potenza. Utilizzando il metodo override una classe può "copiare" un'altra classe, evitando il codice duplicato e allo stesso tempo può migliorarne o personalizzarne una parte.

In Python l'override avviene semplicemente definendo nella classe figlio un metodo con lo stesso nome di un metodo nella classe genitore.

Quando si definisce un metodo nell'oggetto si rende quest'ultimo in grado di soddisfare quella chiamata al metodo, quindi le implementazioni dei suoi genitori non entrano in gioco.

Ad esempio riprendiamo la classe cane come genitore:

```
# classe cane genitore
```

```
class cane:
```

```
    #attributo
```

```
    specie = 'mammifero'
```

```
    def __init__(self, nome, età=0):
```

```
        self.nome = nome
```

```
        self.età = età
```

```
    def getData(self):
```

```
        print("{} ha {} anni ed è {}".format(self.nome,self.età, self.specie))
```

```
    def speak(self, sound):
```

```
        return "{} fa {}".format(self.nome, sound)
```

Se definiamo una classe figlia SiberianHusky e quest'ultima associamo una funzione Getdata come quella della classe padre, succede che viene richiamata quest'ultima e non quella della classe base, quando creo un oggetto SiberianHusky:

classe SiberianHusky figlia

```
class SiberianHusky(cane):
```

```
    def getdata(self):
```

```
        return "Mi chiamo {}".format(self.nome)
```

```
c = SiberianHusky('Rex')
```

```
print(c.getdata())
```

#Output:

#Mi chiamo Rex

8 Errori ed eccezioni

In Python possiamo incorrere spesso in qualche tipo di errore se non digitiamo il codice correttamente. Questi errori possono in genere classificarsi come:

- Errori sintattici;
- Errori logici (eccezioni).

Vediamoli brevemente.

Puoi trovare i seguenti esempi negli script
`Errori_ed_Eccezioni.py` o `Errori_ed_Eccezioni.ipynb`.

8.1 Errori sintattici

Gli errori di sintassi sono errori che avvengono quando ci si sbaglia a scrivere il codice. Quindi quando si commettono errori di ortografia, ci si dimentica ad esempio una virgola, o non si utilizza l'indentazione che il programma si aspetta.

Ad esempio, se scrivo la seguente istruzione:

```
a = 0
```

```
while a = 6:
```

```
# Output: SyntaxError: invalid syntax
```

L'errore è dovuto al fatto che non ho indicato il corpo del ciclo, ma solo l'istruzione di inizializzazione.

8.2 Errori logici

Gli errori che si verificano in fase di esecuzione (dopo aver superato il test di sintassi) sono chiamati eccezioni o errori logici.

Ad esempio, si verificano quando proviamo ad aprire un file (per la lettura) che non esiste (**FileNotFoundError**), proviamo a dividere un numero per zero (**ZeroDivisionError**) o proviamo a importare un modulo che non esiste (**ImportError**).

Ogni volta che si verificano questi tipi di errori di runtime, Python crea un oggetto eccezione. Se non gestito correttamente, stampa un traceback a quell'errore insieme ad alcuni dettagli sul perché quell'errore si è verificato.

Diamo un'occhiata a come Python tratta questi errori.

8.2.1 Eccezioni gestite da Python

Gli errori logici sollevano eccezioni. È possibile visualizzare tutte le eccezioni integrate usando la funzione built-in `local()` come segue:

```
print(dir(locals()['__builtins__']))
```

Risultato:

```
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning', 'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError', 'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning', 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False', 'FileExistsError', 'FileNotFoundError', 'FloatingPointError', 'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning', 'IndentationError', 'IndexError', 'InterruptedError', 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'LookupError', 'MemoryError', 'ModuleNotFoundError', 'NameError', 'None', 'NotADirectoryError', 'NotImplemented', 'NotImplementedError', 'OSError', 'OverflowError', 'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError', 'RecursionError', 'ReferenceError', 'ResourceWarning', 'RuntimeError', 'RuntimeWarning', 'StopAsyncIteration', 'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'TimeoutError', 'True', 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning', 'ValueError', 'Warning', 'WindowsError', 'ZeroDivisionError', '__IPYTHON__', '__build_class__', '__debug__', '__doc__', '__import__', '__loader__', '__name__', '__package__', '__spec__', 'abs', 'all', 'any', 'ascii', 'bin', 'bool', 'breakpoint', 'bytearray', 'bytes', 'callable', 'cell_count', 'chr', 'classmethod', 'compile', 'complex', 'copyright', 'credits', 'debugcell', 'debugfile', 'delattr', 'dict', 'dir', 'display', 'divmod', 'enumerate', 'eval', 'exec', 'filter', 'float', 'format', 'frozenset', 'get_ipython', 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass', 'iter', 'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview', 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'range', 'repr', 'reversed', 'round', 'runcell', 'runfile', 'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']
```

`locals() ['__builtins__']` restituirà un modulo di eccezioni, funzioni e attributi incorporati. `dir` ci permette di elencare questi attributi come stringhe.

Alcune delle eccezioni incorporate comuni nella programmazione Python insieme agli errori che le causano sono elencate di seguito:

Eccezione	Descrizione
AssertionError	Generato quando l'istruzione <code>assert</code> ha esito negativo
AttributeError	Generato sull'assegnazione o sul riferimento dell'attributo non riuscito
EOFError	Generato quando la funzione <code>input()</code> raggiunge la condizione di fine file

FloatingPointError	Generato quando un'operazione in virgola mobile non riesce
GeneratorExit	Generato quando viene chiamato il metodo close() di un generatore.
ImportError	Generato quando il modulo importato non viene trovato
IndexError	Generato quando l'indice di una sequenza è fuori portata
KeyError	Generato quando non viene trovata una chiave in un dizionario
KeyboardInterrupt	Generato quando l'utente preme il tasto di interruzione (Ctrl + c o cancella)
MemoryError	Generato quando un'operazione esaurisce la memoria
NameError	Generato quando non viene trovata una variabile nell'ambito locale o globale
NotImplementedError	Generato da metodi astratti
OSError	Generato quando un'operazione di sistema provoca un errore relativo al sistema
OverflowError	Generato quando il risultato di un'operazione aritmetica è troppo grande per essere rappresentato
ReferenceError	Generato quando un proxy di riferimento debole viene utilizzato per accedere a un referente garbage collection
RuntimeError	Generato quando un errore non rientra in nessun'altra categoria
StopIteration	Generato dalla funzione next() per indicare che l'iteratore non restituisce ulteriori elementi
SyntaxError	Generato quando si verifica un errore di sintassi
IndentationError	Generato in caso di rientro errato
TabError	Generato quando il rientro è costituito da tab e spazi incoerenti
SystemError	Generato quando l'interprete rileva un errore interno
SystemExit	Generato dalla funzione sys.exit()
TypeError	Generato quando una funzione o un'operazione viene applicata a un oggetto di tipo errato
UnboundLocalError	Generato quando viene fatto riferimento a una variabile locale in una funzione o in un metodo, ma nessun valore è stato associato a quella variabile
UnicodeError	Generato quando si verifica un errore di codifica o decodifica relativo a Unicode
UnicodeEncodeError	Generato quando si verifica un errore relativo a Unicode durante la codifica
UnicodeDecodeError	Generato quando si verifica un errore relativo a Unicode durante la decodifica
UnicodeTranslateError	Generato quando si verifica un errore relativo a Unicode durante la traduzione

ValueError	Generato quando una funzione ottiene un argomento di tipo corretto ma valore improprio
ZeroDivisionError	Generato quando il secondo operando di un'operazione di divisione o modulo è zero

Quando si verificano queste eccezioni, l'interprete Python interrompe il processo corrente e lo passa al processo chiamante fino a quando non viene gestito. Ciò significa che il programma si arresterà in modo anomalo.

Ad esempio, consideriamo un programma in cui abbiamo una funzione A che chiama la funzione B. Se si verifica un'eccezione nella funzione B che non viene gestita, l'eccezione passa quindi ad A.

Se mai gestito, viene visualizzato un messaggio di errore e il nostro programma si ferma improvvisamente.

Per gestire queste anomalie è possibile usare una delle seguenti clausole: try, except e finally.

8.2.2 Try ed Except

L'operazione critica che può sollevare un'eccezione viene inserita nella clausola try. Il codice che gestisce le eccezioni è scritto nella clausola Except.

Possiamo quindi scegliere quali operazioni eseguire una volta rilevata l'eccezione. Ad esempio l'istruzione:

```
5 + 'c'
```

genera il seguente errore:

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

Possiamo gestire in questo modo il typeError.

try:

```
n1= int(input('Inserisci un numero: '))
n2 = int(input('Inserisci un secondo numero: '))
print ("il risultato è:", n1+n2)
```

except TypeError:

```
print("Attenzione, non è possibile sommare " + str(type(n1)) + " con " + str(type(n2)) +
      ": reinserisci i valori")
```

Se nel precedente codice utilizziamo due numeri verrà restituito la somma di due numeri interi.

Se scegliamo 5 e la lettera c, otteniamo:

```
#Inserisci un numero: 5
```

```
#Inserisci un secondo numero: c

#Traceback (most recent call last):

# File "<iPython-input-92-ef6a543c150d>", line 3, in <module>

#   n2 = int(input('Inserisci un secondo numero: '))

#ValueError: invalid literal for int() with base 10: 'c'
```

Possiamo gestire anche il ValueError, come il TypeError:

try:

```
n1= int(input('Inserisci un numero: '))

n2 = int(input('Inserisci un secondo numero: '))

print ("il risultato è:", n1+n2)
```

except TypeError:

```
print("Attenzione, non è possibile sommare i due numeri: reinserisci i valori")
```

except ValueError:

```
print("Attenzione, non è possibile convertire in intero uno dei due valori inseriti")
```

Risultato:

Inserisci un numero: 5

Inserisci un secondo numero: c

Attenzione, non è possibile convertire in intero uno dei due valori inseriti

Quindi le precedenti righe di codice vengono eseguite correttamente se vengono inseriti 2 interi. Nel caso in cui si inserisca una stringa o un altro dato non intero, viene visualizzato il messaggio dislocato nell'istruzione except.

In generale, una clausola try può avere un numero qualsiasi di clausole except per gestire diverse eccezioni, tuttavia, solo una verrà eseguita nel caso in cui si verifichi un'eccezione.

Per semplicità, si può anche utilizzare una tupla di valori per specificare più eccezioni in una clausola except. Ad esempio, il codice precedente può essere riscritto nel seguente modo:

try:

```
n1= int(input('Inserisci un numero: '))

n2 = int(input('Inserisci un secondo numero: '))
```

```
print ("il risultato è:", n1+n2)
except (TypeError, ValueError):
    print("Attenzione, non è possibile sommare i due numeri: reinserisci i valori")
```

Il risultato diventa il medesimo di quello visto poco fa.

Ancora: poiché ogni eccezione in Python eredita dalla **classe Exception** di base, possiamo anche eseguire l'attività sopra descritta nel modo seguente:

```
try:
    n1= int(input('Inserisci un numero: '))
    n2 = int(input('Inserisci un secondo numero: '))
    print ("il risultato è:", n1+n2)
except Exception:
    print("Attenzione, non è possibile sommare i due numeri: reinserisci i valori")
```

8.2.3 Try con clausola else

In alcune situazioni, potresti voler eseguire un determinato blocco di codice, se il blocco di codice all'interno di try viene eseguito senza errori. In questi casi, è possibile utilizzare la parola chiave else opzionale con l'istruzione try.

Nota: le eccezioni nella clausola else non sono gestite dalle clausole precedenti except.

Esempio: stampiamo il quadrato dei soli numeri dispari.

```
try:
    num = int(input("inserisci un numero: "))
    assert num % 2 == 1
except:
    print("Non hai inserito un numero dispari")
else:
    num2 = num * num
    print ("il quadrato del numero dispari "+str(num)+" è pari a", num2)
```

Se come numero scelgo 16, ottengo: “Non hai inserito un numero dispari”

Al contrario se scelgo il numero 13 ottengo: “il quadrato del numero dispari 13 è pari a 169”

8.2.4 Raise

Nella programmazione Python vengono sollevate eccezioni quando si verificano errori in fase di esecuzione. Possiamo anche sollevare manualmente le eccezioni usando la parola chiave **raise**.

Possiamo facoltativamente passare valori all'eccezione per chiarire perché tale eccezione è stata sollevata.

```
listacasuale = [0, 3, 5]

for i in listacasuale:
    if i == 0:
        raise ZeroDivisionError("Attenzione: divisione per 0!")
    else:
        r = 1/int(i)
        print("Il valore della lista è pari a ", i)
        break
    print("Il reciproco di ", i, "è", r)
```

Output:

Traceback (most recent call last):

File "<iPython-input-133-8e0ecfc2ba98>", line 5, in <module>

raise ZeroDivisionError("Attenzione: divisione per 0!")

ZeroDivisionError: Attenzione: divisione per 0!

8.2.5 Finally

L'istruzione try in Python può avere una clausola **finally** opzionale. Questa clausola viene eseguita in qualsiasi caso e viene generalmente utilizzata per rilasciare risorse esterne.

Ad esempio, potremmo essere connessi a un centro dati remoto attraverso la rete o lavorare con un file o una GUI (Graphical User Interface).

In tutte queste circostanze, dobbiamo ripulire la risorsa prima che il programma si fermi, indipendentemente dal fatto che sia stato eseguito correttamente. Queste azioni (chiusura di un file, una GUI o disconnessione dalla rete) vengono eseguite nella clausola finally per garantire l'esecuzione.

Il blocco finally viene sempre eseguito dopo la normale chiusura del blocco try o dopo la fine del blocco try a causa di qualche eccezione.

```
file = open('test.txt', 'w')  
  
try:  
    file.write("Testing.")  
    print("Sto scrivendo sul file test.txt.")  
except IOError:  
    print("Non potresti sovrascrivere il file.")  
else:  
    print("Scrittura eseguita correttamente.")  
finally:  
    file.close()  
    print("File chiuso.")
```

Risultato:

Sto scrivendo sul file test.txt.

Scrittura eseguita correttamente.

File chiuso.

Conclusione

Bene, sei arrivato o arrivata in fondo. Complimenti!

E ora?

Se hai letto fino a qua probabilmente ti interessa imparare o tenere rinfrescato Python. Il modo più semplice per farlo è trovare tempo, possibilmente ogni giorno, per testare, fare pratica e scrivere qualche linea di codice, anche solo per divertimento.

Se sei alle prime armi, non fare come me che sono partito in quarta iniziando a studiare il machine learning senza una buona conoscenza di questo linguaggio (l'ho imparato mano a mano che studiavo gli algoritmi, di cui ho pubblicato nel blog). Ammetto che tutt'ora lo sto imparando per migliorarlo, ma soprattutto per non dimenticarlo: chi si ferma è perduto!

Se ne hai la possibilità, conosci qualcuno già che è un programmatore o esperto in questo linguaggio, fatti consigliare. Poi, man mano che impari tieni traccia dei tuoi progressi, serve come stimolo per continuare ad imparare e a raggiungere gli obiettivi.

Appena raggiungi un minimo di conoscenza puoi subito sviluppare un'app, un gioco, o chissà studiare il machine learning :D

Ti auguro una buona fortuna

Lorenzo