Manuale di Python

Tavola dei contenuti

Cosa è Python

Python è un linguaggio di programmazione ad alto livello. È molto versatile e ha applicazioni in ogni campo; dall'analisi dei dati e matematica, alla realizzazione di software.
La sintassi di Python è molto semplice e adatta ai novizi. Essendo un linguaggio interpretato e non compilato funziona allo stesso modo su diverse piattaforme (Windows, Mac, Linux...)

Sintassi di Python

Come linguaggio di programmazione, lo scopo di Python è dare istruzioni da eseguire al computer. Queste sono eseguite dall'alto verso il basso, quindi un codice simile darà in output prima *Hello* e poi *World*:

```
print('Hello')
print('World')
# La funzione print() stampa in console il contenuto passato in input.
# Vedremo meglio le funzioni più avanti
```

Commenti

Nell'esempio di prima è stato usato un commento. Un commento è semplicemente una parte di codice che verrà ignorata. Per usarli bisogna inserire un hashtag (#) e tutto quel che viene dopo verrà ignorato.
Un commento può essere usato per spiegare del codice, come sopra, oppure per ignorare temporaneamente una parte di codice a scopo di *debugging* (La ricerca e correzione di errori di funzionamento in un programma).
Per avere commenti su più righe di codice basta inserire l'hastag all'inizio di ogni riga, oppure usare le *multiline string*, stringhe su più righe che verranno ignorate perché non assegnate a nessuna variabile.

```
# Questo è
# un commento su più righe

'''
Anche questo è un commento su più righe,
ma con una stringa multilinea
'''
```

Indentazione e blocchi di codice

Nella programmazione i *blocchi di codice* sono sezioni del programma che vengono eseguite secondo particolari condizioni o funzionalità, che vedremo più avanti.
I blocchi di codice in Python sono riconosciuti dall **indentazione**, ovvero gli spazi all'inizio di una riga di codice inseriti con il tabulatore, e dai due punti **:** che ne indicano l'inizio. Visivamente il tutto rende meglio che a parole:

```
if 3 < 5:
    print('3 è minore di 5')
    # Questa parte indentata è il blocco di codice
```

Variabili

Le variabili sono paragonabili a dei contenitori che immagazzinano dei dati, come un numero o una stringa di testo.
Per creare una variabile basta *assegnare* un valore ad un nome, usando l'operatore di assegnazione: l'**uguale** (=). L'esatta sintassi è questa:

```
x = 3
# x è il nome della variabile e vi assegnamo il valore 3 con l'uguale
```

Possono essere anche passate in input a delle funzioni, dove in realtà sara passato il valore che immagazzinano:

```
x = 3
print(x)
print(x * 2) # L'asterisco è l'operatore matematico per la moltiplicazione
# L'output sarà:
# 3
# 6
```

Una variabile può essere nominata solo con lettere, numeri e trattini bassi, ma non può iniziare con un numero e non può o essere nominata come una funzione predefinita in Python.

```
variabile_1 = None # Nome valido
2variabile = None # Nome NON valido, inizia con un numero
print = None # Nome NON valido, print è una funzione di Python
```

Nell'esempio sopra possiamo vedere che le variabili hanno come valore *None*. Questo è semplicemente un valore nullo, e può essere utilizzato in vari modi, come inizializzare una variabile senza effettivamente darle un valore.

Se ci fosse bisogno di cancellare una variabile si usa una *parola chiave*, parole che hanno una funzione particolare. In questo caso usiamo **del**, che sta per *delete* (*cancellare* in italiano):

```
variabile = 12

print(variabile) # Controlliamo che la variabile esista

del variabile

print(variabile) # Ora Python darà errore perché la variabile non esiste più
```

Tipi di dati

Se le variabili immagazzinano dei dati, quali sono questi dati? Si può controllare il tipo di una variabile con la funzione **type()**.
I tipi di dati, o almeno quelli più utilizzati e importanti, sono:

Numeri

Sono abbastanza autoesplicativi. Ci sono due tipi: **int** e **float**.
Un *int* è un integrale, un numero positivo o negativo senza cifre decimali. Un *float* invece è un numero positivo o negativo con cifre decimali (Nota che non si usa la virgola, **ma il punto**).

```
x = 4
y = 5.6
print(type(x)) # L'output sarà int
print(type(y)) # L'output sarà float
```

In Python, anche se un *int* ed un *float* hanno lo stesso valore numerico, non saranno la stessa cosa:

```
x = 5
y = 5.0
print(type(x)) # L'output sarà int
print(type(y)) # L'output sarà float
```

Un *float* può essere anche definito senza cifre dopo il punto, che verranno considerate come zero:

```
x = 5. # È uguale a 5.0
```

Stringhe

Le stringhe sono un contenitore per del testo, come un nome o un indirizzo. Si usano con le **virgolette** doppie (") o singole (')

```
stringa1 = 'Questa è una stringa'
stringa2 = "Anche questa è una stringa"
```

Abbiamo visto le *multiline string* come commenti, ma essendo stringhe queste possono essere salvate anche in variabili o passate come input nelle funzioni:

```
stringa = '''Questa stringa
è scritta
su più righe'''
```

Una stringa, come testo, dovrebbe poter contenere apostrofi nonostante sia stata aperta con uno di questi. Ciò chiuderebbe la stringa precocemente. Le soluzioni sono due:
La prima, meno elegante, sarebbe utilizzare le doppie virgolette e viceversa

```
stringa1 = "Questa stringa ha un apostrofo ' "
stringa2 = 'Questa stringa ha delle virgolette " '
```

La seconda, più elegante, è usare lo **backslash** (\) che permette di inserire caratteri nelle stringhe che avrebbero altrimenti funzioni particolare (Come l'aprire e il chiudere la stringa, appunto).

```
stringa1 = 'Qui l'apostrofo termina la stringa'
# Python darà errore
```

```
stringa2 = 'Qui l\'apostrofo non termina la stringa'
# Python non darà errore
```

Il backslash ha anche altre funzioni nelle stringhe, qui alcune:

Codice	Risultato
\'	Inserisce l'apostrofo senza problemi nella stringa
\"	Inserisce le virgolette senza problemi nella stringa
\\	Inserisce il backslash nella stringa
\n	Va a capo
\t	Inserisce degli spazi come dal tabulatore

Le stringhe si possono anche concatenare ad altre stringhe o anche altri tipi di dati.
Ci sono quattro metodi per farlo:

- La virgola (,)
- Il più (+)
- L'asterisco (*)
- Le *f-string*

La virgola e il più concatenano le stringhe allo stesso modo in modo molto semplice, con la differenza che la virgola aggiunge uno spazio in mezzo:

```
print('Ciao', 'Mondo') # Stampa Ciao Mondo
print('Ciao' + 'Mondo') # Stampa CiaoMondo
```

Con l'asterisco e le *f-string* non si tratta di vera e propria concatenazione, ma qualcosa di simile.
L'asterisco moltiplica le stringhe, ripetendole:

```
print('Ciao' * 3) # Stampa CiaoCiaoCiao
```

Le *f-string* sono forse la maniera più semplice per inserire variabili nelle stringhe e andrebbero sempre usate. Basta porre una f prima dell'aperture della stringa e il valore da inserire dentro delle parentesi graffe nella stringa:

```
x = 4
print(f'x è uguale a {x}') # Stampa x è uguale a 4
```

Booleani

Programmando spesso c'è bisogno di valutare espressioni, matematiche e non, e sapere se sono vere o false. L'output di queste valutazioni è un valore **booleano** e questo può essere *True* (Vero) o *False* (False) Vedremo meglio le valutazioni più avanti.

```
bool1 = 5 > 3 # Il valore della variabile è True
bool2 = 5 < 3 # Il valore della variabile è False
```

Liste e tuple

Liste e tuple sono due tipi di dati simili che hanno la funzione di immagazzinare una collezione di dati. Entrambe sono ordinate, ovvero hanno un ordine specifico che generalmente non cambia e questo permette elementi duplici, ma le liste sono mutabili (Possono avere i propri elementi modificati) mentre le tuple no.
Le liste si definiscono con le parentesi *quadre* [] mentre le tuple con le *tonde* (), e gli elementi all'interno sono separati da virgole:

```
lista = [1, 2, 5, 4, 6, 2, 1]
tupla = ('banana', 'mela', 'arancia', 'banana')
```

Gli elementi non devono essere per forza tutti dello stesso tipo, ma anche di tipi diversi:

```
lista = [1, 2, 'banana']
tupla = (5, 'mela', True, 7.4)
```

Gli elementi di liste e tuple sono ordinati per indice, che parte da 0. Per recuperare il valore di un elemento da una lista o una tupla bisogna inserire l'indice dell'elemento dentro delle parentesi quadre vicino al nome della variabile:

```
lista = [1, 2, 7, 4, 6]
print(lista[0]) # L'output sarà 1
print(lista[2]) # L'output sarà 7
```

Come detto prima le liste sono mutabili e le tuple no, quindi per modificare un valore basterà riassegnarlo come fosse una variabile, puntando all'indice della lista:

```
lista = [1, 2, 3, 4]
tupla = (5, 6, 7, 8)

lista[2] = 10 # Il 3 nella lista ora sarà 10

print(lista) # L'output sarà [1, 2, 10, 4]

tupla[2] = 12 # Qui Python darà errore, perché le tuple sono immutabili
```

Ordinato e non ordinato

Il concetto di ordinato e non si applica alle **liste**, **tuple**, **dizionari** e **set** ed è un concetto un po' controintuitivo:

- **Ordinato**: non intende ordinato numericamente, alfabeticamente etc. ma come ordine ben definito:

```
lista = [3, 2, 4, 1]
print(lista)
'''
Stampa la lista così com'è definita perchè l'ordine è definito e non
automatico
'''
```

- **Non ordinato**: quando qualcosa non è ordinato, come un set o un dizionario, il suo ordine viene automaticamente definito alla sua creazione, che sia numerico, alfabetico etc.

```
set1 = {3, 2, 4, 1}
print(lista)
'''
Stampa {1, 2, 3, 4} perché viene ordinato automaticamente, visto che non ha
un ordine definibile dall'utente
'''
```

Dizionari

Un altro tipo di dato che contiene più elementi è il **dizionario**. Un dizionario è composto da:

- *Chiavi*
- *Valori*

Ogni chiave ha un valore assegnato e la coppia di questi due è un **elemento** del dizionario. I dizionari si aprono con le **parentesi graffe** {} e per definire il valore di una chiave si usano i due punti:

```
dizionario = {'chiave': 'valore'}
```

Un valore può essere un qualunque tipo di dato, anche un altro dizionario, ma la chiave può essere solo un numero o una stringa.
Per recuperare un valore di un dizionario, come nella lista, si usano le parentesi quadre con dentro la chiave:

```
dizionario = {'chiave1': 1, 'chiave2': 2}
print(dizionario['chiave1']) # L'output sarà 1
print(dizionario['chiave2']) # L'output sarà 2
```

I dizionari sono mutabili, quindi si possono modificare gli elementi o anche aggiungerne o rimuoverne.
Per modificarli si procede allo stesso modo delle liste.

Per aggiungerli basta assegnarli dopo la creazione del dizionario, per esempio:

```
dizionario = {
  'nome': 'Paolo',
  'eta': 21
}
print(dizionario)
# L'output qui sarà {'nome': 'Paolo', 'eta': 21}

dizionario['citta'] = 'Napoli'
print(dizionario)
# L'output qui sarà {'nome': 'Paolo', 'età': 21, 'citta': 'Napoli'}
```

Per cancellarli basta usare la parola chiave **del** come nelle variabili:

```
dizionario = {
  'nome' = 'Paolo',
  'eta' = 21,
  'citta' = 'Napoli'
}

print(dizionario) # L'output sarà {'nome' = 'Paolo', 'eta' = 21, 'citta' =
'Napoli'}

del dizionario['eta']

print(dizionario) # L'output sarà {'nome' = 'Paolo', 'citta' = 'Napoli'}
```

Set

L'ultimo tipo di dato che contiene elementi è il **set**. Si scrive come le liste, eccetto invece che le parentesi quadre si usano le *graffe*.
I set non sono ordinati e i suoi elementi sono immutabili, ma se ne possono aggiungere o rimuovere. Inoltre non permette elementi doppi:

```
set1 = {1, 3 , 2, 4, 2}
print(set1)
# Stamperà {1, 2, 3, 4}, eliminando i doppioni e ordinando gli
# elementi automaticamente
```

Casting

Il *casting* è l'operazione che consiste nel convertire un tipo di dato in un altro. Per esempio si potrebbe convertire una stringa `'21'` in un integrale `21`.
Il *casting* si attua con funzioni specifiche per ogni tipo di dato:

Funzione	Dato nel quale converte
int()	Integrale (*int*)
float()	Fluttuante (*float*)
str()	Stringa
bool()	Booleano
list()	Lista
tuple()	Tupla
dict()	Dizionario
set()	Set

Operatori logici e dichiarazioni condizionali

Abbiamo visto come un valore booleano si possa ottenere dalle valutazioni di espressioni. Per eseguire queste valutazioni ci serviamo degli **operatori logici**, come in matematica:

Operatore	Significato
==	Uguale
!=	Non uguale
<	Minore
>	Maggiore
<=	Minore o uguale
>=	Maggiore o uguale

Quindi per esempio:

```
bool1 = 5 == 5 # True
bool2 = 5 == 7 # False
bool3 = 3 <  4 # True
```

Gli operatori *uguale e non uguale* (== e !=) non servono solo in espressioni matematiche, ma anche in termini di uguaglianza fra altri tipi di dati, per esempio:

```
bool1 = "ciao" == "ciao" # True
bool2 = [1, 2, 3] != [1, 3, 2] # True
```

```
bool3 = "salve" == "Salve" # False
bool4 = "5" == 5
```

Questi valori sono utili nelle **dichiarazioni condizionali** (*conditional statement* in inglese), che eseguono un blocco di codice se una determinata condizione è vera e si scrivono usando la parola chiave *if*:

```
x = float(input())
if x > 5:
  print(x, 'è maggiore di 5')
```

I *conditional statement* non si fermano qui. Nel caso l'espressione fosse falsa possiamo passare al computer altre istruzioni da eseguire, questo con la parola chiave *else*:

```
x = float(input())
if x > 5:
  print(x, 'è maggiore di 5')
else:
  print(x, 'è minore di 5')
```

C'è una terza parola chiave che serve a effettuare altri controlli nello stesso *conditional statement*, ***elif***:

```
x = float(input())
if x > 5:
  print(x, 'è maggiore di 5')
elif x == 5:
  print(x, 'è uguale a 5')
else:
  print(x, 'è minore di 5')
```

I diversi blocchi di codice in un *conditional statement* non potranno mai essere eseguiti insieme, anche se entrambi veri, per esempio:

```
x = 15
if x > 5:
  print(x, 'è maggiore di 5')
elif x > 10:
  print(x, 'è maggiore di 10')
'''
Qui verrà stampato solo:
15 è maggiore di 5
perché essendo lo stesso conditional statement non possono essere eseguiti insieme
i diversi blocchi di codice che contiene
'''
```

```
if x > 5:
  print(x, 'è maggiore di 5')
if x > 10:
  print(x, 'è maggiore di 10')
'''
Qui verrà stampato sia:
15 è maggiore di 5
sia:
15 è maggiore di 10
perché si tratta di conditional statement diversi, quindi possono essere eseguiti
insieme
'''
```

Il codice è eseguito dall'alto verso il basso, quindi solo la prima espressione vera eseguirà il blocco di codice.

Cicli

Per ciclo si intende un blocco di codice ripetuto più volte. Ci sono due cicli: **for** e **while**.

Cicli *for*

I cicli *for* sono cicli che si ripetono per un determinato numero di volte. Questo numero è dato dall'**iterabile** utilizzato nel ciclo. Cos'è un iterabile? È un elemento che ne contiene altri analizzabili uno ad uno. Gli iterabili sono **liste**, **tuple**, **dizionari**, **set** e **stringhe**.
Un esempio renderà più chiaro il tutto:

```
lista = [1, 2, 3, 5, 4]
for i in lista:
  print(i)
  '''
  La sintassi è: for i in iterabile:
  dove "i" è l'elemento analizzato e può essere chiamato in tutti i modi e
"iterabile" è l'iterabile iterato.
  Questo codice stampa ogni volta l'elemento analizzato, quindi l'output sarà:
  1
  2
  3
  5
  4
  '''
```

Ovviamente gli elementi iterabili di una lista o tupla sono gli elementi che contiene; per una stringa i suoi elementi sono i suoi caratteri, compresi spazi e punteggiatura:

```
for carattere in 'Ciao mondo!':
  print(carattere)
  '''
```

```
  Stamperà:
  C
  i
  a
  o

  m
  o
  n
  d
  o
  !
  '''
```

La funzione *range()*

La funzione *range()* fornisce un iterabile che va di default da zero al numero specificato. Il numero specificato è però escluso:

```
for i in range(10):
  print(i)
'''
Stampa:
0
1
2
3
4
5
6
7
8
9
'''
```

La funzione *range()* ha tre parametri:

- *start*: il numero dal quale inizia a contare, incluso
- *stop*: il numero fino a dove conta, escluso
- *step*: gli incrementi

Di base, se viene inserito un solo parametro, viene inserito lo *stop*, e lo *start* è 0 di base, mentre lo *step* 1.

Quindi se volessi contare a incrementi di 2 da 0 a 10 (incluso) dovrò scrivere:

```
for i in range(0, 11, 2):
  print(1)
'''
Stampa:
```

```
0
2
4
6
8
10
'''
```

Ciclo *while*

Il ciclo *while* continua a eseguire il blocco di codice fino a quando una determinata condizione condizione è vera. Quando questa condizione risulta falsa il ciclo si ferma:

```
i = 0
while i < 10:
  print(i)
  i = i + 1
  '''
  Qui la condizione per il ciclo è che "i" sia minore di 10. Alla fine del ciclo
aumentiamo "i" di 1, così una volta che "i" sarà 10 il ciclo si fermerò. Alla fine
il ciclo stamperà:
  0
  1
  2
  3
  4
  5
  6
  7
  8
  9
  '''
```

Se c'è il bisogno di terminare un ciclo, basta inserire la parola chiave ***break*** e il ciclo si fermerà.

Funzioni

Una funzione è un blocco di codice che può essere eseguito quando richiamato. Più volte negli esempi è stata usata la funzione *print()*, che stampa nella console quel che passi nelle parentesi, ed è solo un esempio delle tante funzioni già definite in Python.
Non siamo limitati solo a usare le funzioni predefinite, ma possiamo anche scrivere le nostre funzioni.
Per definire una funzione si usa la parola chiave *def*, seguito dal nome della funzione e delle parentesi:

```
def una_funzione():
  pass # pass è una parola chiave che serve a
       # definire una funzione nulla, facendo
       # in modo che Python non dia errore
```

Le funzioni possono anche avere dei **parametri**, valori da inserire al momento del richiamo della funzione da utilizzare nel blocco di codice. Per esempio, il testo passato nella funzione *print()* è un parametro.

```
def raddoppia(numero):
  print(numero * 2)

raddoppia(4)
# L'output sarà 8
```

Nell'esempio di prima la funzione raddoppia il numero passato in input. Il risultato viene stampato in console, ma non viene salvato da nessuna parte. Una funzione del genere è sostanzialmente inutile, per questo ci serviamo della parola chiave ***return***.
La parola chiave *return* dà in output della funzione il valore specificato. Ecco un esempio per chiarire il tutto:

```
def raddoppia(numero):
  return numero * 2

variabile = raddoppia(4)

'''
Ora la variabile ha come valore l'output dato dalla funzione raddoppia(), che è il
numero in input (4) raddoppiato.
'''

print(variabile)

'''
Stamperà 8, perché 8 è il valore ritornato dalla funzione, successivamente salvato
nella variabile
'''
```

Si può anche inserire un valore di default per i parametri, nel caso nessun valore venga inserito al richiamo della funzione:

```
def raddoppia(numero=0): # Assegnamo un valore di default dentro le parentesi
  return numero * 2

print(raddoppia()) # Qui non sono inseriti valori
```