

Ripetizioni Luca

Marini Mattia

Dicembre 2025

Ripetizioni Luca is licensed under [CC BY 4.0](#) .

© 2023 [Mattia Marini](#)

Indice

1	Introduzione a python	2
2	La sintassi di base di Python	3
2.1	Commenti	3
2.2	Variabili	3
2.3	Operatori	4
2.4	Indentazione	4
2.5	Istruzioni condizionali (<code>if</code> , <code>elif</code> , <code>else a</code>)	5
2.6	Liste, set, dizionari e tuple	6
2.6.1	Liste	6
2.6.2	Dizionari	7
2.6.3	Set (insiemi)	7
2.6.4	Tuple	8
2.7	Cicli (<code>for</code> , <code>while</code>)	9
2.7.1	Il ciclo <code>for</code>	9
2.7.2	Il ciclo <code>while</code>	9
2.7.3	Iterare strutture dati	9
2.7.4	Range	10
2.8	Definizione di funzioni	11
2.8.1	Parametri posizionali	11
2.8.2	Parametri con nome (keyword arguments)	11
2.8.3	Valori di default per i parametri	11
2.9	Annotazioni di tipo	12

Python è un linguaggio di programmazione di alto livello con innumerevoli applicazioni. La sintassi "leggera" e il grande livello di astrazione che permette di ottenere lo rendono un buon linguaggio per imparare a programmare.

Fra le applicazioni più comuni di Python troviamo:

- *Uso in data science:* Python è molto usato nell'analisi dei dati, machine learning e intelligenza artificiale grazie a librerie come NumPy, pandas, scikit-learn e TensorFlow.
- *Sviluppo web:* Con framework come Django e Flask è possibile creare applicazioni web scalabili, API e backend in modo rapido e flessibile.
- *Automazione e scripting:* Python è spesso utilizzato per scrivere script che automatizzano attività ripetitive, gestione di file, scraping di dati e operazioni di sistema.
- *Sviluppo di software desktop:* Con librerie come Tkinter, PyQt e wxPython è possibile creare applicazioni desktop con interfaccia grafica.
- *Calcolo scientifico e ingegneristico:* Python viene utilizzato in ambiti tecnico-scientifici per simulazioni, calcoli numerici e visualizzazione di dati grazie a librerie come SciPy e matplotlib.
- *Programmazione di rete:* Python è usato per sviluppare server, client e tool per la gestione di protocolli di rete grazie a librerie come socket e asyncio.
- *Didattica e formazione:* Grazie alla sua sintassi semplice, Python è un linguaggio molto usato per l'insegnamento della programmazione a vari livelli.

Dal punto di vista formale possiamo dire che python è un linguaggio Dal punto di vista formale possiamo dire che Python è un linguaggio:

- *Ad alto livello:* Offre un alto grado di astrazione dalle architetture hardware, permettendo di concentrarsi sulla logica del programma piuttosto che sulla gestione delle risorse di basso livello.
- *Interpretato:* Il codice Python viene eseguito da un interprete, istruzione per istruzione, senza la necessità di una compilazione preventiva.
- *Dinamicamente tipizzato:* Non è necessario specificare il tipo delle variabili alla dichiarazione; il tipo viene determinato a runtime in base al valore assegnato.
- *Multi-paradigma:* Supporta diversi paradigmi di programmazione, tra cui quella orientata agli oggetti, imperativa, funzionale e, in misura minore, procedurale.
- *Portabile:* Il codice Python, salvo dipendenze specifiche di sistema, può essere eseguito senza modifiche su diverse piattaforme (Windows, MacOS, Linux, ecc.).
- *A gestione automatica della memoria:* L'allocazione e la deallocazione della memoria sono gestite automaticamente tramite garbage collector.

- *Estensibile*: Permette di integrare facilmente codice scritto in altri linguaggi, come C o C++, per migliorarne le prestazioni o integrare funzionalità di basso livello.
- *Open source*: L'implementazione standard (CPython) è open source e dispone di un vasto ecosistema di librerie gratuite.

2 La sintassi di base di Python

Python è noto per la sua sintassi semplice e leggibile, che favorisce la scrittura di codice chiaro e immediatamente comprensibile. In questa sezione illustreremo alcuni concetti fondamentali della sintassi di base, organizzati per argomento.

2.1 Commenti

Per aggiungere commenti, ossia stringhe di testo spesso utili a spiegare il funzionamento del codice o hints per il programmatore, si può utilizzare il carattere `#`. Queste parti del codice verranno ignorate dall'interprete, non cambiando dunque il comportamento dell'applicazione, ma rendendo la lettura del codice più facile

```
# Questo è un commento
x = 1 # Questo è un commento alla fine della riga
```

2.2 Variabili

L'assegnazione di valori a variabili avviene con l'operatore `=`. La stampa su schermo si effettua con la funzione `print()`.

```
x = 42
nome = "Alice"

# Riassegno x e nome
x = 12
nome = "nuovo nome"
```

Come si può notare non è necessario indicare un tipo di una variabile e non vi è differenza fra la sintassi di dichiarazione e di assegnamento

Per stampare a schermo si usa la funzione `print()`. La funzione può contenere una serie di parametri separati da virgola e li stamperà concatenati:

```
x = 42
nome = "Alice"
print("Ciao", nome, x) #Stampa "Ciao Alice 42"
```

Nota che essendo un linguaggio con *dynamic typing*, allora è del tutto consensito "fregarsene" del tipo delle variabili. Prendiamo come esempio questo codice, disponibile in /files/python/teoria/teo_base/var.py:

```
x = 42
nome = "Alice"
print("x:", x)
print("nome:", nome)
```

```

print('tipo di "x":', type(x))
print('tipo di "nome":', type(nome))

# Cambio il tipo dei parametri assegnando un'altro valore
print()
x = "Ora sono una stringa"
nome = 42
print("x:", x)
print("nome:", nome)
print('tipo di "x":', type(x))
print('tipo di "nome":', type(nome))

```

Questo perchè il tipo di una variabile viene determinato in fase di esecuzione del programma, in base al valore che le viene assegnato.

2.3 Operatori

In base al tipo di variabile, possiamo applicare gli operatori. L'effetto sarà diverso a seconda del tipo di dato coinvolto.

Operatore	Descrizione
+	Addizione
-	Sottrazione
*	Moltiplicazione
/	Divisione (risultato con virgola)
//	Divisione intera (arrotondata per difetto)
%	Modulo (resto della divisione intera)
**	Potenza

Tabella 1: Operazioni fra numeri

Operatore	Descrizione
+	Concatenazione di stringhe
*	Ripetizione della stringa
in	Verifica se una sottostringa è presente
len()	Restituisce la lunghezza della stringa

Tabella 2: Operazioni fra stringhe

E' disponibile un esempio di codice in /files/python/teoria/teo_base/operatori.py

2.4 Indentazione

In Python, l'indentazione non è solo stilistica, ma è fondamentale per definire i blocchi di codice. È consigliato usare 4 spazi per ogni livello di indentazione.

```

if x > 0:
    print("Numero positivo")
    print("Ancora dentro il blocco if")
print("Fuori dal blocco if")

```

2.5 Istruzioni condizionali (if, elif, else a)

Le istruzioni condizionali consentono di eseguire blocchi di codice al verificarsi di una certa condizione.

```

if x > 0:
    print("Positivo")
elif x == 0:
    print("Zero")
else:
    print("Negativo")

```

Dunque, genericamente possiamo dire che la sintassi è la seguente:

```

if <condizione booleana>:
    <blocco indentato>
elif <condizione booleana>:
    <blocco indentato>
else:
    <blocco indentato>

```

Per esprimere una condizione booleana possiamo fare uso dei seguenti operatori:

Operatore	Descrizione
==	Uguaglianza
!=	Diversità
>	Maggiore
<	Minore
>=	Maggiore o uguale
<=	Minore o uguale

Tabella 3: Operatori di confronto

In più, due o più clausole logiche possono essere combinate usando gli operatori logici:

Operatore	Descrizione
and	b1 and b2 è vero se sono vere sia b1 che b2
or	b1 or b2 è vero se sono vere o b1 o b2
!	!b1 è vera solo se b1 <i>non</i> è vera

Tabella 4: Operatori di confronto

Un esempio può essere:

```
# Numero fra -7 e 8 oppure che non sia contemporaneamente positivo e pari
if (x > -8 and x <= 8) or !(x > 0 and x % 2 == 0):
    print("Yeah")
```

2.6 Liste, set, dizionari e tuple

Per contenere insiemi di elementi (più di un elemento), python fornisce delle strutture dati molto potenti e flessibili: liste, set e dizionari.

2.6.1 Liste

La struttura dati più semplice per contenere insiemi di elementi è la lista (array). Le liste sono definite usando parentesi quadre [] e gli elementi sono separati da virgole.

```
l = [1,2,3,4] # Lista di 4 elementi
```

In python, le liste possono contenere elementi di tipi diversi:

```
l=[1, "prova", 3.2, True]
```

L'accesso all'i-esimo elemento della lista avviene tramite l'operatore []. Gli indici partono da 0

```
l=[1, "prova", 3.2, True]
print(l[0]) # 1
print(l[1]) # prova
print(l[2]) # 3.2
print(l[3]) # True
```

```
print(l[4]) # Errore -> IndexError: list index out of range
print(l[-1])# Errore -> IndexError: list index out of range
```

Inoltre, possiamo utilizzare i seguenti operatori e metodi per manipolare le liste:

Operatore/Metodo	Descrizione
+	Concatenazione di liste
*	Ripetizione della lista
in	Verifica la presenza di un elemento
len()	Restituisce la lunghezza della lista
append(x)	Aggiunge l'elemento x in fondo alla lista
extend(L)	Aggiunge tutti gli elementi della lista L
insert(i, x)	Inserisce l'elemento x in posizione i
remove(x)	Rimuove la prima occorrenza di x
pop([i])	Rimuove e restituisce l'elemento in posizione i
index(x)	Restituisce la posizione della prima occorrenza di x
count(x)	Restituisce il numero di occorrenze di x
sort()	Ordina la lista
reverse()	Inverte l'ordine degli elementi nella lista

2.6.2 Dizionari

I dizionari sono strutture dati che associano delle “chiavi” a dei “valori” (mapping). Si definiscono usando parentesi graffe {} e coppie chiave:valore separate da virgole.

```
d = {"nome": "Alice", "età": 25, "studente": True}
```

In Python, le chiavi devono essere di tipo immutabile (stringhe, numeri, tuple), mentre i valori possono essere di qualsiasi tipo.

L'accesso agli elementi avviene tramite la chiave:

```
d = {"nome": "Alice", "età": 25, "studente": True}
print(d["nome"]) # Alice
print(d["età"])  # 25

print(d["peso"]) # Errore -> KeyError: 'peso'
```

Di seguito alcuni operatori e metodi utili per manipolare i dizionari:

Operatore/Metodo	Descrizione
<code>in</code>	Verifica la presenza di una chiave
<code>len()</code>	Restituisce il numero di elementi
<code>d[k]</code>	Restituisce il valore associato alla chiave <code>k</code>
<code>d[k] = v</code>	Modifica o aggiunge la coppia key-value
<code>del d[k]</code>	Rimuove la coppia con chiave <code>k</code>
<code>get(k[, def])</code>	Restituisce il valore di <code>k</code> ; se non esiste, restituisce <code>def</code> o <code>None</code>
<code>keys()</code>	Restituisce una vista delle chiavi
<code>values()</code>	Restituisce una vista dei valori
<code>items()</code>	Restituisce una vista delle coppie chiave-valore
<code>update(d2)</code>	Aggiorna il dizionario con gli elementi di <code>d2</code>
<code>clear()</code>	Rimuove tutti gli elementi

2.6.3 Set (insiemi)

I set sono raccolte non ordinate di elementi unici. Si definiscono con le parentesi graffe {} oppure usando la funzione `set()`.

```
s = {1, 2, 3, 4}          # Set di quattro elementi
t = set([3, 4, 5, 6])    # Costruzione da una lista
```

I set non possono contenere elementi duplicati, e non sono indicizzati.

Esempi di operazioni comuni su set:

```
s = {1, 2, 3, 4}
print(2 in s)    # True
s.add(5)         # Aggiunge 5
s.remove(3)      # Rimuove 3
u = s | t        # Unione
i = s & t        # Intersezione
d = s - t        # Differenza
```

Operatori e metodi per manipolare i set:

Operatore/Metodo	Descrizione
<code>in</code>	Verifica la presenza di un elemento
<code>len()</code>	Restituisce la cardinalità del set
<code>add(x)</code>	Aggiunge l'elemento <code>x</code>
<code>remove(x)</code>	Rimuove l'elemento <code>x</code> (errore se non presente)
<code>discard(x)</code>	Rimuove l'elemento <code>x</code> (senza errore se non presente)
<code>pop()</code>	Rimuove e restituisce un elemento arbitrario
<code>clear()</code>	Svuota il set
<code>union(s2)</code> (oppure <code> </code>)	Unione di set
<code>intersection(s2)</code> (oppure <code>&</code>)	Intersezione di set
<code>difference(s2)</code> (oppure <code>-</code>)	Differenza tra set
<code>issubset(s2)</code>	Verifica se è un sottoinsieme
<code>issuperset(s2)</code>	Verifica se è un sovrainsieme

2.6.4 Tuple

Le tuple sono sequenze ordinate e immutabili di elementi. Si definiscono utilizzando le parentesi tonde () o semplicemente separando gli elementi con una virgola. Si possono intendere come liste non modificabili di dimensione fissa

```
t = (1, 2, 3)      # Tupla di tre elementi
t2 = "a", 1, True # Anche senza parentesi tonde
```

Come le liste, le tuple possono contenere elementi di diversi tipi e si accede agli elementi tramite indici (che partono da zero):

```
print(t[0]) # 1
print(t[1]) # 2
print(t[-1]) # 3
```

A differenza delle liste, le tuple sono *immutabili*: non è possibile aggiungere, modificare o rimuovere elementi dopo la loro creazione.

Ecco alcuni operatori e metodi utili per lavorare con le tuple:

Operatore/Metodo	Descrizione
<code>+</code>	Concatenazione di tuple
<code>*</code>	Ripetizione della tupla
<code>in</code>	Verifica la presenza di un elemento
<code>len()</code>	Restituisce la lunghezza della tupla
<code>t[i]</code>	Accesso all'elemento in posizione <code>i</code>
<code>index(x)</code>	Restituisce la prima posizione dell'elemento <code>x</code>
<code>count(x)</code>	Restituisce il numero di occorrenze di <code>x</code>

Il vantaggio principale delle tuple rispetto alle liste è la loro immutabilità, che le rende più sicure per dati che non devono essere modificati e può migliorare le prestazioni in alcune situazioni. Un uso molto comune è quello di utilizzarle come chiavi di dizionari:


```
d = {[1,2]: "v1", [-2, "a"]: "v2"} # Non ammesso!!
d = {(1,2): "v1", (-2, "a"): "v2"} # Valido
```

Nota che si possono costruire tuple a partire da array facilmente:

```
v1 = [1,2]
v2 = [-2,"a"]
v1_t = tuple(v1)
v2_t = tuple(v2)
d = {v1_t: "v1", v2_t: "v2"}
```

2.7 Cicli (for, while)

Python fornisce due tipi principali di ciclo: `for` e `while`.

2.7.1 Il ciclo for

Il ciclo `for` itera su una sequenza (es. lista, stringa o range numerico):

```
for i in range(5):
    print(i) # Stampa i numeri da 0 a 4

frutta = ["mela", "banana", "ciliegia"]
for elemento in frutta:
    print(elemento)
```

2.7.2 Il ciclo while

Il ciclo `while` ripete un blocco di codice finché una condizione è vera:

```
n = 3
while n > 0:
    print(n)
    n = n - 1
print("Fine ciclo")
```

2.7.3 Iterare strutture dati

Alcuni esempi sono disponibili nel file `/files/python/teoria/teo_base/iterazioni.py`

Python ci permette di iterare su strutture dati in maniera molto efficace tramite il ciclo `for`

```
lista = [1, 2, 3, 4, 5]
stringa = "Ciao mondo"
dizionario = {"a": 1, "b": 2, "c": 3}
insieme = {1, 2, 3, 4, 5}
tupla = (1, 2, 3, 4, 5)

for x in lista: # 1 2 3 4 5
    print(x, " ", end="")
```

```

print()
for x in stringa: # C i a o   m o n d o
    print(x, " ", end="")

print()
for x in dizionario: # a b c
    print(x, " ", end="")

print()
for x in insieme: # 1 2 3 4 5
    print(x, " ", end="")

print()
for x in tupla: # 1 2 3 4 5
    print(x, " ", end="")

```

L'unica struttura dati che richiede un po' di attenzione in più è il *dizionario*. Con la sintassi vista sopra, si itera sulle chiavi del dizionario. Per iterare sui valori o sulle coppie chiave-valore, si possono usare i metodi `values()` e `items()`:

```

# Chiave valore
for v in dizionario.values():
    print("Valore:", v)

# Chiave valore dizionario
for k, v in dizionario.items():
    print("Chiave:", k, "Valore:", v)

```

2.7.4 Range

Per iterare tramite indici (quindi nel caso in ci stessimo usando una lista), python fornisce la funzione `range()` che genera una sequenza di numeri interi. La sintassi è la seguente:

```

range(start_value, end_value, increment)
range(start_value, end_value)
range(end_value)

```

La sintassi completa è la prima. In questo caso la sequenza generata inizia da `start_value` (incluso) e termina a `end_value` (escluso), incrementando di `increment` ad ogni passo. Ad esempio:

`range(-3, 5, 2)` itera sui valori -3, -1, 1, 3

Tramite la seconda sintassi, si indica solo `start_value` e `end_value`. In questo caso l'incremento di default è 1. Ad esempio:

`range(-3, 5)` itera sui valori -3, -2, -1, 0, 1, 2, 3, 4

Nell'ultimo caso, si indica solo `end_value`. In questo caso il valore iniziale è 0 e l'incremento è 1:

`range(4)` itera sui valori 0, 1, 2, 3

2.8 Definizione di funzioni

Le funzioni permettono di organizzare il codice in blocchi riutilizzabili. Si definiscono con la parola chiave `def`.

```
def saluta(nome):  
    print("Ciao,", nome)  
  
saluta("Alice")  
saluta("Bob")
```

Le funzioni possono restituire valori usando la parola chiave `return`:

```
def somma(a, b):  
    return a + b  
  
risultato = somma(2, 5)  
print(risultato)
```

2.8.1 Parametri posizionali

I parametri posizionali sono gli argomenti che vengono assegnati in base all'ordine in cui sono passati alla funzione:

```
def moltiplica(x, y):  
    return x * y  
  
print(moltiplica(2, 3)) # x=2, y=3
```

2.8.2 Parametri con nome (keyword arguments)

È possibile specificare quale valore assegnare a ciascun parametro usando la sintassi `nome_parametro=valore`. In questo modo l'ordine non è più vincolante:

```
def dividi(a, b):  
    return a / b  
  
print(dividi(a=10, b=2))    # a=10, b=2  
print(dividi(b=4, a=20))    # a=20, b=4 (l'ordine non ha importanza)
```

2.8.3 Valori di default per i parametri

È possibile specificare un valore di default per uno o più parametri, che verrà utilizzato se l'argomento corrispondente non viene passato in chiamata:

```
def saluta(nome, messaggio="Ciao"):  
    print(messaggio + ", " + nome)  
  
saluta("Alice")                # Ciao, Alice  
saluta("Bob", messaggio="Salve") # Salve, Bob
```

Se vengono mescolati parametri con e senza valore di default, i parametri con default devono essere dichiarati dopo quelli senza default.

```
def f(a, b=1, c=2):
    print(a, b, c)

f(10)          # a=10, b=1, c=2
f(10, 20)      # a=10, b=20, c=2
f(10, 20, 30)  # a=10, b=20, c=30
```

2.9 Annotazioni di tipo

Python è un linguaggio dinamicamente tipizzato, il che significa che i tipi delle variabili e dei parametri non vengono controllati a tempo di compilazione. Tuttavia, a partire da Python 3, è possibile aggiungere annotazioni di tipo (type hints) per rendere il codice più leggibile e favorire la rilevazione degli errori tramite strumenti come `mypy`.

2.9.0 Annotazioni di tipo per variabili

Si può indicare il tipo atteso di una variabile con la seguente sintassi:

```
x: int = 5
nome: str = "Alice"
prezzi: list[float] = [10.5, 5.2, 7.0]
```

L'annotazione non impone alcun vincolo a runtime, ma può essere usata da editor o strumenti di controllo statico dei tipi.

2.9.0 Annotazioni di tipo nelle funzioni

Si possono annotare i tipi di parametri e del valore restituito da una funzione attraverso una sintassi dedicata:

```
def somma(a: int, b: int) -> int:
    return a + b

def saluta(nome: str) -> None:
    print(f"Ciao, {nome}")
```

- `a: int` e `b: int` specificano che i parametri `a` e `b` dovrebbero essere interi.
- `-> int` indica che la funzione dovrebbe restituire un intero, mentre `-> None` indica che la funzione non restituisce nulla.

Le annotazioni di tipo supportano che iniziano con la lettera minuscola sono native di python 3.9 e successivi. Se vogliamo utilizzare annotazioni di tipo più complesse (come liste, dizionari, tipi opzionali, ecc.) in versioni precedenti di Python, dobbiamo importare i tipi dal modulo `typing`:

```
from typing import List, Dict, Optional

def media(valori: List[float]) -> float:
    return sum(valori) / len(valori)

def ricerca(dati: Dict[str, int], chiave: str) -> Optional[int]:
    return dati.get(chiave)
```

In sintesi, le annotazioni di tipo sono strumenti facoltativi e non bloccanti che permettono di documentare meglio il codice e di rilevare più facilmente possibili errori grazie ad appositi strumenti di analisi statica.