# Guida Base di Python

Angelo Vaccaro

# Contents

# 1 Introduzione

Python è un linguaggio di programmazione ad alto livello, interpretato e orientato agli oggetti. È noto per la sua sintassi semplice e leggibile, che lo rende ideale per i principianti.

# 2 Sintassi

Questo codice chiede all'utente di inserire un messaggio e poi lo stampa.

```
messaggio = input("Inserisci un messaggio: ")
print(messaggio)
```

I commenti in Python iniziano con il simbolo # e continuano fino alla fine della riga. Non esistono i commenti multilinea come in altri linguaggi.

```
#Commento
```

L'indentazione è fondamentale in Python e serve a definire i blocchi di codice. In python a differenza di altri linguaggi se ci saranno errori di indentazione te lo farà notare, non ci saranno più punti e virgole bensì il codice sarà definito dall'indentazione.

```
if True:
    print("Questo e' vero")
else:
    print("Questo non e' vero")
```

# 3 Variabili

Una variabile è un nome che rappresenta un valore. In Python non è necessario dichiarare il tipo di variabile, poiché il linguaggio è dinamicamente tipizzato. Python a differenza di altri linguaggi non ha bisogno di dichiarare il tipo di variabile, ma lo fa in automatico.

```
x = 5
y = "Ciao"
z = 3.14
```

Una parte fondamentale delle variabili è la nomenclatura, che deve essere chiara e significativa. Alcuni nomi di variabili non sono permessi, come ad esempio:

```
1x = 5 # non e' permesso
x-y = 10 # non e' permesso
x y = 15 # non e' permesso
```

E ci sono molti altri nomi di variabili che non sono permessi, come ad esempio:

- and
- as
- assert
- break
- class
- continue
- def
- del
- elif
- else
- except
- False
- finally
- for
- from
- global
- if

- import
- in
- is
- lambda
- None
- nonlocal
- not
- or
- pass
- raise
- return
- True
- try
- while
- with
- yield

Attenzione perché in Python le variabili sono case-sensitive, quindi `variabile` e `Variabile` sono due variabili diverse. Per scrivere variabili possiamo usare diversi case:

- **snake_case**: `variabile_esempio`
- **camelCase**: `variabileEsempio`
- **PascalCase**: `VariabileEsempio`

Quello suggerito è lo **snake_case**, che è il più usato in Python.
Assegnamo multipli valori a più variabili in una sola riga:

```
x, y, z = 1, 2, 3
print(x) # 1
print(y) # 2
print(z) # 3
```

Possiamo avere anche multipli valori in una sola variabile:

```
x = y = z = 1
print(x) # 1
print(y) # 1
print(z) # 1
```

Possiamo fare anche l'unpacking di una collezione:

```
x = [1, 2, 3]
a, b, c = x
print(a) # 1
print(b) # 2
print(c) # 3
```

Così facendo abbiamo tutti i valori salvati in una variabile, e poi li possiamo assegnare a più variabili.

# 4 Tipi di dati

I tipi di dati in Python sono:

- **int**: numeri interi (es. 1, -5, 100)
- **float**: numeri decimali (es. 3.14, -0.001)
- **complex**: numeri complessi (es. 2+3j)
- **bool**: valori booleani (`True`, `False`)
- **str**: stringhe di testo (es. "ciao", 'Python')
- **list**: liste ordinate e modificabili (es. [1, 2, 3])
- **tuple**: tuple ordinate e immutabili (es. (1, 2, 3))
- **range**: intervalli di numeri (es. range(5))
- **dict**: dizionari (coppie chiave-valore) (es. {"a": 1, "b": 2})
- **set**: insiemi non ordinati e senza duplicati (es. {1, 2, 3})
- **frozenset**: insiemi immutabili (es. frozenset([1, 2, 3]))
- **bytes**: sequenze immutabili di byte (es. b"ciao")
- **bytearray**: sequenze mutabili di byte (es. bytearray(5))
- **memoryview**: viste su dati binari (es. memoryview(b"ciao"))
- **NoneType**: tipo speciale per il valore `None`

La funzione per scoprire il tipo di una variabile è `type()`:

```
x = 5
print(type(x)) # <class 'int'>

y = 3.14
print(type(y)) # <class 'float'>

z = "Ciao"
print(type(z)) # <class 'str'>

a = [1, 2, 3]
print(type(a)) # <class 'list'> equivalente a list()
#che sarebbe una lista

b = (1, 2, 3)
print(type(b)) # <class 'tuple'> equivalente a tuple()
#che sarebbe una tupla
```

```
c = {1, 2, 3}
print(type(c)) # <class 'set'>  equivalente a set()
#che sarebbe un insieme

d = {"a": 1, "b": 2}
print(type(d)) # <class 'dict'> equivalente a dict()
#che sarebbe un dizionario

x= range(5)
print(type(x)) # <class 'range'> equivalente a range()
#che sarebbe un range

x= True
print(type(x)) # <class 'bool'> equivalente a bool()
#che sarebbe un booleano
```

In python il tipo di dato viene assegnato in automatico, quindi non è necessario dichiararlo esplicitamente.

**ATTENZIONE QUINDI ALL'UTILIZZO DELLE VARIABILI E DEI LORO TIPI!**

# 5 Casting

Castare significa convertire un tipo di dato in un altro. Ad esempio, possiamo convertire un numero intero in una stringa:

```
x = 5
y = str(x) # ora y e' una stringa "5"
print(y) # "5"
print(type(y)) # <class 'str'>
```

Possiamo anche convertire una stringa in un numero intero:

```
x = "10"
y = int(x) # ora y e' un intero 10
print(y) # 10
print(type(y)) # <class 'int'>
```

Quando si fa? Lo facciamo quando vogliamo convertire un tipo di dato in un altro, ad esempio quando vogliamo sommare due numeri e uno è una stringa.

```
x = "10"
y = 5
z = int(x) + y # ora z e' un intero 15
print(z) # 15
print(type(z)) # <class 'int'>
```

# 6 Stringhe

Le stringhe in Python sono sequenze di caratteri racchiuse tra virgolette singole o doppie. Possiamo usare le virgolette singole o doppie per definire una stringa:

```
stringa1 = "Ciao"
stringa2 = 'Mondo'
```

Possiamo anche usare le triple virgolette per definire stringhe multilinea:

```
stringa_multilinea = """Questa e' una stringa
che si estende su piu' righe."""
print(stringa_multilinea)
# Output:
# Questa e' una stringa
# che si estende su piu' righe.
```

Le stringhe sono una collezione di caratteri, quindi possiamo accedere a un singolo carattere usando l'indice:

```
stringa = "Ciao"
print(stringa[0]) # C
print(stringa[1]) # i
print(stringa[2]) # a
print(stringa[3]) # o
print(stringa[-1]) # o
```

Anche lo spazio è considerato un carattere, quindi se scriviamo:

```
stringa = "Ciao Mondo"
print(stringa[9]) # spazio
```

La funzione per definire la lunghezza di una stringa è `len()`:

```
stringa = "Ciao"
print(len(stringa)) # 4
```

Se utilizziamo l'operatore : possiamo accedere a una parte della stringa, ad esempio:

```
stringa = "Ciao Mondo"
print(stringa[0:4]) # Ciao
print(stringa[5:10]) # Mondo
print(stringa[:4]) # Ciao
print(stringa[5:]) # Mondo
print(stringa[:]) # Ciao Mondo
print(stringa[-5:]) # Mondo
print(stringa[-5:-1]) # Mond
```

Possiamo modificare una stringa utilizzando dei metodi, ad esempio:

```
    stringa = "Ciao Mondo"
    print(stringa.upper()) # CIAO MONDO

    print(stringa.lower()) # ciao mondo

    print(stringa.title()) # Ciao Mondo

    print(stringa.capitalize()) # Ciao mondo

    print(stringa.strip()) # Ciao Mondo (rimuove gli spazi)

    print(stringa.replace("Ciao", "Salve")) # Salve Mondo

    print(stringa.split()) # ['Ciao', 'Mondo']
    #(divide la stringa in una lista)

    print(stringa.split(" ")) # ['Ciao', 'Mondo']
    #(divide la stringa in una lista)

    print(stringa.split("o")) # ['Ciao M', 'nd']
    #(divide la stringa in una lista)
```

La concatenazione delle stringhe avviene con l'operatore +:

```
    stringa1 = "Ciao"
    stringa2 = "Mondo"
    stringa3 = stringa1 + " " + stringa2
    print(stringa3) # Ciao Mondo
    print(stringa1 + stringa2) # CiaoMondo
    print(stringa1 + " " + stringa2) # Ciao Mondo
```

Ci viene in aiuto anche l'operatore `.format()` per formattare le stringhe:

```
nome = "Mario"
eta = 30
stringa ="Ciao, mi chiamo {} e ho {} anni.".format(nome, eta)
print(stringa) # Ciao, mi chiamo Mario e ho 30 anni.
stringa ="Ciao, mi chiamo {0} e ho {1} anni.".format(nome,eta)
print(stringa) # Ciao, mi chiamo Mario e ho 30 anni.
```

L'escape dei caratteri è un modo per inserire caratteri speciali in una stringa. Ad esempio, per inserire un apice singolo o doppio all'interno di una stringa, possiamo usare il backslash \:

```
stringa = 'Ciao, mi chiamo \'Mario\' e ho 30 anni.'
print(stringa) # Ciao, mi chiamo 'Mario' e ho 30 anni.

stringa = "Ciao, mi chiamo \"Mario\" e ho 30 anni."
print(stringa) # Ciao, mi chiamo "Mario" e ho 30 anni.
```

```
stringa ="Ciao, mi chiamo \"Mario\" e ho 30 anni.\nCome stai?"
print(stringa) # Ciao, mi chiamo "Mario" e ho 30 anni.
               # Come stai?
```

# 7 booleani

I booleani in Python sono un tipo di dato che può assumere solo due valori: `True` (vero) e `False` (falso). Sono molto usati nelle condizioni, nei cicli e nei confronti.

```
vero = True
falso = False
print(type(vero))   # <class 'bool'>
print(type(falso))  # <class 'bool'>
```

## Operatori di confronto

Gli operatori di confronto restituiscono un valore booleano:

```
print(5 > 3)    # True
print(2 == 4)   # False
print(7 != 8)   # True
print(10 >= 10) # True
print(3 < 1)    # False
```

## Operatori logici

Gli operatori logici permettono di combinare più condizioni:

- `and`: restituisce `True` solo se entrambe le condizioni sono vere
- `or`: restituisce `True` se almeno una delle condizioni è vera
- `not`: inverte il valore booleano

```
a = True
b = False
print(a and b) # False
print(a or b)  # True
print(not a)   # False
```

## Valori considerati falsi

In Python, alcuni valori sono considerati automaticamente `False` in un contesto booleano:

- `0`, `0.0`
- '' (stringa vuota)
- `[]` (lista vuota)
- `()` (tupla vuota)
- `{}` (dizionario vuoto)
- `set()` (insieme vuoto)
- `None`

Tutti gli altri valori sono considerati `True`.

```
print(bool(0))          # False
print(bool(""))         # False
print(bool([]))         # False
print(bool(None))       # False
print(bool(123))        # True
print(bool("Python"))   # True
```

### Uso dei booleani nelle condizioni

I booleani sono fondamentali nelle istruzioni `if`, `while` e in tutte le strutture di controllo:

```
x = 10
if x > 5:
    print("x e' maggiore di 5")
else:
    print("x non e' maggiore di 5")
```

**RICORDA:** In Python la prima lettera di `True` e `False` è maiuscola!

## 8 Operazioni Aritmetiche

Le operazioni aritmetiche in Python permettono di eseguire calcoli tra numeri. Python supporta tutti gli operatori aritmetici di base, oltre ad alcune operazioni avanzate.

### Operatori aritmetici di base

- `+` **(addizione)**: somma due valori.
- `-` **(sottrazione)**: sottrae il secondo valore dal primo.
- `*` **(moltiplicazione)**: moltiplica due valori.

- `/` **(divisione)**: divide il primo valore per il secondo (risultato `float`).
- `//` **(divisione intera)**: divide e restituisce solo la parte intera del risultato.
- `%` **(modulo)**: restituisce il resto della divisione.
- `**` **(potenza)**: eleva il primo valore alla potenza del secondo.

```
a = 10
b = 3
print(a + b)    # 13
print(a - b)    # 7
print(a * b)    # 30
print(a / b)    # 3.333...
print(a // b)   # 3
print(a % b)    # 1
print(a ** b)   # 1000
```

## Precedenza degli operatori

Gli operatori aritmetici seguono le regole di precedenza matematica:

1. Parentesi `()`
2. Potenza `**`
3. Moltiplicazione, divisione, divisione intera, modulo `* / // %`
4. Addizione e sottrazione `+ -`

Puoi usare le parentesi per forzare l'ordine di esecuzione:

```
print(2 + 3 * 4)       # 14
print((2 + 3) * 4)     # 20
print(2 ** 3 ** 2)     # 512 (equivale a 2 ** (3 ** 2))
```

## Assegnamento con operatore

Puoi combinare l'assegnamento con un'operazione aritmetica:

```
x = 5
x += 2    # x = x + 2 -> 7
x -= 1    # x = x - 1 -> 6
x *= 3    # x = x * 3 -> 18
x /= 2    # x = x / 2 -> 9.0
x //= 2   # x = x // 2 -> 4.0
x %= 3    # x = x % 3 -> 1.0
x **= 4   # x = x ** 4 -> 1.0
```

### Divisione tra interi e float

Se uno degli operandi è un `float`, il risultato sarà un `float`:

```
print(5 / 2)     # 2.5
print(5 // 2)    # 2
print(5.0 / 2)   # 2.5
print(5.0 // 2) # 2.0
```

### Operazioni con numeri negativi

Attenzione al comportamento di divisione intera e modulo con numeri negativi:

```
print(-7 // 3)   # -3
print(-7 % 3)    # 2
print(7 // -3)   # -3
print(7 % -3)    # -2
```

La divisione intera arrotonda sempre verso il basso (floor division).

### Funzioni matematiche utili

Python include la libreria `math` per operazioni avanzate:

```
import math
print(math.sqrt(16))     # 4.0 (radice quadrata)
print(math.pow(2, 5))    # 32.0 (potenza)
print(math.ceil(2.3))    # 3 (arrotonda per eccesso)
print(math.floor(2.7))   # 2 (arrotonda per difetto)
print(math.fabs(-5))     # 5.0 (valore assoluto)
print(math.factorial(5))# 120 (fattoriale)
print(math.pi)           # 3.141592...
print(math.e)            # 2.718281...
```

### Arrotondamenti

La funzione `round()` arrotonda un numero al numero di decimali desiderato:

```
print(round(3.14159, 2)) # 3.14
print(round(2.718, 0))   # 3.0
```

### Conversione tra tipi numerici

Puoi convertire tra `int`, `float` e `complex`:

```
x = 5
y = float(x)      # 5.0
z = int(3.7)      # 3
c = complex(2,3) # (2+3j)
```

### Numeri complessi

Python supporta i numeri complessi nativamente:

```
c1 = 2 + 3j
c2 = 1 - 4j
print(c1 + c2)    # (3-1j)
print(c1 * c2)    # (14-5j)
print(c1.real)    # 2.0 (parte reale)
print(c1.imag)    # 3.0 (parte immaginaria)
```

### Operatori unari

- `+x`: restituisce il valore di `x`
- `-x`: restituisce l'opposto di `x`

```
x = 5
print(+x)  # 5
print(-x)  # -5
```

### Funzione `abs()`

Restituisce il valore assoluto di un numero:

```
print(abs(-10))  # 10
print(abs(3.5))  # 3.5
```

### Esempi pratici

```
# Calcolo dell'area di un cerchio
raggio = 5
area = math.pi * (raggio ** 2)
print(area)  # 78.53981633974483

# Calcolo della media di tre numeri
a, b, c = 4, 7, 10
media = (a + b + c) / 3
print(media) # 7.0
```

**RICORDA:** In Python la divisione `/` restituisce sempre un `float`, anche se i numeri sono interi!

## 9 Condizioni con if, elif, else

Le istruzioni `if`, `elif` ed `else` permettono di eseguire blocchi di codice in base a condizioni booleane. Sono fondamentali per il controllo del flusso nei programmi Python.

### Struttura base

La sintassi di base è la seguente:

```
if condizione:
    # blocco eseguito se la condizione e' vera
elif altra\_condizione:
    # blocco eseguito se la prima condizione e' falsa e
    #questa e' vera
else:
    # blocco eseguito se tutte le condizioni precedenti
    #sono false
```

**Nota:** L'indentazione è obbligatoria in Python per delimitare i blocchi di codice.

### Esempio semplice

```
x = 10
if x > 0:
    print("x e' positivo")
elif x == 0:
    print("x e' zero")
else:
    print("x e' negativo")
```

### Più condizioni con `elif`

Puoi usare quanti `elif` vuoi:

```
voto = 85
if voto >= 90:
    print("Ottimo")
elif voto >= 70:
    print("Buono")
elif voto >= 60:
    print("Sufficiente")
else:
    print("Insufficiente")
```

### Condizioni annidate (nested if)

Puoi inserire un `if` dentro un altro `if`:

```
x = 5
if x > 0:
    print("Positivo")
    if x % 2 == 0:
        print("Pari")
    else:
```

```
        print("Dispari")
else:
    print("Non positivo")
```

## Operatori di confronto e logici

Le condizioni possono usare operatori di confronto (`==`, `!=`, `>`, `<`, `>=`, `<=`) e operatori logici (`and`, `or`, `not`):

```
eta = 20
patente = True
if eta >= 18 and patente:
    print("Puoi guidare")
else:
    print("Non puoi guidare")
```

## Condizioni su valori "falsy"

Qualsiasi valore che è "falsy" (come `0`, `""`, `[]`, `None`) è considerato `False` in una condizione:

```
nome = ""
if nome:
    print("Hai inserito un nome")
else:
    print("Nome mancante")
```

## Condizione su più righe

Puoi scrivere condizioni lunghe su più righe usando la barra inversa \:

```
x = 5
y = 10
if x > 0 and \
   y > 0:
    print("Entrambi positivi")
```

## Istruzione `pass`

Se vuoi lasciare vuoto un blocco `if`, usa `pass`:

```
if x > 0:
    pass  # da implementare in futuro
else:
    print("x non e' positivo")
```

### Istruzione `if` su una sola riga

Per condizioni semplici puoi scrivere tutto su una riga:

```
if x > 0: print("Positivo")
```

### Operatore ternario (if in una riga)

Per assegnare un valore in base a una condizione:

```
messaggio = "Maggiore di 10" if x > 10 else
"Minore o uguale a 10"
print(messaggio)
```

### Esempio pratico

```
numero = int(input("Inserisci un numero: "))
if numero % 2 == 0:
    print("Il numero e' pari")
else:
    print("Il numero e' dispari")
```

**RICORDA:** In Python l'indentazione è fondamentale per il corretto funzionamento delle condizioni!

## 10 Cicli While

Il ciclo `while` in Python permette di eseguire ripetutamente un blocco di codice finché una condizione booleana è vera. È uno degli strumenti fondamentali per la programmazione iterativa.

### Sintassi di base

La struttura di un ciclo `while` è la seguente:

```
while condizione:
    # blocco di istruzioni da ripetere
```

Il blocco di codice viene eseguito finché la `condizione` è vera (`True`). Appena la condizione diventa falsa (`False`), il ciclo termina e l'esecuzione prosegue dopo il ciclo.

### Esempio semplice

```
contatore = 0
while contatore < 5:
    print("Contatore:", contatore)
    contatore += 1
# Output:
# Contatore: 0
# Contatore: 1
# Contatore: 2
# Contatore: 3
# Contatore: 4
```

## Funzionamento dettagliato

- La condizione viene valutata **prima** di ogni iterazione.
- Se la condizione è `True`, il blocco viene eseguito.
- Alla fine del blocco, la condizione viene rivalutata.
- Se la condizione è `False`, il ciclo termina.

## Ciclo potenzialmente infinito

Se la condizione non diventa mai falsa, il ciclo `while` continua all'infinito (loop infinito):

```
while True:
    print("Questo ciclo non termina mai!")
```

Per interrompere manualmente un ciclo infinito, si può usare la combinazione di tasti `Ctrl+C` nel terminale.

## Uso della variabile di controllo

Spesso si utilizza una variabile di controllo che viene aggiornata all'interno del ciclo per evitare loop infiniti:

```
x = 10
while x > 0:
    print(x)
    x -= 1
```

## Istruzione `break`

L'istruzione `break` permette di uscire immediatamente dal ciclo, anche se la condizione è ancora vera:

```
while True:
    risposta = input("Scrivi 'esci' per terminare: ")
    if risposta == "esci":
        break
    print("Hai scritto:", risposta)
```

### Istruzione `continue`

L'istruzione `continue` interrompe l'iterazione corrente e passa subito alla valutazione della condizione per la prossima iterazione:

```
i = 0
while i < 10:
    i += 1
    if i % 2 == 0:
        continue
    print(i)  # stampa solo i numeri dispari da 1 a 9
```

### Istruzione `else` con while

Il ciclo `while` può avere una clausola `else`, che viene eseguita solo se il ciclo termina normalmente (cioè non tramite `break`):

```
x = 3
while x > 0:
    print(x)
    x -= 1
else:
    print("Ciclo terminato normalmente")
```

Se il ciclo viene interrotto con `break`, il blocco `else` non viene eseguito:

```
x = 3
while x > 0:
    print(x)
    if x == 2:
        break
    x -= 1
else:
    print("Questo non viene stampato")
```

### Esempio: input fino a condizione

```
password = ""
while password != "python":
    password = input("Inserisci la password: ")
print("Accesso consentito!")
```

### Esempio: conteggio con while

```
n = 1
while n <= 5:
    print(n)
    n += 1
```

### Esempio: somma dei numeri inseriti

```
somma = 0
while True:
    numero = input("Inserisci un numero (q per uscire): ")
    if numero == "q":
        break
    somma += int(numero)
print("Somma totale:", somma)
```

### Attenzione ai loop infiniti

Se dimentichi di aggiornare la variabile di controllo o la condizione non diventa mai falsa, il ciclo non termina mai:

```
x = 5
while x > 0:
    print(x)
    # x non viene mai modificato: ciclo infinito!
```

### Uso con valori "falsy"

Il ciclo `while` può essere usato con qualsiasi espressione che restituisce un valore booleano:

```
lista = [1, 2, 3]
while lista:
    elemento = lista.pop()
    print(elemento)
# Il ciclo termina quando la lista e' vuota (valore "falsy")
```

### Nidificazione di while

Puoi annidare cicli `while` all'interno di altri cicli `while`:

```
i = 1
while i <= 3:
    j = 1
    while j <= 2:
        print(f"i={i}, j={j}")
```

```
        j += 1
    i += 1
```

### Quando usare while invece di for

Usa `while` quando:

- Non conosci a priori il numero di iterazioni.
- Vuoi ripetere un blocco finché una condizione è vera.
- Gestisci input utente o condizioni che possono cambiare dinamicamente.

Se invece conosci il numero di iterazioni o devi scorrere una sequenza, è preferibile usare il ciclo `for`.

### Riepilogo

- Il ciclo `while` ripete un blocco finché la condizione è vera.
- Attenzione ai loop infiniti: aggiorna sempre la variabile di controllo.
- Puoi usare `break` per uscire dal ciclo e `continue` per saltare all'iterazione successiva.
- La clausola `else` viene eseguita solo se il ciclo termina normalmente.

**RICORDA:** Il ciclo `while` è molto potente, ma va usato con attenzione per evitare cicli infiniti!

## 11 Ciclo For

Il ciclo `for` in Python è uno strumento fondamentale per iterare su sequenze (come liste, tuple, stringhe, dizionari, insiemi) e su oggetti iterabili in generale. È molto versatile e permette di eseguire un blocco di codice per ogni elemento di una sequenza.

### Sintassi di base

La sintassi generale del ciclo `for` è:

```
for variabile in sequenza:
    # blocco di istruzioni
```

Ad ogni iterazione, la variabile assume il valore successivo della sequenza, fino a che la sequenza è esaurita.

### Esempio semplice

```
frutti = ["mela", "banana", "ciliegia"]
for frutto in frutti:
    print(frutto)
# Output:
# mela
# banana
# ciliegia
```

### Iterazione su stringhe

Le stringhe sono sequenze di caratteri, quindi puoi iterare su ogni carattere:

```
parola = "Python"
for lettera in parola:
    print(lettera)
# Output:
# P
# y
# t
# h
# o
# n
```

### Iterazione su tuple, set e dizionari

**Tuple:**

```
coordinate = (10, 20, 30)
for valore in coordinate:
    print(valore)
```

**Set:** (l'ordine non è garantito)

```
numeri = {1, 2, 3}
for n in numeri:
    print(n)
```

**Dizionari:**

```
diz = {"a": 1, "b": 2, "c": 3}
for chiave in diz:
    print(chiave, diz[chiave])
# oppure
for chiave, valore in diz.items():
    print(chiave, valore)
```

## La funzione `range()`

`range()` genera una sequenza di numeri interi, molto usata nei cicli `for`:

```
for i in range(5):
    print(i)
# Output: 0 1 2 3 4
```

`range(start, stop, step)`:

```
for i in range(2, 10, 2):
    print(i)
# Output: 2 4 6 8
```

## Iterazione inversa

Puoi iterare all'indietro usando un passo negativo:

```
for i in range(5, 0, -1):
    print(i)
# Output: 5 4 3 2 1
```

## Enumerare una sequenza

`enumerate()` restituisce coppie (indice, valore):

```
frutti = ["mela", "banana", "ciliegia"]
for indice, frutto in enumerate(frutti):
    print(indice, frutto)
# Output:
# 0 mela
# 1 banana
# 2 ciliegia
```

## Iterare su più sequenze contemporaneamente

`zip()` permette di iterare su più sequenze in parallelo:

```
nomi = ["Anna", "Luca", "Marta"]
eta = [25, 30, 22]
for nome, anni in zip(nomi, eta):
    print(nome, anni)
# Output:
# Anna 25
# Luca 30
# Marta 22
```

### Istruzione `break`

Interrompe il ciclo prima che la sequenza sia esaurita:

```
for n in range(10):
    if n == 5:
        break
    print(n)
# Output: 0 1 2 3 4
```

### Istruzione `continue`

Salta l'iterazione corrente e passa alla successiva:

```
for n in range(5):
    if n == 2:
        continue
    print(n)
# Output: 0 1 3 4
```

### Istruzione `else` con for

Il blocco `else` viene eseguito solo se il ciclo termina normalmente (non tramite `break`):

```
for n in range(3):
    print(n)
else:
    print("Ciclo terminato")
# Output:
# 0
# 1
# 2
# Ciclo terminato
```

Se il ciclo viene interrotto con `break`, il blocco `else` non viene eseguito:

```
for n in range(3):
    if n == 1:
        break
    print(n)
else:
    print("Ciclo terminato")
# Output:
# 0
```

### Ciclo for annidato (nested for)

Puoi inserire un ciclo `for` dentro un altro:

```
for i in range(1, 4):
    for j in range(1, 3):
        print(f"i={i}, j={j}")
```

## Modifica di una sequenza durante l'iterazione

Non è consigliato modificare una lista mentre la stai iterando. Se devi rimuovere elementi, crea una copia:

```
numeri = [1, 2, 3, 4]
for n in numeri[:]:  # copia della lista
    if n % 2 == 0:
        numeri.remove(n)
print(numeri)  # [1, 3]
```

## Comprensioni di lista (list comprehension)

Una forma compatta per creare nuove liste:

```
quadrati = [x**2 for x in range(5)]
print(quadrati)  # [0, 1, 4, 9, 16]
```

Puoi aggiungere condizioni:

```
pari = [x for x in range(10) if x % 2 == 0]
print(pari)  # [0, 2, 4, 6, 8]
```

## Iterazione su oggetti personalizzati

Un oggetto può essere iterato in un ciclo `for` se implementa il metodo speciale `__iter__()` e restituisce un iteratore (che implementa `__next__()`):

```
class Contatore:
    def __init__(self, limite):
        self.limite = limite
        self.valore = 0
    def __iter__(self):
        return self
    def __next__(self):
        if self.valore < self.limite:
            self.valore += 1
            return self.valore
        else:
            raise StopIteration

for n in Contatore(3):
    print(n)
# Output: 1 2 3
```

## Iterazione su file

Puoi iterare direttamente sulle righe di un file:

```
with open("file.txt") as f:
    for riga in f:
        print(riga.strip())
```

## Iterazione su dizionari: chiavi, valori, coppie

```
d = {"a": 1, "b": 2}
for chiave in d:
    print(chiave)
for valore in d.values():
    print(valore)
for chiave, valore in d.items():
    print(chiave, valore)
```

## Iterazione su oggetti non indicizzabili

Non tutte le sequenze sono indicizzabili (come i set), ma puoi comunque iterare su di esse con `for`.

## Riepilogo e consigli

- Il ciclo `for` è ideale per scorrere sequenze e oggetti iterabili.
- Usa `range()` per iterare su intervalli di numeri.
- Puoi usare `break`, `continue` ed `else` per controllare il flusso.
- Le comprensioni di lista sono una forma compatta e potente di ciclo `for`.
- Puoi annidare cicli `for` per lavorare su strutture complesse (matrici, tabelle, ecc.).
- Non modificare la sequenza su cui stai iterando, a meno che non sia una copia.
- Il ciclo `for` è preferibile al `while` quando conosci la sequenza o il numero di iterazioni.

**RICORDA:** In Python il ciclo `for` funziona su qualsiasi oggetto iterabile, non solo su liste!

# 12 Collezione di dati

Le collezioni di dati in Python sono strutture che permettono di raggruppare più valori in un'unica variabile. Sono fondamentali per gestire insiemi di dati, manipolarli, ordinarli, filtrarli e molto altro. Python offre diversi tipi di collezioni, ognuna con caratteristiche specifiche:

- **Liste (list)**
- **Tuple (tuple)**
- **Set (set)**
- **Dizionari (dict)**
- **Frozenset (frozenset)**

Vediamo in dettaglio ciascuna di queste collezioni.

## Liste (`list`)

Le liste sono collezioni ordinate, modificabili (mutabili) e possono contenere elementi duplicati di qualsiasi tipo.

**Creazione di una lista:**

```
lista = [1, 2, 3, "Python", True]
```

**Caratteristiche principali:**

- **Ordinata**: mantiene l'ordine di inserimento.
- **Indicizzabile**: puoi accedere agli elementi tramite indice (`lista[0]`).
- **Mutabile**: puoi modificare, aggiungere o rimuovere elementi.
- **Permette duplicati**: puoi avere più elementi uguali.

**Operazioni comuni sulle liste:**

```
frutti = ["mela", "banana", "ciliegia"]
print(frutti[0])          # mela
frutti[1] = "pera"        # Modifica elemento
frutti.append("kiwi")     # Aggiunge in fondo
frutti.insert(1, "arancia") # Inserisce in posizione 1
frutti.remove("mela")     # Rimuove il primo valore uguale
elemento = frutti.pop()   # Rimuove e restituisce
                          #l'ultimo elemento
lunghezza = len(frutti)   # Lunghezza della lista
```

**Slicing:**

```
numeri = [0, 1, 2, 3, 4, 5]
print(numeri[2:5])   # [2, 3, 4]
print(numeri[:3])    # [0, 1, 2]
print(numeri[::2])   # [0, 2, 4]
```

**Metodi utili:**

```
lista.count(x)     # Conta le occorrenze di x
lista.index(x)     # Restituisce l'indice della
                   #prima occorrenza di x
lista.sort()       # Ordina la lista (in place)
lista.reverse()    # Inverte l'ordine (in place)
lista.copy()       # Restituisce una copia superficiale
lista.clear()      # Svuota la lista
```

**Liste annidate:**

```
matrice = [[1, 2], [3, 4], [5, 6]]
print(matrice[1][0])  # 3
```

## Tuple (`tuple`)

Le tuple sono collezioni ordinate e immutabili (non modificabili dopo la creazione). Permettono elementi duplicati.

**Creazione di una tupla:**

```
tupla = (1, 2, 3, "Python")
```

**Caratteristiche principali:**

- **Ordinata** e **indicizzabile**
- **Immutabile**: non puoi aggiungere, rimuovere o modificare elementi
- **Permette duplicati**
- Più veloce delle liste e usata per dati costanti

**Operazioni sulle tuple:**

```
print(tupla[0])      # 1
print(len(tupla))    # 4
nuova = tupla + (4,) # Concatenazione
```

**Unpacking:**

```
a, b, c, d = tupla
```

**Tuple monoelemento:**

```
t = (5,)  # Attenzione alla virgola!
```

## Set (`set`)

I set sono collezioni non ordinate, mutabili e non indicizzate. Non permettono duplicati.

**Creazione di un set:**

```
insieme = {1, 2, 3, 3, 2}
print(insieme)  # {1, 2, 3}
```

**Caratteristiche principali:**

- **Non ordinato**: nessuna garanzia sull'ordine
- **Mutabile**: puoi aggiungere o rimuovere elementi
- **Nessun duplicato**
- **Non indicizzabile**

**Operazioni sui set:**

```
insieme.add(4)          # Aggiunge un elemento
insieme.remove(2)       # Rimuove un elemento
                        #(errore se non esiste)
insieme.discard(5)      # Rimuove se esiste,
                        #altrimenti non fa nulla
insieme.pop()           # Rimuove un elemento casuale
insieme.clear()         # Svuota il set
```

**Operazioni insiemistiche:**

```
a = {1, 2, 3}
b = {3, 4, 5}
print(a | b)   # Unione: {1, 2, 3, 4, 5}
print(a & b)   # Intersezione: {3}
print(a - b)   # Differenza: {1, 2}
print(a ^ b)   # Differenza simmetrica: {1, 2, 4, 5}
```

## Dizionari (`dict`)

I dizionari sono collezioni non ordinate (in Python 3.7+ mantengono l'ordine di inserimento), mutabili e indicizzate tramite chiavi univoche.

**Creazione di un dizionario:**

```
diz = {"nome": "Mario", "eta": 30}
```

**Caratteristiche principali:**

- **Coppie chiave-valore**
- **Chiavi univoche** (immutabili: stringhe, numeri, tuple)
- **Valori di qualsiasi tipo**

- **Mutabile**

**Operazioni sui dizionari:**

```
print(diz["nome"])          # Mario
diz["eta"] = 31             # Modifica valore
diz["citta"] = "Roma"       # Aggiunge nuova coppia
diz.pop("eta")              # Rimuove chiave e
                            #restituisce valore
valore = diz.get("email", "Non presente") # Accesso sicuro
chiavi = diz.keys()         # Tutte le chiavi
valori = diz.values()       # Tutti i valori
coppie = diz.items()        # Tutte le coppie (chiave, valore)
diz.clear()                 # Svuota il dizionario
```

**Iterazione su dizionari:**

```
for chiave in diz:
    print(chiave, diz[chiave])
for chiave, valore in diz.items():
    print(chiave, valore)
```

## Frozenset (`frozenset`)

Il `frozenset` è una variante immutabile del set. Una volta creato, non può essere modificato.

**Creazione:**

```
f = frozenset([1, 2, 3])
```

**Caratteristiche:**

- **Immutabile**
- **Nessun duplicato**
- Supporta operazioni insiemistiche come i set

## Conversione tra collezioni

Puoi convertire tra diversi tipi di collezioni:

```
lista = [1, 2, 3]
tupla = tuple(lista)
insieme = set(lista)
diz = dict([("a", 1), ("b", 2)])
```

## Comprensioni (Comprehensions)

Le comprensioni permettono di creare collezioni in modo compatto:

```
# Lista dei quadrati
quadrati = [x**2 for x in range(5)]
# Set delle lettere uniche in una parola
lettere = {c for c in "banana"}
# Dizionario di numeri e loro quadrati
d = {x: x**2 for x in range(3)}
```

## Collezioni annidate

Puoi avere collezioni dentro altre collezioni:

```
matrice = [[1, 2], [3, 4]]
diz = {"nomi": ["Anna", "Luca"], "eta": [20, 30]}
```

## Moduli avanzati: `collections`

Il modulo `collections` offre collezioni speciali:

- `namedtuple`: tuple con nomi ai campi
- `deque`: lista doppiamente terminata (più efficiente per inserimenti/rimozioni alle estremità)
- `Counter`: conteggio di elementi hashable
- `OrderedDict`: dizionario ordinato (in Python 3.7+ i dict standard sono già ordinati)
- `defaultdict`: dizionario con valore di default per chiavi mancanti
- `ChainMap`: unisce più dizionari in una sola vista

**Esempi:**

```
from collections import Counter, deque, namedtuple, defaultdict

# Counter
conta = Counter("banana")
print(conta)  # Counter({'a': 3, 'b': 1, 'n': 2})

# deque
dq = deque([1, 2, 3])
dq.appendleft(0)
dq.append(4)
dq.pop()
dq.popleft()
```

```
# namedtuple
Punto = namedtuple("Punto", ["x", "y"])
p = Punto(1, 2)
print(p.x, p.y)

# defaultdict
d = defaultdict(int)
d["a"] += 1
print(d["a"])  # 1
print(d["b"])  # 0 (valore di default)
```

## Mutabilità e immutabilità

- **Mutabili**: list, set, dict, deque
- **Immutabili**: tuple, frozenset, namedtuple

## Quando usare quale collezione

- **list**: sequenze ordinate e modificabili, accesso rapido per indice
- **tuple**: dati costanti, chiavi di dizionari, sicurezza da modifiche
- **set**: insiemi di elementi unici, operazioni insiemistiche
- **frozenset**: insiemi immutabili, chiavi di dizionari/set
- **dict**: associazioni chiave-valore, lookup rapido per chiave
- **deque**: code e pile efficienti
- **Counter**: conteggio frequenze
- **defaultdict**: dizionari con valori di default

## Esempi pratici

```
# Lista di nomi unici ordinati alfabeticamente
nomi = ["Anna", "Luca", "Anna", "Marco"]
nomi_unici = sorted(set(nomi))

# Dizionario da due liste
chiavi = ["a", "b", "c"]
valori = [1, 2, 3]
diz = dict(zip(chiavi, valori))

# Lista piatta da lista di liste
liste = [[1, 2], [3, 4], [5]]
piatta = [x for sub in liste for x in sub]
```

### Conclusioni

Le collezioni di dati sono strumenti essenziali in Python per organizzare, manipolare e analizzare dati. Scegli la collezione più adatta in base alle esigenze di ordinamento, mutabilità, unicità e tipo di accesso richiesto. La conoscenza approfondita di queste strutture è fondamentale per scrivere codice Python efficiente e leggibile.

## 13 Liste

Le **liste** in Python sono una delle strutture dati più versatili e utilizzate. Permettono di memorizzare una sequenza ordinata di elementi, che possono essere di qualsiasi tipo (numeri, stringhe, oggetti, altre liste, ecc.). Le liste sono **mutabili**, cioè possono essere modificate dopo la loro creazione.

### Creazione di una lista

Puoi creare una lista racchiudendo gli elementi tra parentesi quadre `[]` separati da virgole:

```
lista_vuota = []
numeri = [1, 2, 3, 4, 5]
mista = [1, "ciao", 3.14, True]
annidata = [1, [2, 3], 4]
```

### Accesso agli elementi

Gli elementi sono indicizzati a partire da 0:

```
frutti = ["mela", "banana", "ciliegia"]
print(frutti[0])   # mela
print(frutti[1])   # banana
print(frutti[-1])  # ciliegia (indice negativo:
                   #conta da destra)
```

### Slicing (fette di lista)

Puoi ottenere sotto-liste usando la notazione `lista[start:stop:step]`:

```
numeri = [0, 1, 2, 3, 4, 5, 6]
print(numeri[2:5])     # [2, 3, 4]
print(numeri[:3])      # [0, 1, 2]
print(numeri[::2])     # [0, 2, 4, 6]
print(numeri[::-1])    # [6, 5, 4, 3, 2, 1, 0]
                       #(lista invertita)
```

## Modifica degli elementi

Le liste sono mutabili:

```
frutti[1] = "pera"
print(frutti)  # ["mela", "pera", "ciliegia"]
```

## Aggiunta di elementi

- `append(x)`: aggiunge `x` in fondo alla lista
- `insert(i, x)`: inserisce `x` in posizione `i`
- `extend(iterabile)`: aggiunge tutti gli elementi di un iterabile

```
frutti.append("kiwi")
frutti.insert(1, "arancia")
frutti.extend(["uva", "melone"])
```

## Rimozione di elementi

- `remove(x)`: rimuove la prima occorrenza di `x`
- `pop([i])`: rimuove e restituisce l'elemento in posizione `i` (default: ultimo)
- `clear()`: svuota la lista
- `del lista[i]`: elimina l'elemento in posizione `i`

```
frutti.remove("pera")
ultimo = frutti.pop()
del frutti[0]
frutti.clear()
```

## Ricerca di elementi

- `in`: verifica se un elemento è presente
- `index(x, [start, end])`: restituisce l'indice della prima occorrenza di `x`
- `count(x)`: conta le occorrenze di `x`

```
if "banana" in frutti:
    print("Presente!")
pos = frutti.index("ciliegia")
n = frutti.count("mela")
```

## Ordinamento e inversione

- `sort()`: ordina la lista in place (modifica la lista)
- `sorted(lista)`: restituisce una nuova lista ordinata
- `reverse()`: inverte l'ordine degli elementi

```
numeri = [3, 1, 4, 2]
numeri.sort()
numeri.reverse()
ordinata = sorted(numeri)
```

## Copia di una lista

Attenzione: l'assegnamento crea un alias, non una copia!

```
a = [1, 2, 3]
b = a           # b e' un alias di a
c = a.copy()    # copia superficiale
d = list(a)     # altra copia superficiale
e = a[:]        # copia tramite slicing
```

## Liste annidate (matrici)

Le liste possono contenere altre liste:

```
matrice = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
print(matrice[1][2])  # 6
```

## Iterazione su una lista

```
for elemento in frutti:
    print(elemento)
for i, elemento in enumerate(frutti):
    print(i, elemento)
```

## Comprensioni di lista

Modo compatto per creare nuove liste:

```
quadrati = [x**2 for x in range(10)]
pari = [x for x in range(20) if x % 2 == 0]
```

## Funzioni e metodi utili

- `len(lista)`: lunghezza
- `min(lista)`, `max(lista)`, `sum(lista)`
- `any(lista)`, `all(lista)`
- `zip()`, `map()`, `filter()`

```
print(len(frutti))
print(min(numeri), max(numeri), sum(numeri))
```

## Concatenazione e ripetizione

```
a = [1, 2]
b = [3, 4]
c = a + b          # [1, 2, 3, 4]
d = a * 3          # [1, 2, 1, 2, 1, 2]
```

## Mutabilità e aliasing

Le liste sono mutabili: modificare una lista tramite un alias modifica anche l'originale.

```
a = [1, 2, 3]
b = a
b[0] = 99
print(a)  # [99, 2, 3]
```

## Copia profonda (deep copy)

Se la lista contiene altre liste (annidate), la copia superficiale non basta:

```
import copy
matrice = [[1, 2], [3, 4]]
copia_profonda = copy.deepcopy(matrice)
```

## Metodi principali delle liste

- `append(x)`
- `extend(iterabile)`
- `insert(i, x)`
- `remove(x)`
- `pop([i])`

- `clear()`
- `index(x, [start, end])`
- `count(x)`
- `sort(key=None, reverse=False)`
- `reverse()`
- `copy()`

## Liste di oggetti

Le liste possono contenere oggetti di qualsiasi tipo, inclusi oggetti personalizzati:

```
class Persona:
    def __init__(self, nome):
        self.nome = nome

persone = [Persona("Anna"), Persona("Luca")]
print(persone[0].nome)
```

## Liste come stack e queue

**Stack (pila):** usa `append()` e `pop()`

```
stack = []
stack.append(1)
stack.append(2)
x = stack.pop()  # 2
```

**Queue (coda):** per efficienza usa `collections.deque`, ma con le liste:

```
queue = [1, 2, 3]
x = queue.pop(0)  # 1 (inefficiente per liste lunghe)
```

## Conversione da e verso altri tipi

```
lista = list("ciao")  # ['c', 'i', 'a', 'o']
stringa = "".join(lista)  # 'ciao'
```

## Liste e funzioni

Le liste possono essere passate come argomenti e restituite dalle funzioni:

```
def somma_lista(l):
    return sum(l)

risultato = somma_lista([1, 2, 3])
```

### Liste e unpacking

```
a, b, c = [1, 2, 3]
x, *resto = [10, 20, 30, 40]  # x=10, resto=[20, 30, 40]
```

### Liste e generatori

Puoi convertire un generatore in lista:

```
gen = (x**2 for x in range(5))
lista = list(gen)
```

### Limitazioni delle liste

- Non sono thread-safe
- Non sono efficienti per inserimenti/rimozioni in testa (usa `deque`)
- Non sono tipizzate (possono contenere tipi diversi)

### Esempi pratici

```
# Rimuovere duplicati mantenendo l'ordine
lista = [1, 2, 2, 3, 1]
senza_duplicati = []
for x in lista:
    if x not in senza_duplicati:
        senza_duplicati.append(x)

# Flatten di una lista di liste
liste = [[1, 2], [3, 4], [5]]
piatta = [x for sub in liste for x in sub]
```

### Conclusioni

Le liste sono fondamentali in Python per la gestione di sequenze di dati. Sono flessibili, potenti e supportano numerose operazioni. La loro conoscenza approfondita è essenziale per programmare in Python in modo efficace.

## 14 Tuple

Le **tuple** in Python sono una delle strutture dati fondamentali. Sono simili alle liste, ma con una differenza chiave: **le tuple sono immutabili**, cioè una volta create non possono essere modificate (non puoi aggiungere, rimuovere o cambiare i loro elementi). Questa caratteristica le rende utili in molti contesti dove la sicurezza e l'integrità dei dati sono importanti.

## Creazione di una tupla

Puoi creare una tupla racchiudendo gli elementi tra parentesi tonde `()` separati da virgole:

```
tupla = (1, 2, 3)
vuota = ()
singolo = (5,)  # Attenzione alla virgola!
```

**Nota:** Per creare una tupla con un solo elemento, è obbligatorio mettere la virgola dopo l'elemento, altrimenti Python la interpreta come un'espressione tra parentesi.

Puoi anche creare tuple senza parentesi, separando gli elementi con la virgola:

```
tupla = 1, 2, 3
```

## Accesso agli elementi

Le tuple sono indicizzate e ordinate, quindi puoi accedere agli elementi tramite l'indice:

```
t = (10, 20, 30)
print(t[0])    # 10
print(t[-1])   # 30 (indice negativo: conta da destra)
```

## Slicing

Come le liste, puoi ottenere sotto-tuple tramite slicing:

```
t = (0, 1, 2, 3, 4)
print(t[1:4])    # (1, 2, 3)
print(t[:3])     # (0, 1, 2)
print(t[::-1])   # (4, 3, 2, 1, 0) (tupla invertita)
```

## Immutabilità

Le tuple non possono essere modificate dopo la creazione:

```
t = (1, 2, 3)
# t[0] = 10  # Errore: TypeError
# t.append(4)  # Errore: AttributeError
```

Tuttavia, se una tupla contiene oggetti mutabili (come liste), questi oggetti possono essere modificati:

```
t = (1, [2, 3], 4)
t[1][0] = 99
print(t)  # (1, [99, 3], 4)
```

## Quando usare le tuple

- Quando vuoi dati costanti che non devono essere modificati.
- Come chiavi nei dizionari (le tuple sono hashable se contengono solo oggetti immutabili).
- Come valori di ritorno multipli da una funzione.
- Per garantire l'integrità dei dati.
- Per efficienza: le tuple sono più leggere e veloci delle liste.

## Operazioni sulle tuple

- `len(tupla)`: lunghezza della tupla.
- `tupla + tupla2`: concatenazione.
- `tupla * n`: ripetizione.
- `in`, `not in`: verifica presenza elemento.
- `tupla.count(x)`: conta le occorrenze di x.
- `tupla.index(x)`: indice della prima occorrenza di x.

```
t = (1, 2, 3)
print(len(t))          # 3
print(t + (4, 5))      # (1, 2, 3, 4, 5)
print(t * 2)           # (1, 2, 3, 1, 2, 3)
print(2 in t)          # True
print(t.count(1))      # 1
print(t.index(3))      # 2
```

## Unpacking delle tuple

Puoi assegnare i valori di una tupla a più variabili in una sola riga:

```
t = (10, 20, 30)
a, b, c = t
print(a, b, c)  # 10 20 30
```

Puoi anche usare l'unpacking esteso:

```
t = (1, 2, 3, 4, 5)
a, *b, c = t
print(a)  # 1
print(b)  # [2, 3, 4]
print(c)  # 5
```

### Tuple come valori di ritorno multipli

Le funzioni possono restituire più valori usando una tupla:

```
def divmod(a, b):
    return a // b, a % b

q, r = divmod(17, 5)
print(q, r)  # 3 2
```

### Tuple annidate

Le tuple possono contenere altre tuple (o altri oggetti):

```
t = ((1, 2), (3, 4), (5, 6))
print(t[1][0])  # 3
```

### Tuple e dizionari

Le tuple possono essere usate come chiavi nei dizionari, purché siano composte solo da oggetti immutabili:

```
d = { (1, 2): "a", (3, 4): "b" }
print(d[(1, 2)])  # "a"
```

### Tuple e funzioni built-in

Molte funzioni Python accettano tuple come argomenti:

```
min((3, 1, 2))  # 1
max((3, 1, 2))  # 3
sum((1, 2, 3))  # 6
```

### Tuple e comprensioni

Non esistono "tuple comprehension", ma puoi creare una tupla da una list comprehension usando `tuple()`:

```
t = tuple(x**2 for x in range(5))  # (0, 1, 4, 9, 16)
```

### Tuple e metodi

Le tuple hanno solo due metodi: `count()` e `index()`.

```
t = (1, 2, 2, 3)
print(t.count(2))   # 2
print(t.index(3))   # 3
```

## Tuple e performance

Le tuple sono più leggere e veloci delle liste per:

- Creazione e accesso agli elementi.
- Iterazione.
- Uso come chiavi nei dizionari/set.

Questo perché Python può ottimizzare le tuple grazie alla loro immutabilità.

## Tuple e sicurezza

L'immutabilità delle tuple garantisce che i dati non vengano accidentalmente modificati, rendendole ideali per:

- Costanti.
- Dati di configurazione.
- Argomenti di funzioni che non devono essere alterati.

## Tuple e oggetti mutabili

Se una tupla contiene oggetti mutabili (come liste), la tupla resta immutabile, ma gli oggetti interni possono essere modificati:

```
t = ([1, 2], 3)
t[0].append(4)
print(t)  # ([1, 2, 4], 3)
```

## Tuple monoelemento

Per creare una tupla con un solo elemento, serve la virgola:

```
t = (5,)    # tupla
t2 = (5)    # non e' una tupla, e' un int
```

## Tuple vuote

La tupla vuota si scrive con le parentesi tonde senza elementi:

```
t = ()
```

## Conversione tra tuple e altri tipi

```
lista = [1, 2, 3]
t = tuple(lista)
l = list(t)
```

## Tuple e unpacking con funzioni

Puoi passare una tupla come argomenti a una funzione usando l'operatore `*`:

```
def somma(a, b, c):
    return a + b + c

t = (1, 2, 3)
print(somma(*t))  # 6
```

## Namedtuple

Il modulo `collections` offre le `namedtuple`, tuple con campi accessibili per nome:

```
from collections import namedtuple
Punto = namedtuple("Punto", ["x", "y"])
p = Punto(1, 2)
print(p.x, p.y)  # 1 2
```

Le `namedtuple` sono immutabili come le tuple normali.

## Tuple e hashabilità

Le tuple sono hashable se tutti i loro elementi sono hashable (cioè immutabili). Questo permette di usarle come chiavi nei dizionari e come elementi nei set.

## Differenze tra tuple e liste

- **Mutabilità**: le liste sono mutabili, le tuple no.
- **Metodi**: le liste hanno molti metodi, le tuple solo `count()` e `index()`.
- **Prestazioni**: le tuple sono più veloci e leggere.
- **Uso**: le tuple per dati costanti, le liste per dati variabili.

## Esempi pratici

```
# Scambio di variabili
a, b = 1, 2
a, b = b, a

# Iterazione su lista di tuple
dati = [("Anna", 25), ("Luca", 30)]
for nome, eta in dati:
    print(nome, eta)

# Uso come chiavi di dizionario
coordinate = {}
coordinate[(45.0, 9.0)] = "Milano"
```

### Conclusioni

Le tuple sono fondamentali in Python per rappresentare sequenze di dati costanti, garantire l'integrità delle informazioni e migliorare le prestazioni. La loro immutabilità le rende sicure e adatte a molti scenari, come chiavi di dizionari, valori di ritorno multipli e strutture dati complesse. La conoscenza approfondita delle tuple è essenziale per una programmazione Python efficace e sicura.

## 15 Set

I **set** in Python sono una struttura dati fondamentale per rappresentare insiemi di elementi unici, non ordinati e mutabili. Sono ispirati alla teoria degli insiemi della matematica e permettono di eseguire efficientemente operazioni insiemistiche come unione, intersezione, differenza e differenza simmetrica.

### Caratteristiche principali dei set

- **Non ordinati**: gli elementi non hanno un ordine definito e non sono accessibili tramite indice.
- **Elementi unici**: ogni elemento può comparire una sola volta.
- **Mutabili**: puoi aggiungere e rimuovere elementi dopo la creazione.
- **Non indicizzabili**: non puoi accedere agli elementi tramite indice o slicing.
- **Elementi hashable**: solo oggetti immutabili (hashable) possono essere inseriti in un set (es. numeri, stringhe, tuple di oggetti immutabili).

### Creazione di un set

```
# Set vuoto (attenzione: {} crea un dizionario!)
s = set()
# Set con elementi
numeri = {1, 2, 3, 4}
# Da lista, tuple, stringhe
s = set([1, 2, 2, 3])       # {1, 2, 3}
s = set((4, 5, 6, 4))       # {4, 5, 6}
s = set("banana")           # {'b', 'a', 'n'}
```

**Nota:** `{}` crea un dizionario vuoto, non un set vuoto!

### Elementi ammessi nei set

Gli elementi devono essere **immutabili** e **hashable**:

- Ammessi: int, float, str, tuple di oggetti immutabili, frozenset.
- Non ammessi: list, dict, set, oggetti mutabili.

```
s = {1, 2, (3, 4)}
# s = {[1, 2]}  # Errore: lista non hashable
```

## Operazioni fondamentali sui set

- `add(x)`: aggiunge l'elemento `x`
- `remove(x)`: rimuove `x` (errore se non esiste)
- `discard(x)`: rimuove `x` se esiste, altrimenti non fa nulla
- `pop()`: rimuove e restituisce un elemento arbitrario
- `clear()`: svuota il set
- `len(s)`: numero di elementi
- `in`, `not in`: verifica presenza elemento

```
s = {1, 2, 3}
s.add(4)
s.remove(2)
s.discard(10)  # Non errore
x = s.pop()
s.clear()
```

## Operazioni insiemistiche

- `|` oppure `union()`: unione
- `&` oppure `intersection()`: intersezione
- `-` oppure `difference()`: differenza
- `^` oppure `symmetric_difference()`: differenza simmetrica

```
a = {1, 2, 3}
b = {3, 4, 5}
print(a | b)   # {1, 2, 3, 4, 5}
print(a & b)   # {3}
print(a - b)   # {1, 2}
print(a ^ b)   # {1, 2, 4, 5}
```

**Nota:** Queste operazioni restituiscono un nuovo set, non modificano gli originali.

## Metodi in-place (modifica del set)

- `update(iterabile)`: unione in-place
- `intersection_update(iterabile)`: intersezione in-place
- `difference_update(iterabile)`: differenza in-place
- `symmetric_difference_update(iterabile)`: differenza simmetrica in-place

```
a = {1, 2, 3}
a.update([3, 4])  # a = {1, 2, 3, 4}
a.intersection_update({2, 3, 5})  # a = {2, 3}
```

## Confronto tra set

- `==`, `!=`: uguaglianza
- `<=`, `<`: sottoinsieme (subset)
- `>=`, `>`: sovrainsieme (superset)
- `isdisjoint(other)`: True se non hanno elementi in comune
- `issubset(other)`: True se è sottoinsieme
- `issuperset(other)`: True se è sovrainsieme

```
a = {1, 2}
b = {1, 2, 3}
print(a < b)  # True
print(b > a)  # True
print(a.issubset(b))  # True
print(b.issuperset(a))  # True
print(a.isdisjoint({3, 4}))  # True
```

## Iterazione su un set

```
s = {"a", "b", "c"}
for elemento in s:
    print(elemento)
```

L'ordine di iterazione è arbitrario e può cambiare tra esecuzioni.

## Set immutabili: frozenset

`frozenset` è la versione immutabile del set:

```
f = frozenset([1, 2, 3])
# f.add(4)  # Errore: AttributeError
```

Può essere usato come chiave di un dizionario o elemento di un altro set.

## Conversione tra set e altri tipi

```
lista = [1, 2, 2, 3]
s = set(lista)          # {1, 2, 3}
l = list(s)             # [1, 2, 3] (ordine arbitrario)
t = tuple(s)            # (1, 2, 3)
```

## Uso pratico dei set

- Rimuovere duplicati da una lista: `list(set(lista))`
- Test di appartenenza rapido: `x in s` è molto veloce
- Operazioni insiemistiche su grandi quantità di dati
- Filtrare elementi unici

## Esempi pratici

```
# Rimuovere duplicati mantenendo l'ordine
lista = [1, 2, 2, 3, 1]
visti = set()
senza_duplicati = []
for x in lista:
    if x not in visti:
        senza_duplicati.append(x)
        visti.add(x)

# Parole uniche in una frase
frase = "il gatto e il cane"
parole_uniche = set(frase.split())
# {'gatto', 'e', 'il', 'cane'}

# Elementi comuni tra due liste
l1 = [1, 2, 3]
l2 = [2, 3, 4]
comuni = set(l1) & set(l2)  # {2, 3}
```

## Limitazioni e attenzioni

- Gli elementi devono essere hashable (immutabili).
- L'ordine non è garantito.
- Non puoi accedere agli elementi tramite indice.
- Non puoi avere set di set (ma puoi avere set di frozenset).

## Performance

- Le operazioni di appartenenza (`in`) e aggiunta/rimozione sono molto veloci (O(1) in media).
- Le operazioni insiemistiche sono ottimizzate.

## Metodi principali dei set

- `add(x)`
- `remove(x)`
- `discard(x)`
- `pop()`
- `clear()`
- `union(*others)`
- `intersection(*others)`
- `difference(*others)`
- `symmetric_difference(other)`
- `update(*others)`
- `intersection_update(*others)`
- `difference_update(*others)`
- `symmetric_difference_update(other)`
- `issubset(other)`
- `issuperset(other)`
- `isdisjoint(other)`
- `copy()`

## Set comprehension

Come le list comprehension, puoi creare set in modo compatto:

```
quadrati = {x**2 for x in range(5)}  # {0, 1, 4, 9, 16}
pari = {x for x in range(10) if x % 2 == 0}
```

## Set e funzioni built-in

- `len(s)`: numero di elementi
- `min(s)`, `max(s)`
- `sum(s)`
- `all(s)`, `any(s)`
- `sorted(s)`: restituisce una lista ordinata

```
s = {3, 1, 2}
print(sorted(s))  # [1, 2, 3]
```

### Set annidati

Non puoi avere set di set (perché i set sono mutabili e non hashable), ma puoi avere set di frozenset:

```
a = frozenset([1, 2])
b = frozenset([3, 4])
s = {a, b}
```

### Differenze tra set, frozenset e dict

- **set**: mutabile, non ordinato, elementi unici, non indicizzabile.
- **frozenset**: come set, ma immutabile e hashable.
- **dict**: collezione di coppie chiave-valore, chiavi uniche e hashable.

### Quando usare i set

- Quando serve garantire l'unicità degli elementi.
- Per operazioni insiemistiche (unione, intersezione, ecc.).
- Per test di appartenenza rapidi.
- Per rimuovere duplicati da sequenze.

### Conclusioni

I set sono strumenti potenti e versatili per la gestione di insiemi di dati unici e per operazioni insiemistiche efficienti. La loro conoscenza è fondamentale per scrivere codice Python efficace, pulito e performante, soprattutto quando si lavora con grandi quantità di dati o si devono garantire proprietà di unicità.

## 16 Dizionari

I **dizionari** (`dict`) in Python sono una delle strutture dati più potenti e versatili. Permettono di memorizzare coppie **chiave-valore**, dove ogni chiave è univoca e associata a un valore. Sono noti anche come *mappe*, *hashmap* o *associative array* in altri linguaggi.

### Caratteristiche principali dei dizionari

- **Collezione di coppie chiave-valore**: ogni elemento è formato da una chiave e un valore associato.
- **Chiavi univoche**: ogni chiave può comparire una sola volta.

- **Chiavi hashable**: le chiavi devono essere oggetti immutabili (stringhe, numeri, tuple di oggetti immutabili, ecc.).
- **Valori di qualsiasi tipo**: i valori possono essere di qualsiasi tipo (anche altri dizionari).
- **Mutabili**: puoi aggiungere, modificare o rimuovere coppie dopo la creazione.
- **Ordinati**: da Python 3.7+ mantengono l'ordine di inserimento (prima erano non ordinati).
- **Accesso rapido**: l'accesso ai valori tramite chiave è molto veloce (O(1) in media).

## Creazione di un dizionario

```
# Dizionario vuoto
d = {}

# Dizionario con elementi
d = {"nome": "Mario", "eta": 30, "citta": "Roma"}

# Usando la funzione dict()
d = dict(nome="Anna", eta=25)

# Da lista di tuple/coppie
d = dict([("a", 1), ("b", 2)])

# Da zip di due liste
chiavi = ["x", "y"]
valori = [10, 20]
d = dict(zip(chiavi, valori))
```

## Accesso ai valori

```
d = {"a": 1, "b": 2}
print(d["a"])          # 1

# Accesso sicuro con get()
print(d.get("c"))      # None
print(d.get("c", 0))   # 0 (valore di default)
```

**Nota:** Se accedi a una chiave inesistente con `d["chiave"]`, ottieni un `KeyError`. Usa `get()` per evitare errori.

## Aggiunta e modifica di elementi

```
d["nuova"] = 123          # Aggiunge nuova coppia
d["a"] = 99               # Modifica valore esistente
```

### Rimozione di elementi

```
d.pop("a")                # Rimuove e restituisce
                          #il valore associato a "a"
del d["b"]                # Rimuove la coppia con chiave "b"
d.popitem()               # Rimuove e restituisce l'ultima
                          #coppia (da Python 3.7+)
d.clear()                 # Svuota il dizionario
```

### Verifica presenza di una chiave

```
if "nome" in d:
    print("Presente!")
```

### Iterazione su dizionari

```
# Solo chiavi
for chiave in d:
    print(chiave)

# Chiavi e valori
for chiave, valore in d.items():
    print(chiave, valore)

# Solo valori
for valore in d.values():
    print(valore)
```

### Metodi principali dei dizionari

- `d.keys()`
- `d.values()`
- `d.items()`
- `d.get(k, default)`
- `d.pop(k, default)`
- `d.popitem()`
- `d.clear()`
- `d.update(other)`
- `d.setdefault(k, default)`
- `d.copy()`

## Comprensione dei dizionari (dict comprehension)

Modo compatto per creare dizionari:

```
quadrati = {x: x**2 for x in range(5)}
# {0:0, 1:1, 2:4, 3:9, 4:16}
pari = {x: "pari" for x in range(10) if x % 2 == 0}
```

## Dizionari annidati

I valori possono essere altri dizionari:

```
studenti = {
    "Anna": {"eta": 20, "corso": "Informatica"},
    "Luca": {"eta": 22, "corso": "Fisica"}
}
print(studenti["Anna"]["corso"])  # Informatica
```

## Copia di dizionari

```
a = {"x": 1}
b = a               # alias, non copia!
c = a.copy()        # copia superficiale
import copy
d = copy.deepcopy(a)  # copia profonda (per dizionari annidati)
```

## Mutabilità e aliasing

Modificare un dizionario tramite un alias modifica anche l'originale.

## Chiavi ammesse

Le chiavi devono essere **immutabili** e **hashable**:

- Ammesse: stringhe, numeri, tuple di oggetti immutabili, frozenset.
- Non ammesse: liste, set, dizionari, oggetti mutabili.

```
d = {(1, 2): "a"}   # OK
# d = {[1, 2]: "b"} # Errore: lista non hashable
```

## Valori ammissibili

I valori possono essere di qualsiasi tipo, anche mutabili o altri dizionari.

## Ordinamento dei dizionari

Da Python 3.7+ i dizionari mantengono l'ordine di inserimento. Per ordinare un dizionario per chiave o valore:

```
d = {"b": 2, "a": 1, "c": 3}
ordinato = dict(sorted(d.items()))  # Ordina per chiave
ordinato_val = dict(sorted(d.items(), key=lambda x: x[1]))
# Per valore
```

## Unione e aggiornamento di dizionari

```
d1 = {"a": 1, "b": 2}
d2 = {"b": 3, "c": 4}
d1.update(d2)  # d1 = {"a": 1, "b": 3, "c": 4}

# Da Python 3.9+
d3 = d1 | d2   # Unione (nuovo dict)
```

## Funzioni built-in utili

- `len(d)`: numero di coppie
- `list(d)`: lista delle chiavi
- `sorted(d)`: lista delle chiavi ordinate
- `min(d)`, `max(d)`: chiave minima/massima
- `sum(d.values())`: somma dei valori (se numerici)

## Uso pratico dei dizionari

- Rappresentare dati strutturati (es. record, oggetti, JSON)
- Contare frequenze (`Counter`)
- Lookup rapido di valori tramite chiave
- Rappresentare grafi, tabelle di configurazione, mapping

## Moduli avanzati: collections.defaultdict e Counter

```
from collections import defaultdict, Counter

# defaultdict: valore di default per chiavi mancanti
d = defaultdict(int)
d["a"] += 1  # d["a"] = 1
```

```
# Counter: conteggio frequenze
c = Counter("banana")
print(c)  # Counter({'a': 3, 'b': 1, 'n': 2})
```

## Metodi avanzati

- `setdefault(k, default)`: restituisce il valore di k, se non esiste lo crea con default.
- `fromkeys(seq, value=None)`: crea un nuovo dict con chiavi da seq e valore uguale per tutte.

```
d = dict.fromkeys(["a", "b", "c"], 0)  #{'a': 0,'b': 0,'c': 0}
```

## Dizionari e JSON

I dizionari sono la struttura dati Python più simile agli oggetti JSON. Puoi convertire facilmente tra dict e JSON:

```
import json
d = {"nome": "Anna", "eta": 20}
s = json.dumps(d)           # dict -> stringa JSON
d2 = json.loads(s)          # stringa JSON -> dict
```

## Dizionari e unpacking

Da Python 3.5+ puoi unire dizionari con l'unpacking:

```
d1 = {"a": 1}
d2 = {"b": 2}
d3 = {**d1, **d2}  # {'a': 1, 'b': 2}
```

## Dizionari e funzioni

Puoi passare un dizionario come argomenti a una funzione usando **:

```
def f(x, y):
    return x + y

d = {"x": 1, "y": 2}
print(f(**d))  # 3
```

## Dizionari e oggetti personalizzati

Le chiavi possono essere oggetti personalizzati, purché siano hashable (implementino `__hash__` e `__eq__`).

### Limitazioni e attenzioni

- Le chiavi devono essere hashable (immutabili).
- L'ordine è garantito solo da Python 3.7+.
- Non puoi avere chiavi duplicate.
- L'accesso tramite chiave inesistente genera `KeyError` (usa `get()` o `defaultdict`).

### Performance

- L'accesso, inserimento e rimozione tramite chiave sono molto veloci (O(1) in media).
- L'iterazione su chiavi, valori o coppie è efficiente.

### Esempi pratici

```
# Conta frequenze di parole in una frase
frase = "il gatto e il cane"
conta = {}
for parola in frase.split():
    conta[parola] = conta.get(parola, 0) + 1

# Invertire chiavi e valori
d = {"a": 1, "b": 2}
inverso = {v: k for k, v in d.items()}

# Dizionario da due liste
chiavi = ["x", "y"]
valori = [10, 20]
d = dict(zip(chiavi, valori))
```

### Conclusioni

I dizionari sono fondamentali in Python per rappresentare dati strutturati, eseguire lookup rapidi, contare frequenze, gestire configurazioni e molto altro. La loro conoscenza approfondita è essenziale per scrivere codice Python efficace, leggibile e performante.

## 17 Funzioni

Le **funzioni** in Python sono blocchi di codice riutilizzabili che eseguono un compito specifico. Permettono di organizzare il codice, evitare ripetizioni, migliorare la leggibilità e facilitare la manutenzione. Le funzioni possono ricevere dati in ingresso (argomenti), restituire valori in uscita (valori di ritorno) e possono essere definite sia dall'utente sia essere già presenti nel linguaggio (funzioni built-in).

## Definizione di una funzione

Per definire una funzione si usa la parola chiave `def`, seguita dal nome della funzione, parentesi tonde (che possono contenere parametri) e i due punti. Il corpo della funzione va indentato.

```
def saluta():
    print("Ciao!")
```

## Chiamata di una funzione

Per eseguire una funzione basta scrivere il suo nome seguito da parentesi:

```
saluta()  # Output: Ciao!
```

## Parametri e argomenti

Le funzioni possono accettare uno o più parametri (variabili che ricevono i valori passati dall'esterno):

```
def somma(a, b):
    return a + b

risultato = somma(3, 5)  # 8
```

**Parametri** sono le variabili nella definizione della funzione. **Argomenti** sono i valori passati quando la funzione viene chiamata.

## Valore di ritorno (`return`)

La parola chiave `return` serve per restituire un valore dalla funzione:

```
def quadrato(x):
    return x * x

print(quadrato(4))  # 16
```

Se non viene specificato `return`, la funzione restituisce `None`.

## Funzioni senza parametri e senza valore di ritorno

```
def stampa_benvenuto():
    print("Benvenuto!")

stampa_benvenuto()
```

## Parametri opzionali (default)

Puoi assegnare un valore di default ai parametri:

```
def saluta(nome="Mondo"):
    print(f"Ciao, {nome}!")

saluta()          # Ciao, Mondo!
saluta("Anna")    # Ciao, Anna!
```

## Argomenti posizionali e nominati

Gli argomenti possono essere passati per posizione o per nome:

```
def descrivi_persona(nome, eta):
    print(f"{nome} ha {eta} anni.")

descrivi_persona("Luca", 30)           # Posizionali
descrivi_persona(eta=25, nome="Anna")  # Nominati
```

## Argomenti variabili: `*args` e `**kwargs`

`*args` permette di passare un numero variabile di argomenti posizionali (come tupla):

```
def somma_tutti(*numeri):
    return sum(numeri)

print(somma_tutti(1, 2, 3))  # 6
```

`**kwargs` permette di passare un numero variabile di argomenti nominati (come dizionario):

```
def stampa_info(**info):
    for chiave, valore in info.items():
        print(f"{chiave}: {valore}")

stampa_info(nome="Anna", eta=22)
# nome: Anna
# eta: 22
```

## Ordine dei parametri

L'ordine corretto nella definizione è:

```
def funzione(parametri_posizionali, parametri_default, *args, parametri_keyw
    pass
```

## Unpacking di argomenti

Puoi "spacchettare" una lista/tupla o un dizionario come argomenti:

```
def f(a, b, c):
    print(a, b, c)

t = (1, 2, 3)
f(*t)  # 1 2 3

d = {"a": 10, "b": 20, "c": 30}
f(**d)  # 10 20 30
```

## Funzioni come oggetti di prima classe

Le funzioni in Python sono oggetti: possono essere assegnate a variabili, passate come argomenti, restituite da altre funzioni.

```
def saluta():
    print("Ciao!")

f = saluta
f()  # Ciao!
```

## Funzioni annidate (nested functions)

Puoi definire una funzione dentro un'altra:

```
def esterna():
    def interna():
        print("Interna")
    interna()

esterna()  # Interna
```

## Scope delle variabili (ambito)

Le variabili definite all'interno di una funzione sono locali e non visibili all'esterno. Quelle definite fuori sono globali.

```
x = 10  # globale

def funzione():
    y = 5  # locale
    print(x, y)

funzione()  # 10 5
# print(y)  # Errore: y non e' definita fuori dalla funzione
```

### Parole chiave `global` e `nonlocal`

`global` permette di modificare una variabile globale dall'interno di una funzione:

```
x = 0

def incrementa():
    global x
    x += 1

incrementa()
print(x)  # 1
```

`nonlocal` permette di modificare una variabile non locale (ma non globale) in una funzione annidata:

```
def esterna():
    x = 10
    def interna():
        nonlocal x
        x += 1
    interna()
    print(x)  # 11

esterna()
```

### Documentazione delle funzioni (docstring)

Puoi documentare una funzione con una stringa tra triple virgolette subito dopo la definizione:

```
def somma(a, b):
    """Restituisce la somma di due numeri."""
    return a + b

print(somma.__doc__)  # Restituisce la somma di due numeri.
```

### Annotazioni di tipo (type hints)

Puoi specificare i tipi attesi per parametri e valore di ritorno (non obbligatorio):

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

### Funzioni anonime (lambda)

Le `lambda` sono funzioni senza nome, usate per operazioni semplici:

```python
doppio = lambda x: x * 2
print(doppio(5))  # 10

# Usate spesso con map, filter, sorted, ecc.
numeri = [1, 2, 3]
quadrati = list(map(lambda x: x**2, numeri))  # [1, 4, 9]
```

## Funzioni built-in

Python offre molte funzioni già pronte, come `len()`, `sum()`, `max()`, `min()`, `sorted()`, `print()`, `type()`, `range()`, ecc.

```python
print(len([1, 2, 3]))  # 3
print(sorted([3, 1, 2]))  # [1, 2, 3]
```

## Funzioni ricorsive

Una funzione può chiamare sé stessa (ricorsione):

```python
def fattoriale(n):
    if n == 0:
        return 1
    else:
        return n * fattoriale(n - 1)

print(fattoriale(5))  # 120
```

## Funzioni generatori (`yield`)

Le funzioni che usano `yield` restituiscono un generatore, cioè una sequenza di valori calcolati "al volo":

```python
def conta_fino_a(n):
    for i in range(1, n+1):
        yield i

for x in conta_fino_a(3):
    print(x)
# 1
# 2
# 3
```

## Funzioni come parametri e valori di ritorno

```python
def applica(funzione, valore):
    return funzione(valore)

def triplica(x):
    return x * 3

print(applica(triplica, 4))  # 12
```

## Decoratori

I decoratori sono funzioni che modificano il comportamento di altre funzioni:

```python
def decoratore(f):
    def wrapper(*args, **kwargs):
        print("Prima")
        risultato = f(*args, **kwargs)
        print("Dopo")
        return risultato
    return wrapper

@decoratore
def saluta():
    print("Ciao!")

saluta()
# Prima
# Ciao!
# Dopo
```

## Funzioni built-in avanzate: map, filter, reduce, zip, enumerate

```python
# map: applica una funzione a tutti gli elementi
numeri = [1, 2, 3]
doppio = list(map(lambda x: x*2, numeri))  # [2, 4, 6]

# filter: filtra elementi secondo una condizione
pari = list(filter(lambda x: x % 2 == 0, numeri))  # [2]

# reduce: riduce una sequenza a un singolo valore
from functools import reduce
somma = reduce(lambda x, y: x + y, numeri)  # 6

# zip: combina piu' sequenze
a = [1, 2]
b = ['a', 'b']
z = list(zip(a, b))  # [(1, 'a'), (2, 'b')]
```

```python
# enumerate: restituisce coppie (indice, valore)
for i, val in enumerate(['x', 'y']):
    print(i, val)
```

## Funzioni variadiche e keyword-only

**Variadiche:** accettano un numero arbitrario di argomenti (`*args`, `**kwargs`).
**Keyword-only:** parametri che possono essere passati solo per nome (dopo `*`).

```python
def f(a, b, *, c=0):
    print(a, b, c)

f(1, 2, c=3)  # OK
# f(1, 2, 3)  # Errore
```

## Funzioni e scope LEGB

Python cerca le variabili in questo ordine:

- **L**ocal: variabili locali alla funzione
- **E**nclosing: scope delle funzioni esterne (per funzioni annidate)
- **G**lobal: variabili globali del modulo
- **B**uilt-in: nomi predefiniti di Python

## Esempi pratici

```python
# Funzione che restituisce piu' valori
def min_max(lista):
    return min(lista), max(lista)

mn, mx = min_max([1, 5, 3])
print(mn, mx)  # 1 5

# Funzione che modifica una lista (mutabilita')
def aggiungi_elemento(l, x):
    l.append(x)

a = [1, 2]
aggiungi_elemento(a, 3)
print(a)  # [1, 2, 3]
```

## Conclusioni

Le funzioni sono fondamentali per strutturare, riutilizzare e organizzare il codice Python. Permettono di scrivere programmi modulari, leggibili e manutenibili. La conoscenza approfondita delle funzioni (parametri, scope, decoratori, generatori, lambda, ecc.) è essenziale per una programmazione Python efficace e professionale.

Le **classi** e gli **oggetti** sono i concetti fondamentali della programmazione orientata agli oggetti (OOP, Object-Oriented Programming) in Python. Permettono di modellare dati e comportamenti in modo strutturato, riutilizzabile e modulare.

## Cos'è una classe?

Una **classe** è un "modello" o "prototipo" che definisce le proprietà (attributi) e i comportamenti (metodi) che gli oggetti di quel tipo avranno. Puoi pensare a una classe come a un "progetto" o "stampo" per creare oggetti.

## Cos'è un oggetto?

Un **oggetto** è un'istanza concreta di una classe. Ogni oggetto ha i propri dati (attributi) e può eseguire azioni (metodi) definite dalla classe.

## Definizione di una classe

Per definire una classe si usa la parola chiave `class`:

```
class Persona:
    pass  # classe vuota
```

## Costruttore: `__init__`

Il metodo speciale `__init__` è il costruttore: viene chiamato automaticamente quando si crea un nuovo oggetto. Serve per inizializzare gli attributi dell'oggetto.

```
class Persona:
    def __init__(self, nome, eta):
        self.nome = nome
        self.eta = eta
```

`self` è il riferimento all'istanza corrente (obbligatorio come primo parametro nei metodi di istanza).

## Creazione di oggetti (istanze)

```
p1 = Persona("Anna", 25)
p2 = Persona("Luca", 30)
print(p1.nome, p1.eta)  # Anna 25
```

## Attributi di istanza e di classe

- **Attributi di istanza**: specifici per ogni oggetto (`self.nome`)
- **Attributi di classe**: condivisi da tutte le istanze

```
class Persona:
    specie = "Homo sapiens"  # attributo di classe
    def __init__(self, nome):
        self.nome = nome

print(Persona.specie)  # Homo sapiens
p = Persona("Anna")
print(p.specie)        # Homo sapiens
```

## Metodi di istanza, di classe e statici

- **Metodi di istanza**: ricevono `self` (l'oggetto)
- **Metodi di classe**: ricevono `cls` (la classe), decorati con `@classmethod`
- **Metodi statici**: non ricevono né `self` né `cls`, decorati con `@staticmethod`

```
class Persona:
    popolazione = 0

    def __init__(self, nome):
        self.nome = nome
        Persona.popolazione += 1

    def saluta(self):
        print(f"Ciao, sono {self.nome}")

    @classmethod
    def quante(cls):
        print(f"Popolazione: {cls.popolazione}")

    @staticmethod
    def specie():
        return "Homo sapiens"
```

## Incapsulamento e visibilità

In Python non esistono modificatori di accesso veri e propri (come `private`, `protected`, `public`), ma si usano convenzioni:

- `_attributo`: convenzione per "protetto" (uso interno)
- `__attributo`: name mangling, difficile da accedere dall'esterno

```
class Persona:
    def __init__(self, nome):
        self._nome = nome       # protetto
        self.__segreto = 123  # privato (name mangling)
```

## Proprietà (property) e getter/setter

Per controllare l'accesso agli attributi si usano le `property`:

```
class Persona:
    def __init__(self, nome):
        self._nome = nome

    @property
    def nome(self):
        return self._nome

    @nome.setter
    def nome(self, valore):
        if not valore:
            raise ValueError("Nome vuoto!")
        self._nome = valore
```

## Metodi speciali (dunder methods)

I metodi che iniziano e finiscono con doppio underscore (`__`) sono "speciali" e permettono di personalizzare il comportamento degli oggetti:

- `__init__`: costruttore
- `__str__`: rappresentazione stringa (print)
- `__repr__`: rappresentazione tecnica
- `__eq__`, `__lt__`, `__gt__`: operatori di confronto
- `__len__`, `__getitem__`, `__setitem__`: comportamento come sequenze
- `__call__`: oggetto chiamabile come funzione
- `__del__`: distruttore (chiamato alla cancellazione)

```
class Persona:
    def __init__(self, nome):
        self.nome = nome

    def __str__(self):
        return f"Persona: {self.nome}"
```

```
    def __eq__(self, other):
        return self.nome == other.nome
```

## Ereditarietà

Una classe può ereditare da un'altra (classe base o superclasse):

```
class Studente(Persona):
    def __init__(self, nome, matricola):
        super().__init__(nome)
        self.matricola = matricola
```

`super()` permette di chiamare metodi della superclasse.

## Polimorfismo

Oggetti di classi diverse possono essere trattati allo stesso modo se implementano gli stessi metodi:

```
class Animale:
    def parla(self):
        pass

class Cane(Animale):
    def parla(self):
        print("Bau!")

class Gatto(Animale):
    def parla(self):
        print("Miao!")

def fai_parlare(animale):
    animale.parla()

fai_parlare(Cane())
fai_parlare(Gatto())
```

## Ereditarietà multipla

Una classe può ereditare da più classi:

```
class A:
    pass

class B:
    pass

class C(A, B):
```

```
    pass
```

Python risolve i conflitti con il Method Resolution Order (MRO).

## Classi astratte e interfacce

Python non ha interfacce come Java, ma puoi definire classi astratte con il modulo `abc`:

```
from abc import ABC, abstractmethod

class Animale(ABC):
    @abstractmethod
    def parla(self):
        pass
```

Non puoi istanziare una classe astratta finché non implementi tutti i metodi astratti.

## Composizione

Un oggetto può contenere altri oggetti come attributi (relazione "has-a"):

```
class Motore:
    pass

class Auto:
    def __init__(self):
        self.motore = Motore()
```

## Duck typing

In Python conta ciò che un oggetto *sa fare*, non la sua classe. Se un oggetto implementa i metodi richiesti, può essere usato (principio "se cammina come un'anatra...").

## Classi annidate

Puoi definire una classe dentro un'altra:

```
class Esterna:
    class Interna:
        pass
```

## Metaclassi

Le metaclassi sono "classi di classi", permettono di personalizzare la creazione delle classi (avanzato):

```
class MiaMeta(type):
    def __new__(cls, nome, basi, dct):
        return super().__new__(cls, nome, basi, dct)

class MiaClasse(metaclass=MiaMeta):
    pass
```

## Slot

Per ottimizzare la memoria puoi usare `__slots__`:

```
class Persona:
    __slots__ = ['nome', 'eta']
    def __init__(self, nome, eta):
        self.nome = nome
        self.eta = eta
```

## Operatori di confronto e ordinamento

Puoi personalizzare gli operatori (==, <, >, ecc.) implementando i metodi speciali:

```
class Punto:
    def __init__(self, x):
        self.x = x
    def __eq__(self, other):
        return self.x == other.x
    def __lt__(self, other):
        return self.x < other.x
```

## Oggetti come funzioni: `__call__`

Se implementi `__call__`, puoi "chiamare" l'oggetto come una funzione:

```
class Moltiplicatore:
    def __init__(self, n):
        self.n = n
    def __call__(self, x):
        return self.n * x

m = Moltiplicatore(3)
print(m(5))  # 15
```

## Oggetti iterabili e iteratori

Per rendere un oggetto iterabile, implementa `__iter__` e `__next__`:

```
class Contatore:
    def __init__(self, limite):
        self.limite = limite
        self.valore = 0
    def __iter__(self):
        return self
    def __next__(self):
        if self.valore < self.limite:
            self.valore += 1
            return self.valore
        else:
            raise StopIteration
```

## Oggetti context manager (__enter__, __exit__)

Per usare un oggetto con `with`, implementa:

```
class FileManager:
    def __enter__(self):
        print("Apro risorsa")
        return self
    def __exit__(self, exc_type, exc_val, exc_tb):
        print("Chiudo risorsa")

with FileManager():
    print("Dentro il blocco")
```

## Documentazione delle classi

Puoi documentare una classe con una docstring:

```
class Persona:
    """Rappresenta una persona con nome ed eta'."""
    pass

print(Persona.__doc__)
```

## Classi e moduli

Le classi possono essere definite in moduli e importate:

```
# file persona.py
class Persona:
    pass

# file main.py
from persona import Persona
```

### Esempio completo

```
class Persona:
    popolazione = 0

    def __init__(self, nome, eta):
        self.nome = nome
        self.eta = eta
        Persona.popolazione += 1

    def saluta(self):
        print(f"Ciao, sono {self.nome} e ho {self.eta} anni.")

    @classmethod
    def quante(cls):
        print(f"Popolazione: {cls.popolazione}")

    @staticmethod
    def specie():
        return "Homo sapiens"

p1 = Persona("Anna", 25)
p2 = Persona("Luca", 30)
p1.saluta()
Persona.quante()
print(Persona.specie())
```

### Conclusioni

Le classi e gli oggetti sono il cuore della programmazione orientata agli oggetti in Python. Permettono di modellare dati e comportamenti, favoriscono il riuso del codice, l'incapsulamento, l'astrazione e la modularità. La conoscenza approfondita di classi, oggetti, ereditarietà, polimorfismo, metodi speciali e property è fondamentale per scrivere codice Python robusto, scalabile e professionale.

## 18 Ereditarietà

L'**ereditarietà** è uno dei concetti fondamentali della programmazione orientata agli oggetti (OOP) e permette di creare una nuova classe (**classe derivata** o **sottoclasse**) che eredita attributi e metodi da un'altra classe (**classe base** o **superclasse**). In Python, l'ereditarietà consente di riutilizzare codice, estendere funzionalità e modellare relazioni gerarchiche tra oggetti.

### Cos'è l'ereditarietà?

L'ereditarietà permette a una classe di acquisire automaticamente tutte le proprietà (attributi) e i comportamenti (metodi) di un'altra classe. La classe derivata può:

- Usare attributi e metodi della classe base senza ridefinirli.
- Ridefinire (override) metodi della classe base per modificarne il comportamento.
- Aggiungere nuovi attributi e metodi propri.

## Sintassi dell'ereditarietà in Python

Per dichiarare che una classe eredita da un'altra, si specifica la superclasse tra parentesi dopo il nome della sottoclasse:

```
class Animale:
    def parla(self):
        print("L'animale fa un verso")

class Cane(Animale):
    pass

c = Cane()
c.parla()  # Output: L'animale fa un verso
```

## Override dei metodi

La sottoclasse può ridefinire un metodo della superclasse per modificarne il comportamento:

```
class Cane(Animale):
    def parla(self):
        print("Bau!")
```

Ora, chiamando `parla()` su un oggetto `Cane`, verrà eseguito il metodo ridefinito.

## Aggiunta di nuovi attributi e metodi

La sottoclasse può avere attributi e metodi aggiuntivi:

```
class Gatto(Animale):
    def miagola(self):
        print("Miao!")
```

## Costruttore e super()

Se la sottoclasse ha un proprio costruttore (`__init__`), può richiamare quello della superclasse con `super()`:

```
class Animale:
    def __init__(self, nome):
        self.nome = nome
```

```
class Cane(Animale):
    def __init__(self, nome, razza):
        super().__init__(nome)
        self.razza = razza
```

`super()` è fondamentale per inizializzare correttamente la parte ereditata.

## Ereditarietà multipla

Python supporta l'ereditarietà multipla: una classe può ereditare da più classi base.

```
class A:
    def metodo_a(self):
        print("A")

class B:
    def metodo_b(self):
        print("B")

class C(A, B):
    pass

c = C()
c.metodo_a()  # Output: A
c.metodo_b()  # Output: B
```

### Method Resolution Order (MRO)

Quando una classe eredita da più classi, Python segue un ordine preciso per risolvere i metodi: il **Method Resolution Order** (MRO). Puoi visualizzare l'MRO con:

```
print(C.mro())
# oppure
help(C)
```

L'MRO segue l'algoritmo C3 linearization.

## Ereditarietà e tipi di metodi

- **Metodi di istanza**: ereditati e possono essere ridefiniti.
- **Metodi di classe** (`@classmethod`): ereditati e possono essere ridefiniti.
- **Metodi statici** (`@staticmethod`): ereditati e possono essere ridefiniti.

## Ereditarietà e attributi

- **Attributi di istanza**: ereditati se inizializzati dalla superclasse (tramite `super().__init__()`).

- **Attributi di classe**: condivisi tra tutte le istanze e sottoclassi, ma possono essere ridefiniti nella sottoclasse.

## Ereditarietà e funzioni built-in

- `isinstance(obj, Classe)`: verifica se `obj` è istanza di `Classe` o di una sua sottoclasse.
- `issubclass(Sottoclasse, Superclasse)`: verifica se una classe è sottoclasse di un'altra.

```
print(isinstance(c, Animale))  # True
print(issubclass(Cane, Animale))  # True
```

## Ereditarietà e polimorfismo

Grazie all'ereditarietà, puoi trattare oggetti di sottoclassi come oggetti della superclasse:

```
def fai_parlare(animale):
    animale.parla()

fai_parlare(Cane())
fai_parlare(Gatto())
```

Questo è il **polimorfismo**: oggetti diversi rispondono allo stesso metodo in modo diverso.

## Classi astratte e metodi astratti

Per forzare le sottoclassi a implementare certi metodi, si usano le **classi astratte** con il modulo `abc`:

```
from abc import ABC, abstractmethod

class Animale(ABC):
    @abstractmethod
    def parla(self):
        pass

class Cane(Animale):
    def parla(self):
        print("Bau!")
```

Non puoi istanziare una classe astratta finché non implementi tutti i metodi astratti.

## Ereditarietà e metodi speciali

I metodi speciali (`__str__`, `__eq__`, ecc.) sono ereditati e possono essere ridefiniti per personalizzare il comportamento degli oggetti.

## Ereditarietà e composizione

L'ereditarietà modella una relazione "è un" (*is-a*), mentre la composizione modella una relazione "ha un" (*has-a*). Usa l'ereditarietà solo quando la relazione logica è appropriata.

## Ereditarietà e protected/private

In Python, gli attributi preceduti da `_` sono considerati "protetti" (convenzione), mentre quelli con `__` sono soggetti a name mangling e meno accessibili dalle sottoclassi.

## Ereditarietà e costruttori multipli

Se una sottoclasse non chiama `super().__init__()`, gli attributi della superclasse non vengono inizializzati.

## Ereditarietà multipla e diamond problem

Se più classi base hanno un antenato comune, si può creare il **diamond problem**. Python lo risolve con l'MRO (C3 linearization).

```
class A:
    def metodo(self):
        print("A")

class B(A):
    def metodo(self):
        print("B")

class C(A):
    def metodo(self):
        print("C")

class D(B, C):
    pass

d = D()
d.metodo()  # Output: B (segue l'MRO)
print(D.mro())  # [D, B, C, A, object]
```

## Ereditarietà e built-in

Puoi ereditare anche dalle classi built-in di Python (come `list`, `dict`, ecc.) per estenderne il comportamento:

```
class MiaLista(list):
    def somma(self):
        return sum(self)
```

```
ml = MiaLista([1, 2, 3])
print(ml.somma())  # 6
```

## Ereditarietà e metaclassi

Le metaclassi permettono di personalizzare il comportamento dell'ereditarietà a livello di creazione delle classi (avanzato).

## Ereditarietà e documentazione

Le sottoclassi ereditano anche la docstring della superclasse, a meno che non venga ridefinita.

## Ereditarietà e performance

L'ereditarietà aggiunge un piccolo overhead nella risoluzione dei metodi, ma in generale è molto efficiente grazie all'MRO.

## Ereditarietà e `__slots__`

Se la superclasse definisce `__slots__`, la sottoclasse deve ridefinirli se vuole aggiungere nuovi attributi.

## Ereditarietà e isinstance/issubclass

`isinstance(obj, Classe)` e `issubclass(Sottoclasse, Superclasse)` funzionano anche con ereditarietà multipla e classi astratte.

## Ereditarietà e override di attributi di classe

Se una sottoclasse ridefinisce un attributo di classe, questo nasconde quello della superclasse solo nella sottoclasse.

## Ereditarietà e metodi privati

I metodi con doppio underscore (`__metodo`) sono soggetti a name mangling e non sono direttamente accessibili dalle sottoclassi.

## Ereditarietà e super() avanzato

`super()` può essere usato anche per chiamare metodi diversi da `__init__`, e in presenza di ereditarietà multipla segue l'MRO.

## Ereditarietà e cooperative multiple inheritance

In presenza di ereditarietà multipla, è buona pratica usare sempre `super()` per garantire che tutti i costruttori delle classi base vengano chiamati secondo l'MRO.

```
class Base:
    def __init__(self):
        print("Base")

class A(Base):
    def __init__(self):
        super().__init__()
        print("A")

class B(Base):
    def __init__(self):
        super().__init__()
        print("B")

class C(A, B):
    def __init__(self):
        super().__init__()
        print("C")

c = C()
# Output:
# Base
# B
# A
# C
```

## Ereditarietà e classi built-in ABC

Molte classi built-in di Python (come `collections.abc.Sequence`, `Mapping`, ecc.) sono pensate per essere ereditate e forniscono metodi astratti da implementare.

## Ereditarietà e introspezione

Puoi esplorare la gerarchia di ereditarietà con:

```
print(obj.__class__.__bases__)  # tuple delle superclassi dirette
print(obj.__class__.mro())      # MRO completo
```

## Ereditarietà e best practice

- Usa l'ereditarietà solo quando esiste una relazione logica "is-a".
- Preferisci la composizione quando la relazione è "has-a".

- Usa sempre `super()` in presenza di ereditarietà multipla.
- Documenta chiaramente la gerarchia e le responsabilità delle classi.
- Evita gerarchie troppo profonde o complesse.

### Esempio pratico completo

```
from abc import ABC, abstractmethod

class Veicolo(ABC):
    def __init__(self, marca):
        self.marca = marca

    @abstractmethod
    def muovi(self):
        pass

class Auto(Veicolo):
    def muovi(self):
        print(f"L'auto {self.marca} si muove su strada.")

class Barca(Veicolo):
    def muovi(self):
        print(f"La barca {self.marca} naviga sull'acqua.")

class Anfibio(Auto, Barca):
    def muovi(self):
        print(f"L'anfibio {self.marca} si muove ovunque!")

v = Anfibio("Amphicar")
v.muovi()  # L'anfibio Amphicar si muove ovunque!
print(Anfibio.mro())
```

### Conclusioni

L'ereditarietà è uno strumento potente per la riusabilità, l'estensione e la modellazione delle relazioni tra oggetti. Permette di scrivere codice più modulare, mantenibile e scalabile. Tuttavia, va usata con attenzione per evitare gerarchie troppo complesse e per mantenere il codice leggibile e facilmente manutenibile. La comprensione approfondita dell'ereditarietà, dell'MRO, del polimorfismo e delle classi astratte è fondamentale per una programmazione Python avanzata e professionale.

## 19 Scope delle variabili

Lo **scope** (ambito) di una variabile in Python indica la porzione di codice in cui quella variabile è visibile e accessibile. Comprendere lo scope è fondamentale per evitare errori, bug e comportamenti inattesi nei programmi.

## Tipi di scope in Python (LEGB Rule)

Python segue la regola LEGB per la risoluzione dei nomi delle variabili:

- **L**ocal: variabili definite all'interno di una funzione (scope locale).
- **E**nclosing: scope delle funzioni esterne (per funzioni annidate).
- **G**lobal: variabili definite a livello di modulo (fuori da funzioni/classi).
- **B**uilt-in: nomi predefiniti di Python (come `len`, `print`, ecc.).

## Scope locale

Una variabile definita dentro una funzione è locale a quella funzione e non è accessibile dall'esterno:

```
def funzione():
    x = 10  # locale
    print(x)

funzione()  # 10
# print(x)  # Errore: x non e' definita fuori dalla funzione
```

## Scope globale

Una variabile definita fuori da tutte le funzioni è globale e accessibile ovunque nel modulo:

```
x = 5  # globale

def stampa():
    print(x)  # accede alla variabile globale

stampa()  # 5
```

## Scope enclosing (funzioni annidate)

Quando una funzione è definita dentro un'altra, le variabili della funzione esterna sono visibili a quella interna (enclosing scope):

```
def esterna():
    x = 20
    def interna():
        print(x)  # accede a x della funzione esterna
    interna()

esterna()  # 20
```

### Scope built-in

Python ha uno scope speciale per i nomi predefiniti:

```
print(len([1, 2, 3]))  # print e len sono built-in
```

### Regola di risoluzione dei nomi (LEGB)

Quando accedi a una variabile, Python cerca nell'ordine:

1. Scope locale (funzione corrente)
2. Scope enclosing (funzioni esterne, se presenti)
3. Scope globale (modulo)
4. Scope built-in (nomi predefiniti)

### Modifica delle variabili globali: `global`

Per modificare una variabile globale dentro una funzione, usa la parola chiave `global`:

```
x = 0

def incrementa():
    global x
    x += 1

incrementa()
print(x)  # 1
```

Senza `global`, Python crea una nuova variabile locale con lo stesso nome.

### Modifica delle variabili non locali: `nonlocal`

Per modificare una variabile dello scope enclosing (non globale), usa `nonlocal`:

```
def esterna():
    x = 10
    def interna():
        nonlocal x
        x += 1
    interna()
    print(x)  # 11

esterna()
```

`nonlocal` funziona solo nelle funzioni annidate.

## Shadowing (oscuramento) delle variabili

Se una variabile locale ha lo stesso nome di una globale, la locale "oscura" la globale nello scope della funzione:

```
x = 100

def f():
    x = 5  # locale, oscura la globale
    print(x)

f()      # 5
print(x) # 100
```

## Scope nelle classi

Gli attributi delle classi e delle istanze hanno uno scope diverso:

```
x = 1

class A:
    x = 2  # attributo di classe

    def metodo(self):
        x = 3  # locale al metodo
        print(x, self.x)

a = A()
a.metodo()  # 3 2
```

## Scope nei cicli e nelle comprensioni

Le variabili dei cicli `for` e delle comprensioni di lista/set/dict sono visibili nello scope in cui sono definite:

```
for i in range(3):
    pass
print(i)  # 2 (l'ultimo valore)

# Dal Python 3, le variabili nelle comprensioni
#hanno uno scope separato:
x = 10
lst = [x for x in range(5)]
print(x)  # 10
```

## Scope nei moduli

Ogni modulo ha il proprio scope globale. Variabili globali in un modulo non sono visibili in altri moduli a meno che non vengano importate.

## Scope e funzioni lambda

Le lambda seguono le stesse regole di scope delle funzioni normali:

```
def f():
    x = 10
    return lambda y: x + y

g = f()
print(g(5))  # 15
```

## Esempio completo: LEGB

```
x = "globale"

def esterna():
    x = "enclosing"
    def interna():
        x = "locale"
        print(x)  # locale
    interna()
    print(x)      # enclosing

esterna()
print(x)          # globale
```

## Best practice e attenzioni

- Evita di usare troppe variabili globali: rendono il codice difficile da mantenere.
- Usa `global` e `nonlocal` solo quando necessario.
- Preferisci passare variabili come argomenti alle funzioni.
- Fai attenzione allo shadowing: nomi uguali in scope diversi possono causare confusione.
- Ricorda che le variabili mutabili (liste, dict) possono essere modificate anche senza `global`, ma solo i loro contenuti, non il riferimento.

## Esempi pratici

```
# Modifica di una lista globale senza global
lista = []

def aggiungi(x):
    lista.append(x)  # OK: modifica il contenuto
```

```
aggiungi(5)
print(lista)  # [5]

# Ma per riassegnare serve global
def resetta():
    global lista
    lista = []

resetta()
print(lista)  # []
```

### Conclusioni

Lo scope delle variabili è fondamentale per la corretta gestione dei dati e per evitare errori di visibilità e modifica indesiderata. Comprendere la regola LEGB, l'uso di `global` e `nonlocal`, e le differenze tra scope locale, globale, enclosing e built-in è essenziale per scrivere codice Python robusto, leggibile e privo di bug.

## 20 Moduli

I **moduli** in Python sono file che contengono definizioni di variabili, funzioni, classi e istruzioni eseguibili. Permettono di organizzare il codice in unità riutilizzabili, facilitando la manutenzione, la leggibilità e la condivisione. I moduli sono la base della modularità in Python e consentono di suddividere programmi complessi in parti più semplici.

### Cos'è un modulo?

Un modulo è semplicemente un file con estensione `.py` che può essere importato in altri file Python. Il nome del modulo corrisponde al nome del file (senza estensione).

```
# file: mio_modulo.py
def saluta(nome):
    print(f"Ciao, {nome}!")
```

### Importazione di moduli

Per usare un modulo, si utilizza la parola chiave `import`:

```
import mio_modulo
mio_modulo.saluta("Anna")
```

Puoi importare solo alcune parti di un modulo:

```
from mio_modulo import saluta
saluta("Luca")
```

Puoi anche rinominare il modulo o la funzione importata:

```python
import mio_modulo as mm
mm.saluta("Marco")

from mio_modulo import saluta as s
s("Giulia")
```

## Dove cerca Python i moduli?

Quando importi un modulo, Python cerca il file corrispondente nei percorsi elencati in `sys.path`:

- La directory corrente (dove viene eseguito lo script)
- Le directory specificate nella variabile d'ambiente `PYTHONPATH`
- Le directory di installazione standard di Python (site-packages, ecc.)

Puoi vedere i percorsi con:

```python
import sys
print(sys.path)
```

## Tipi di moduli

- **Moduli standard**: forniti con Python (es. `math`, `os`, `sys`, `random`, ecc.)
- **Moduli di terze parti**: installabili tramite `pip` (es. `numpy`, `pandas`, `requests`, ecc.)
- **Moduli personalizzati**: creati dall'utente (file `.py`)
- **Moduli built-in**: scritti in C e integrati nell'interprete (es. `sys`, `time`)

## Importazione di tutto il contenuto

```python
from mio_modulo import *
saluta("Mario")
```

**Nota:** Non è consigliato usare `*` perché può causare conflitti di nomi e rende il codice meno leggibile.

## Il file `__init__.py` e i package

Un **package** è una cartella che contiene un insieme di moduli e un file speciale `__init__.py` (può essere vuoto). Questo file indica a Python che la cartella è un package.

```
mio_package/
    __init__.py
    modulo1.py
    modulo2.py
```

Importazione:

```
import mio_package.modulo1
from mio_package import modulo2
```

## Struttura dei package annidati

I package possono essere annidati:

```
mio_package/
    __init__.py
    subpackage/
        __init__.py
        modulo3.py
```

Importazione:

```
from mio_package.subpackage import modulo3
```

## Import relativi e assoluti

- **Import assoluto**: dal root del progetto/package
- **Import relativo**: usa il punto `.` per riferirsi al package corrente o ai parent

```
# In mio_package/modulo1.py
from . import modulo2      # import relativo
from .. import altro_modulo # dal package padre
```

## Il modulo `__main__`

Quando un modulo viene eseguito direttamente, la variabile speciale `__name__` vale `"__main__"`. Quando viene importato, vale il nome del modulo.

```
# file: mio_modulo.py
def saluta():
    print("Ciao!")

if __name__ == "__main__":
    saluta()
```

Questo permette di scrivere codice che viene eseguito solo se il file è eseguito direttamente, non quando è importato.

## Variabili speciali dei moduli

- `__name__`: nome del modulo
- `__file__`: percorso del file del modulo
- `__package__`: nome del package
- `__doc__`: docstring del modulo
- `__path__`: solo per package, lista dei percorsi

## Ricaricare un modulo

Se modifichi un modulo durante una sessione interattiva, puoi ricaricarlo con:

```
import importlib
import mio_modulo
importlib.reload(mio_modulo)
```

## Moduli compilati e cache (`__pycache__`)

Quando importi un modulo, Python lo compila in bytecode (`.pyc`) e lo salva nella cartella `__pycache__` per velocizzare i successivi import.

## Moduli built-in

Alcuni moduli sono integrati nell'interprete e non hanno un file `.py` (es. `sys`, `math`, `time`). Puoi vedere la lista con:

```
import sys
print(sys.builtin_module_names)
```

## Installazione di moduli di terze parti

Usa `pip` per installare moduli esterni:

```
pip install nome_modulo
```

## Documentazione dei moduli

Puoi documentare un modulo con una docstring all'inizio del file:

```
"""Questo modulo contiene funzioni di esempio."""
def saluta():
    pass
```

## Esportare solo alcune parti: __all__

Nel modulo puoi definire la lista `__all__` per specificare cosa viene importato con `from modulo import *`:

```
__all__ = ["saluta", "addio"]
```

## Organizzazione di un progetto Python

Un progetto Python ben strutturato usa moduli e package per separare logica, test, configurazione, ecc.

```
progetto/
    main.py
    modulo1.py
    utils/
        __init__.py
        helper.py
    tests/
        test_modulo1.py
```

## Moduli e namespace

Ogni modulo ha il proprio namespace: variabili, funzioni e classi definite in un modulo non sono visibili in altri moduli a meno che non vengano importate.

## Moduli e import ciclici

Se due moduli si importano a vicenda, si crea un *import ciclico* che può causare errori. È buona pratica evitare dipendenze circolari.

## Moduli e script eseguibili

Un modulo può essere sia importato che eseguito come script. Usa il controllo su `__name__` per distinguere i due casi.

## Moduli e variabili globali

Le variabili definite a livello di modulo sono globali solo all'interno di quel modulo.

## Moduli e sys.modules

Python mantiene una cache dei moduli già importati in `sys.modules`. Se importi più volte lo stesso modulo, viene eseguito solo la prima volta.

## Moduli e introspezione

Puoi esplorare il contenuto di un modulo con `dir()`:

```
import math
print(dir(math))
```

## Moduli e import dinamico

Puoi importare moduli dinamicamente usando `importlib`:

```
import importlib
modulo = importlib.import_module("math")
print(modulo.sqrt(16))
```

## Moduli e distribuzione

Per distribuire i tuoi moduli o package, puoi creare un pacchetto installabile (`setup.py`, `pyproject.toml`) e pubblicarlo su PyPI.

## Moduli e virtual environment

Per isolare le dipendenze dei moduli di terze parti, usa ambienti virtuali (`venv`, `virtualenv`).

## Esempio pratico di modulo

```
# file: calcoli.py
"""Modulo di esempio per operazioni matematiche."""

def somma(a, b):
    """Restituisce la somma di due numeri."""
    return a + b

def moltiplica(a, b):
    """Restituisce il prodotto di due numeri."""
    return a * b

if __name__ == "__main__":
    print(somma(2, 3))
    print(moltiplica(4, 5))
```

## Esempio pratico di package

```
mio_package/
    __init__.py
    aritmetica.py
    geometria.py
```

```
# aritmetica.py
def somma(a, b):
    return a + b

# geometria.py
def area_rettangolo(base, altezza):
    return base * altezza

# __init__.py
from .aritmetica import somma
from .geometria import area_rettangolo
```

```
# main.py
from mio_package import somma, area_rettangolo
print(somma(1, 2))
print(area_rettangolo(3, 4))
```

### Conclusioni

I moduli sono fondamentali per la scrittura di codice Python organizzato, riutilizzabile e scalabile. Permettono di suddividere il programma in parti logiche, facilitano la collaborazione e la manutenzione, e sono la base per la creazione di librerie e applicazioni complesse. La conoscenza approfondita dei moduli, dei package, delle regole di importazione e delle best practice di organizzazione del codice è essenziale per ogni programmatore Python.

## 21 Datetime

La gestione di date e orari in Python è affidata principalmente al modulo `datetime`, che fa parte della libreria standard. Questo modulo permette di lavorare con date, orari, intervalli temporali, fusi orari e molto altro. Di seguito una panoramica completa delle funzionalità principali.

### Importazione del modulo

```
import datetime
from datetime import date, time, datetime, timedelta, timezone
```

### Classi principali di `datetime`

- `date`: rappresenta una data (anno, mese, giorno)
- `time`: rappresenta un orario (ora, minuti, secondi, microsecondi)
- `datetime`: rappresenta una data e un orario

- `timedelta`: rappresenta una differenza tra due date/ore (intervallo di tempo)
- `timezone`: rappresenta un fuso orario

### Classe `date`

```
d = datetime.date(2024, 6, 1)
print(d)  # 2024-06-01

oggi = datetime.date.today()
print(oggi)  # data odierna

# Attributi
print(d.year, d.month, d.day)

# Metodi
d2 = d.replace(year=2025)
print(d2)
```

### Classe `time`

```
t = datetime.time(14, 30, 15, 123456)
print(t)  # 14:30:15.123456

# Attributi
print(t.hour, t.minute, t.second, t.microsecond)
```

### Classe `datetime`

```
dt = datetime.datetime(2024, 6, 1, 14, 30, 0)
print(dt)  # 2024-06-01 14:30:00

adesso = datetime.datetime.now()
print(adesso)  # data e ora attuali

utcnow = datetime.datetime.utcnow()
print(utcnow)  # data e ora UTC

# Attributi
print(dt.year, dt.month, dt.day, dt.hour, dt.minute, dt.second)

# Metodi
dt2 = dt.replace(year=2025, hour=10)
print(dt2)
```

### Classe `timedelta`

```
delta = datetime.timedelta(days=5, hours=3, minutes=10)
print(delta)  # 5 days, 3:10:00

# Operazioni con timedelta
dopo = dt + delta
prima = dt - delta
print(dopo, prima)

# Differenza tra due date/datetime
diff = dt - datetime.datetime(2024, 5, 1)
print(diff.days, diff.total_seconds())
```

### Fusi orari (`timezone`)

```
from datetime import timezone, timedelta

# UTC
dt_utc = datetime.datetime.now(timezone.utc)
print(dt_utc)

# Fuso orario personalizzato (es. Italia, UTC+2)
tz = timezone(timedelta(hours=2))
dt_tz = datetime.datetime.now(tz)
print(dt_tz)
```

### Conversione tra fusi orari

```
dt = datetime.datetime(2024, 6, 1, 12, 0, tzinfo=timezone.utc)
dt_italia = dt.astimezone(timezone(timedelta(hours=2)))
print(dt_italia)
```

### Parsing e formattazione di date e orari

**Formattazione:**

```
dt = datetime.datetime.now()
s = dt.strftime("%d/%m/%Y %H:%M:%S")
print(s)  # es: 01/06/2024 14:30:00
```

**Parsing:**

```
s = "01/06/2024 14:30:00"
dt = datetime.datetime.strptime(s, "%d/%m/%Y %H:%M:%S")
print(dt)
```

**Codici di formattazione principali:**

- `%Y`: anno (es. 2024)
- `%m`: mese (01-12)
- `%d`: giorno (01-31)
- `%H`: ora (00-23)
- `%M`: minuti (00-59)
- `%S`: secondi (00-59)
- `%f`: microsecondi
- `%A`: giorno della settimana (nome completo)
- `%a`: giorno della settimana (abbreviato)
- `%B`: mese (nome completo)
- `%b`: mese (abbreviato)

### Esempi pratici

```
# Calcolare l'eta'
nascita = datetime.date(2000, 1, 1)
oggi = datetime.date.today()
eta = oggi.year - nascita.year - (
    (oggi.month, oggi.day) < (nascita.month, nascita.day)
)
print(eta)

# Giorno della settimana
print(oggi.weekday())       # 0=lunedi', 6=domenica
print(oggi.isoweekday())    # 1=lunedi', 7=domenica

# Primo e ultimo giorno del mese
from calendar import monthrange
anno, mese = oggi.year, oggi.month
primo = datetime.date(anno, mese, 1)
ultimo = datetime.date(anno, mese, monthrange(anno, mese)[1])
print(primo, ultimo)
```

### Gestione timestamp Unix

```
# Da datetime a timestamp
dt = datetime.datetime.now()
ts = dt.timestamp()
print(ts)
```

```
# Da timestamp a datetime
dt2 = datetime.datetime.fromtimestamp(ts)
print(dt2)
```

## Date e orari ISO 8601

```
dt = datetime.datetime.now()
iso = dt.isoformat()
print(iso)  # es: 2024-06-01T14:30:00.123456

# Parsing da stringa ISO
dt2 = datetime.datetime.fromisoformat(iso)
print(dt2)
```

## Date e orari in formato locale

```
import locale
locale.setlocale(locale.LC_TIME, "it_IT.UTF-8")
# Su sistemi compatibili
dt = datetime.datetime.now()
print(dt.strftime("%A %d %B %Y"))  # es: sabato 01 giugno 2024
```

## Modulo `calendar`

Il modulo `calendar` permette di lavorare con calendari, settimane, giorni del mese, ecc.

```
import calendar
print(calendar.month(2024, 6))
print(calendar.isleap(2024))  # True se anno bisestile
```

## Modulo `time`

Il modulo `time` offre funzioni a basso livello per lavorare con il tempo (timestamp, sleep, ecc.):

```
import time
print(time.time())  # timestamp corrente
time.sleep(2)       # pausa di 2 secondi
```

## Gestione avanzata dei fusi orari

Per una gestione avanzata dei fusi orari (es. ora legale, database IANA), si consiglia il modulo `zoneinfo` (da Python 3.9+) o librerie esterne come `pytz`:

```
from zoneinfo import ZoneInfo
dt = datetime.datetime(2024, 6, 1, 12, 0, tzinfo=ZoneInfo("Europe/
    Rome"))
print(dt)
```

### Librerie esterne per date e orari

- `pytz`: gestione avanzata dei fusi orari (ora deprecato in favore di `zoneinfo`)
- `dateutil`: parsing flessibile, ricorrenze, intervalli
- `arrow`, `pendulum`, `moment`: API più semplici e potenti per date e orari

### Esempio con `dateutil`

```
from dateutil.parser import parse
dt = parse("1 giugno 2024 14:30", dayfirst=True)
print(dt)
```

### Best practice e attenzioni

- Preferisci sempre oggetti `datetime` con fuso orario esplicito (`tzinfo`) per evitare ambiguità.
- Usa `timedelta` per calcoli temporali, non sommare direttamente giorni/ore/minuti.
- Per serializzare date e orari, usa formati standard come ISO 8601.
- Attenzione alle differenze tra ora locale e UTC.
- Per calcoli complessi su ricorrenze, festività, ecc., usa librerie esterne.

### Riepilogo

- `datetime` è il modulo standard per gestire date e orari in Python.
- Offre classi per date, orari, intervalli, fusi orari e parsing/formatting.
- Supporta operazioni aritmetiche, confronto, conversione tra formati e fusi orari.
- Per esigenze avanzate, esistono librerie esterne come `dateutil`, `arrow`, `pendulum`.

## 22 Classe Math

La **classe** (in realtà **modulo**) `math` in Python fornisce funzioni matematiche standard, costanti e strumenti per lavorare con numeri reali (float). È parte della libreria standard di Python e va importato prima dell'uso.

## Importazione del modulo

```
import math
```

## Costanti principali

- `math.pi`: il valore di $\pi$ (3.141592...)
- `math.e`: il valore di $e$ (2.718281...)
- `math.tau`: il valore di $\tau$ ($2\pi$)
- `math.inf`: infinito positivo
- `math.nan`: valore "not a number"

```
print(math.pi)     # 3.141592653589793
print(math.e)      # 2.718281828459045
print(math.tau)    # 6.283185307179586
print(math.inf)    # inf
print(math.nan)    # nan
```

## Funzioni aritmetiche di base

- `math.ceil(x)`: arrotonda per eccesso (al prossimo intero)
- `math.floor(x)`: arrotonda per difetto (all'intero precedente)
- `math.trunc(x)`: tronca la parte decimale
- `math.fabs(x)`: valore assoluto (float)
- `math.copysign(x, y)`: valore di $x$ col segno di $y$
- `math.fmod(x, y)`: resto della divisione (float)
- `math.remainder(x, y)`: resto secondo la regola IEEE 754
- `math.modf(x)`: restituisce parte frazionaria e intera

```
print(math.ceil(2.3))     # 3
print(math.floor(2.7))    # 2
print(math.trunc(-2.7))   # -2
print(math.fabs(-5))      # 5.0
print(math.copysign(3, -1)) # -3.0
print(math.fmod(7, 3))    # 1.0
print(math.modf(3.14))    # (0.14000000000000012, 3.0)
```

## Radici, potenze e logaritmi

- `math.sqrt(x)`: radice quadrata
- `math.pow(x, y)`: $x^y$ (sempre float)
- `math.exp(x)`: $e^x$
- `math.expm1(x)`: $e^x - 1$ (preciso per $x$ vicino a 0)
- `math.log(x[, base])`: logaritmo (default base $e$)
- `math.log10(x)`: logaritmo in base 10
- `math.log2(x)`: logaritmo in base 2
- `math.log1p(x)`: $\log(1 + x)$ (preciso per $x$ vicino a 0)

```
print(math.sqrt(16))      # 4.0
print(math.pow(2, 5))     # 32.0
print(math.exp(2))        # 7.38905609893065
print(math.log(8, 2))     # 3.0
print(math.log10(100))    # 2.0
print(math.log2(32))      # 5.0
```

## Funzioni trigonometriche

- `math.sin(x)`, `math.cos(x)`, `math.tan(x)`: seno, coseno, tangente (in radianti)
- `math.asin(x)`, `math.acos(x)`, `math.atan(x)`: arcotrigonometriche
- `math.atan2(y, x)`: arcotangente di $y/x$ (considera il quadrante)
- `math.hypot(x, y)`: $\sqrt{x^2 + y^2}$ (modulo vettore)
- `math.degrees(x)`: converte radianti in gradi
- `math.radians(x)`: converte gradi in radianti

```
print(math.sin(math.pi/2))    # 1.0
print(math.cos(0))            # 1.0
print(math.tan(math.pi/4))    # 1.0
print(math.degrees(math.pi)) # 180.0
print(math.radians(180))      # 3.141592653589793
print(math.hypot(3, 4))       # 5.0
```

## Funzioni iperboliche

- `math.sinh(x)`, `math.cosh(x)`, `math.tanh(x)`
- `math.asinh(x)`, `math.acosh(x)`, `math.atanh(x)`

```
print(math.sinh(1))
print(math.cosh(0))
print(math.tanh(2))
```

## Funzioni speciali

- `math.factorial(n)`: fattoriale di $n$ (solo interi $\geq 0$)
- `math.gamma(x)`: funzione gamma generalizzata
- `math.lgamma(x)`: logaritmo della funzione gamma
- `math.comb(n, k)`: combinazioni di $n$ su $k$
- `math.perm(n, k)`: permutazioni di $n$ su $k$
- `math.gcd(a, b)`: massimo comune divisore
- `math.lcm(a, b)`: minimo comune multiplo (da Python 3.9)
- `math.isqrt(n)`: radice intera di $n$

```
print(math.factorial(5))    # 120
print(math.comb(5, 2))      # 10
print(math.perm(5, 2))      # 20
print(math.gcd(12, 18))     # 6
print(math.lcm(12, 18))     # 36
print(math.isqrt(10))       # 3
```

## Funzioni di confronto e test

- `math.isfinite(x)`: True se $x$ è finito
- `math.isinf(x)`: True se $x$ è infinito
- `math.isnan(x)`: True se $x$ è NaN
- `math.isclose(a, b, rel_tol=..., abs_tol=...)`: True se $a$ e $b$ sono "vicini"

```
print(math.isfinite(1e100))     # True
print(math.isinf(math.inf))     # True
print(math.isnan(math.nan))     # True
print(math.isclose(0.1+0.2, 0.3)) # True
```

## Funzioni di arrotondamento avanzate

- `math.frexp(x)`: scompone $x$ in mantissa ed esponente
- `math.ldexp(m, e)`: $m \times 2^e$
- `math.nextafter(x, y)`: prossimo float dopo $x$ verso $y$
- `math.ulp(x)`: unità di ultimo posto (precisione macchina)

```
print(math.frexp(8))       # (0.5, 4) perche' 8 = 0.5 * 2^4
print(math.ldexp(0.5, 4)) # 8.0
```

## Funzioni di segno e manipolazione bit

- `math.copysign(x, y)`: valore di $x$ col segno di $y$
- `math.fsum(iterable)`: somma precisa di float
- `math.prod(iterable)`: prodotto di tutti gli elementi (da Python 3.8)

```
print(math.fsum([0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1])
    ) # 1.0
print(math.prod([1, 2, 3, 4])) # 24
```

## Funzioni per numeri complessi

Per numeri complessi, usa il modulo `cmath`, non `math`.

## Limitazioni del modulo math

- Lavora solo con numeri reali (float), non con complessi.
- Per funzioni statistiche, random, ecc., usa i moduli `statistics`, `random`, ecc.
- Per numeri complessi: `cmath`.

## Esempi pratici

```
# Calcolo area di un cerchio
r = 5
area = math.pi * r ** 2

# Calcolo distanza euclidea tra due punti
x1, y1 = 1, 2
x2, y2 = 4, 6
distanza = math.hypot(x2 - x1, y2 - y1)
```

### Documentazione e approfondimenti

Per la lista completa delle funzioni e costanti, consulta la documentazione ufficiale: `https://docs.python.org/3/library/math.html`

### Conclusioni

Il modulo `math` è essenziale per calcoli matematici di base e avanzati in Python. Offre funzioni veloci, precise e ottimizzate per la maggior parte delle esigenze scientifiche, ingegneristiche e di calcolo numerico.

## 23 Json

Il modulo **json** in Python permette di lavorare con dati in formato JSON (JavaScript Object Notation), uno standard molto diffuso per lo scambio di dati tra applicazioni, servizi web, API e file di configurazione. JSON è un formato testuale, leggibile sia da umani che da macchine, e rappresenta strutture dati come oggetti (dizionari), array (liste), stringhe, numeri, booleani e null.

### Importazione del modulo

```
import json
```

### Funzionalità principali

Il modulo `json` consente di:

- Serializzare oggetti Python in stringhe JSON (**dumping**)
- Deserializzare stringhe JSON in oggetti Python (**loading**)
- Leggere e scrivere file JSON
- Personalizzare la serializzazione/deserializzazione

### Conversione tra Python e JSON

**Mappatura dei tipi:**

| Python | JSON |
|---|---|
| dict | object |
| list, tuple | array |
| str | string |
| int, float | number |
| True | true |
| False | false |
| None | null |

## Serializzazione: da Python a JSON

- `json.dumps(obj, ...)`: converte un oggetto Python in una stringa JSON
- `json.dump(obj, file, ...)`: scrive un oggetto Python in un file come JSON

```
dati = {"nome": "Anna", "eta": 25, "iscritta": True}
s = json.dumps(dati)
print(s)  # {"nome": "Anna", "eta": 25, "iscritta": true}

with open("dati.json", "w", encoding="utf-8") as f:
    json.dump(dati, f)
```

## Deserializzazione: da JSON a Python

- `json.loads(s, ...)`: converte una stringa JSON in oggetto Python
- `json.load(file, ...)`: legge JSON da file e restituisce oggetto Python

```
s = '{"nome": "Anna", "eta": 25, "iscritta": true}'
dati = json.loads(s)
print(dati["nome"])  # Anna

with open("dati.json", "r", encoding="utf-8") as f:
    dati = json.load(f)
```

## Opzioni di serializzazione

- `indent`: aggiunge indentazione per una stampa leggibile
- `separators`: personalizza i separatori (default: (', ', ': '))
- `sort_keys`: ordina le chiavi degli oggetti
- `ensure_ascii`: True (default) esegue escape dei caratteri non ASCII, False mantiene UTF-8

```
print(json.dumps(dati, indent=4, sort_keys=True, ensure_ascii=False
    ))
```

## Gestione di tipi non supportati

Per serializzare tipi non standard (es. datetime, oggetti personalizzati), puoi:

- Usare il parametro `default` in `dumps`/`dump`
- Implementare un encoder personalizzato

```
import datetime

def encoder(obj):
    if isinstance(obj, datetime.datetime):
        return obj.isoformat()
    raise TypeError(f"Tipo non serializzabile: {type(obj)}")

dati = {"ora": datetime.datetime.now()}
print(json.dumps(dati, default=encoder))
```

## Decodifica personalizzata

Puoi personalizzare la decodifica con il parametro `object_hook`:

```
def decoder(dct):
    if "ora" in dct:
        dct["ora"] = datetime.datetime.fromisoformat(dct["ora"])
    return dct

s = '{"ora": "2024-06-01T12:00:00"}'
dati = json.loads(s, object_hook=decoder)
```

## Gestione degli errori

- `json.JSONDecodeError`: errore di parsing JSON
- `TypeError`: errore di serializzazione di tipo non supportato

```
try:
    dati = json.loads("non e' json")
except json.JSONDecodeError as e:
    print("Errore di parsing:", e)
```

## Lettura e scrittura di file JSON

```
# Scrittura
with open("file.json", "w", encoding="utf-8") as f:
    json.dump(dati, f, indent=2, ensure_ascii=False)

# Lettura
with open("file.json", "r", encoding="utf-8") as f:
    dati = json.load(f)
```

## Parsing di JSON parziale (streaming)

Per file molto grandi, puoi usare librerie esterne come `ijson` per il parsing incrementale.

## Limiti e attenzioni

- Solo tipi base Python sono supportati nativamente (dict, list, str, int, float, bool, None)
- Le chiavi degli oggetti JSON devono essere stringhe
- I commenti non sono ammessi in JSON standard
- I numeri molto grandi possono perdere precisione
- Le tuple vengono convertite in array (list)
- Attenzione a serializzare oggetti mutabili/nidificati

## Esempi pratici

```
# Convertire una lista di dizionari in JSON
utenti = [{"nome": "Anna"}, {"nome": "Luca"}]
s = json.dumps(utenti, indent=2)

# Caricare dati da una API REST
import requests
r = requests.get("https://jsonplaceholder.typicode.com/todos/1")
dati = r.json()  # oppure: json.loads(r.text)
```

## Serializzazione avanzata: JSONEncoder e JSONDecoder

Puoi estendere `json.JSONEncoder` per personalizzare la serializzazione:

```
class MioEncoder(json.JSONEncoder):
    def default(self, obj):
        if isinstance(obj, datetime.datetime):
            return obj.isoformat()
        return super().default(obj)

json.dumps({"ora": datetime.datetime.now()}, cls=MioEncoder)
```

## Compatibilità con altri linguaggi

Il formato JSON prodotto da Python è compatibile con JavaScript, Java, Go, PHP, ecc.

## Librerie esterne

- `ujson`: più veloce, ma meno compatibile
- `orjson`: molto veloce, supporta tipi avanzati
- `simplejson`: estende il modulo standard

### Best practice

- Usa `indent` per file leggibili
- Usa `ensure_ascii=False` per caratteri Unicode
- Gestisci gli errori di parsing
- Valida i dati dopo il caricamento
- Per dati complessi, serializza manualmente i tipi non standard

### Documentazione ufficiale

`https://docs.python.org/3/library/json.html`

### Conclusioni

Il modulo `json` è fondamentale per la serializzazione, la comunicazione tra servizi e la persistenza di dati strutturati in Python. La sua conoscenza è essenziale per lavorare con API, file di configurazione, web e applicazioni moderne.

## 24 Pip

Il comando `pip` è il gestore di pacchetti ufficiale di Python. Permette di installare, aggiornare, rimuovere e gestire pacchetti (librerie, moduli) Python provenienti dal Python Package Index (PyPI) e da altre fonti.

### Cos'è pip

- **pip** sta per "Pip Installs Packages" o "Pip Installs Python".
- È incluso di default in Python dalla versione 3.4+.
- Gestisce l'installazione di pacchetti da PyPI (`https://pypi.org/`), repository privati, file locali, URL, ecc.
- Permette di gestire le dipendenze di progetto.

### Installazione e aggiornamento di pip

- **Verifica se pip è installato:**

```
python -m pip --version
pip --version
```

- **Aggiornamento di pip:**

```
python -m pip install --upgrade pip
```

- **Installazione manuale (se necessario):** Scarica `get-pip.py` da `https://bootstrap.pypa.io/get-pip.py` e lancia:

```
python get-pip.py
```

## Comandi principali di pip

- `pip install nome_pacchetto`: installa un pacchetto
- `pip install nome==versione`: installa una versione specifica
- `pip install "nome>=1.0,<2.0"`: installa una versione compresa tra due limiti
- `pip install -U nome`: aggiorna un pacchetto (`-U` = `-upgrade`)
- `pip uninstall nome`: disinstalla un pacchetto
- `pip show nome`: mostra informazioni su un pacchetto
- `pip list`: elenca tutti i pacchetti installati
- `pip freeze`: elenca i pacchetti installati in formato `requirements.txt`
- `pip search parola`: cerca pacchetti su PyPI (`search` è deprecato)
- `pip check`: verifica conflitti tra dipendenze

## Installazione da file, URL e repository

- `pip install ./pacchetto.whl`: da file wheel locale
- `pip install ./cartella/`: da directory locale con `setup.py`
- `pip install https://url/del/pacchetto.tar.gz`: da URL
- `pip install git+https://github.com/utente/repo.git`: da repository Git

## Gestione delle dipendenze

- `pip install -r requirements.txt`: installa tutti i pacchetti elencati in un file
- `pip freeze > requirements.txt`: salva l'elenco dei pacchetti installati

## Virtual environment e pip

- È buona pratica usare ambienti virtuali (`venv`, `virtualenv`) per isolare le dipendenze di progetto.
- Quando attivi un ambiente virtuale, `pip` installa i pacchetti solo in quell'ambiente.
- `python -m venv nome_env` crea un ambiente virtuale.

## Opzioni utili di pip

- `-user`: installa il pacchetto solo per l'utente corrente
- `-upgrade`: aggiorna il pacchetto all'ultima versione
- `-force-reinstall`: reinstalla anche se già presente
- `-no-deps`: non installa le dipendenze
- `-pre`: include versioni pre-release
- `-proxy`: usa un proxy per la connessione
- `-trusted-host`: aggiunge host fidati (utile con proxy/firewall)

## Configurazione di pip

- File di configurazione: `pip.ini` (Windows), `.pip/pip.conf` (Linux/Mac)
- Puoi configurare repository alternativi, proxy, opzioni di default, ecc.

## Installazione di pacchetti da repository privati

```
pip install --index-url https://mio.repo/simple nome
pip install --extra-index-url https://altro.repo/simple nome
```

## Disinstallazione di pacchetti

```
pip uninstall nome_pacchetto
```

## Aggiornamento di tutti i pacchetti

Non esiste un comando unico, ma puoi usare:

```
pip list --outdated
pip install --upgrade nome_pacchetto
```

Oppure uno script per aggiornare tutto:

```
pip list --outdated --format=freeze |
    grep -v '^\-e' |
    cut -d = -f 1 |
    xargs -n1 pip install -U
```

## Installazione di pacchetti in modalità "editable"

```
pip install -e ./mio_pacchetto/
```

Utile per lo sviluppo di pacchetti locali.

## Cache di pip

- pip mantiene una cache dei pacchetti scaricati per velocizzare le installazioni successive.
- Puoi svuotare la cache con `pip cache purge`.

## Problemi comuni e soluzioni

- **Permessi**: usa `-user` o esegui come amministratore/root se necessario.
- **Proxy/firewall**: configura `-proxy` o `-trusted-host`.
- **Versioni multiple di Python**: usa `python3 -m pip` o `python -m pip` per evitare conflitti.
- **Conflitti di dipendenze**: usa `pip check` per individuarli.

## pip e PyPI

- PyPI (`https://pypi.org/`) è il repository ufficiale di pacchetti Python.
- Puoi pubblicare i tuoi pacchetti su PyPI usando `twine`.

## pipx

- `pipx` è uno strumento per installare ed eseguire applicazioni Python in ambienti isolati.
- Utile per tool da linea di comando.

## pipdeptree

- `pipdeptree` mostra la struttura delle dipendenze dei pacchetti installati.
- Installa con `pip install pipdeptree`.

## pip e sicurezza

- Installa pacchetti solo da fonti affidabili.
- Controlla sempre le dipendenze e le versioni.
- Usa ambienti virtuali per evitare conflitti e rischi.

## pip e requirements.txt

- `requirements.txt` è il file standard per elencare le dipendenze di un progetto.
- Puoi specificare versioni, intervalli, URL, riferimenti a repository.
- Esempio:

```
requests>=2.25,<3.0
numpy==1.24.0
git+https://github.com/utente/repo.git
```

## pip e wheel

- `wheel` è il formato binario standard per la distribuzione di pacchetti Python.
- pip preferisce installare da wheel se disponibile (più veloce).
- Puoi creare un wheel con `python setup.py bdist_wheel`.

## pip e compatibilità

- pip funziona sia con Python 2 che con Python 3 (ma Python 2 è deprecato).
- Alcuni pacchetti potrebbero non essere compatibili con tutte le versioni di Python.

## pip e documentazione

- Documentazione ufficiale: `https://pip.pypa.io/`
- Lista completa dei comandi: `pip -help`
- Per ogni comando: `pip <comando> -help`

## Esempi pratici

```
# Installare una libreria
pip install requests

# Aggiornare una libreria
pip install --upgrade numpy

# Disinstallare una libreria
pip uninstall pandas

# Installare tutte le dipendenze di un progetto
pip install -r requirements.txt

# Salvare tutte le dipendenze in un file
```

```
pip freeze > requirements.txt

# Mostrare informazioni su un pacchetto
pip show flask

# Elencare tutti i pacchetti installati
pip list

# Verificare conflitti tra dipendenze
pip check
```

### Conclusioni

`pip` è uno strumento fondamentale per la gestione delle dipendenze e dei pacchetti in Python. Permette di installare, aggiornare, rimuovere e gestire librerie in modo semplice e potente, facilitando lo sviluppo, la distribuzione e la manutenzione dei progetti Python. La conoscenza approfondita di pip e delle sue funzionalità è essenziale per ogni sviluppatore Python moderno.

## 25 Try except

Il costrutto `try-except` in Python serve per la gestione delle eccezioni, ovvero per intercettare e gestire errori che si verificano durante l'esecuzione del programma senza interrompere il flusso del codice. È uno strumento fondamentale per scrivere codice robusto, sicuro e professionale.

### Cos'è un'eccezione?

Un'eccezione è un evento anomalo (errore) che si verifica durante l'esecuzione di un programma e che interrompe il normale flusso delle istruzioni. In Python, le eccezioni sono oggetti che rappresentano errori di vario tipo (es. divisione per zero, file non trovato, indice fuori range, ecc.).

### Sintassi base di try-except

```
try:
    # blocco di codice che potrebbe generare un'eccezione
    x = 1 / 0
except ZeroDivisionError:
    # blocco eseguito se si verifica l'eccezione specificata
    print("Divisione per zero!")
```

**Funzionamento:**

- Il codice nel blocco `try` viene eseguito normalmente.

- Se si verifica un'eccezione, l'esecuzione passa immediatamente al blocco `except` corrispondente.
- Se nessuna eccezione si verifica, il blocco `except` viene saltato.
- Se l'eccezione non viene gestita, il programma si interrompe e viene mostrato un traceback.

### Gestione di più eccezioni

Puoi gestire diversi tipi di eccezioni con più blocchi `except`:

```
try:
    # codice
except ValueError:
    print("Valore non valido")
except ZeroDivisionError:
    print("Divisione per zero")
except Exception as e:
    print("Errore generico:", e)
```

### Gestione multipla in un solo except

Puoi gestire più tipi di eccezioni con una sola istruzione `except` usando una tupla:

```
try:
    # codice
except (ValueError, TypeError):
    print("Errore di valore o di tipo")
```

### Ottenere dettagli sull'eccezione

Puoi accedere all'oggetto eccezione usando `as`:

```
try:
    x = int("abc")
except ValueError as e:
    print("Errore:", e)
```

### Blocco else

Il blocco `else` viene eseguito solo se nessuna eccezione si è verificata nel blocco `try`:

```
try:
    x = 10 / 2
except ZeroDivisionError:
    print("Divisione per zero")
else:
    print("Nessuna eccezione, risultato:", x)
```

### Blocco finally

Il blocco `finally` viene sempre eseguito, sia che si verifichi un'eccezione sia che non si verifichi:

```
try:
    f = open("file.txt")
except FileNotFoundError:
    print("File non trovato")
else:
    print("File aperto correttamente")
finally:
    print("Operazione terminata")
```

Il `finally` è utile per rilasciare risorse (file, connessioni, ecc.).

### Eccezioni annidate

Puoi annidare blocchi `try-except`:

```
try:
    try:
        x = int(input("Numero: "))
        y = 10 / x
    except ValueError:
        print("Non e' un numero")
except ZeroDivisionError:
    print("Divisione per zero")
```

### Rilanciare un'eccezione

Puoi rilanciare un'eccezione con `raise`:

```
try:
    x = 1 / 0
except ZeroDivisionError:
    print("Errore, rilancio l'eccezione")
    raise
```

### Sollevare eccezioni personalizzate

Puoi sollevare un'eccezione con `raise`:

```
def dividi(a, b):
    if b == 0:
        raise ValueError("Il divisore non puo' essere zero")
    return a / b
```

## Definire eccezioni personalizzate

Puoi creare le tue eccezioni ereditando da `Exception`:

```
class ErrorePersonalizzato(Exception):
    pass

try:
    raise ErrorePersonalizzato("Messaggio di errore")
except ErrorePersonalizzato as e:
    print("Eccezione personalizzata:", e)
```

## Gerarchia delle eccezioni

Tutte le eccezioni derivano dalla classe base `BaseException`, ma normalmente si eredita da `Exception`. Alcune eccezioni comuni:

- `Exception`: base per tutte le eccezioni standard
- `ValueError`, `TypeError`, `IndexError`, `KeyError`, `ZeroDivisionError`, `FileNotFoundError`, ecc.
- `KeyboardInterrupt`, `SystemExit`: derivano da `BaseException`, non vanno normalmente intercettate

## Best practice

- Gestisci solo le eccezioni che puoi realmente trattare.
- Non usare `except:` senza specificare il tipo (rischi di nascondere errori gravi).
- Usa `except Exception:` per intercettare tutte le eccezioni standard, ma solo se necessario.
- Usa il blocco `finally` per chiudere file, connessioni, ecc.
- Documenta le eccezioni che una funzione può sollevare.
- Preferisci sollevare eccezioni specifiche e informative.

## Esempi pratici

```
# Lettura sicura di un file
try:
    with open("dati.txt") as f:
        dati = f.read()
except FileNotFoundError:
    print("File non trovato")
except Exception as e:
    print("Errore generico:", e)
```

```
else:
    print("File letto correttamente")
finally:
    print("Operazione completata")

# Input numerico sicuro
while True:
    try:
        x = int(input("Inserisci un numero: "))
        break
    except ValueError:
        print("Devi inserire un numero intero!")
```

## Eccezioni e funzioni built-in

Molte funzioni Python sollevano eccezioni in caso di errore (es. `int()`, `open()`, `list.index()`, ecc.).

## Eccezioni e traceback

Quando un'eccezione non viene gestita, Python stampa un traceback che mostra la sequenza delle chiamate che hanno portato all'errore.

## Modulo traceback

Per gestire e stampare manualmente il traceback:

```
import traceback

try:
    1 / 0
except Exception:
    traceback.print_exc()
```

## Eccezioni e context manager

I context manager (`with`) gestiscono automaticamente le eccezioni e rilasciano le risorse anche in caso di errore.

## Eccezioni e performance

Lanciare e gestire eccezioni è più lento rispetto al normale flusso di codice: usale per gestire errori, non per il controllo di flusso ordinario.

## Eccezioni e documentazione

Documenta sempre le eccezioni che una funzione può sollevare, soprattutto se sono personalizzate.

### Conclusioni

La gestione delle eccezioni con `try-except` è fondamentale per scrivere codice Python robusto, sicuro e professionale. Permette di gestire gli errori in modo controllato, migliorare l'esperienza utente e prevenire crash inaspettati. La conoscenza approfondita di questo costrutto e delle best practice è essenziale per ogni sviluppatore Python.

## 26 Input Dati

La funzione **input** è una funzione built-in fondamentale per acquisire dati dall'utente tramite la tastiera. Permette di leggere stringhe inserite dall'utente durante l'esecuzione del programma. Di seguito una panoramica completa sull'input dei dati in Python.

### Funzione `input()`

La funzione `input()` legge una riga di testo dalla tastiera e la restituisce come stringa (`str`).

```
nome = input("Come ti chiami? ")
print("Ciao,", nome)
```

**Nota:** Il testo tra parentesi viene visualizzato come *prompt* per l'utente.

### Tipo di ritorno

`input()` restituisce sempre una stringa, anche se l'utente inserisce numeri.

```
x = input("Inserisci un numero: ")
print(type(x))  # <class 'str'>
```

### Conversione del tipo di dato (casting)

Per ottenere un numero intero o float, bisogna convertire la stringa:

```
x = int(input("Inserisci un intero: "))
y = float(input("Inserisci un numero decimale: "))
```

Se l'utente inserisce un valore non valido, viene sollevata un'eccezione (`ValueError`).

### Gestione degli errori

Per evitare crash, usa `try-except`:

```
try:
    eta = int(input("Quanti anni hai? "))
except ValueError:
    print("Devi inserire un numero intero!")
```

### Input multiplo su una riga

Puoi acquisire più valori separati da spazi e convertirli con `split()`:

```
a, b = input("Inserisci due numeri separati da spazio: ").split()
a = int(a)
b = int(b)
```

Oppure in una sola riga:

```
a, b = map(int, input("Due numeri: ").split())
```

### Input di liste o sequenze

Per acquisire una lista di numeri:

```
numeri = list(map(int, input("Inserisci numeri separati da spazio:
    ").split()))
```

### Input senza prompt

Se non passi argomenti a `input()`, non viene mostrato alcun messaggio:

```
x = input()
```

### Input e spazi bianchi

`input()` legge tutta la riga, inclusi spazi e caratteri speciali. Puoi usare `strip()` per rimuovere spazi iniziali/finali:

```
nome = input("Nome: ").strip()
```

### Input e caratteri speciali

Tutto ciò che l'utente digita viene letto come stringa, inclusi caratteri speciali, lettere accentate, ecc.

### Input e EOF

Se l'utente invia un EOF (Ctrl+D su Linux/Mac, Ctrl+Z su Windows), viene sollevata un'eccezione `EOFError`.

```
try:
    x = input()
except EOFError:
    print("Fine input")
```

## Input in cicli

Spesso si usa `input()` in un ciclo per acquisire più dati:

```
while True:
    riga = input("Scrivi qualcosa (q per uscire): ")
    if riga == "q":
        break
    print("Hai scritto:", riga)
```

## Input e Unicode

`input()` supporta caratteri Unicode (accenti, simboli, ecc.), purché il terminale li supporti.

## Input e Python 2 vs Python 3

In Python 2 esistevano `input()` e `raw_input()`. In Python 3 esiste solo `input()`, che si comporta come il vecchio `raw_input()`.

## Input da file (redirezione)

Puoi simulare l'input da tastiera redirigendo un file:

```
python mio_script.py < dati.txt
```

## Input e automazione

Per test automatici, puoi simulare l'input usando `unittest.mock` o passando dati tramite stdin.

## Input e sicurezza

Non fidarti mai ciecamente dell'input utente: valida e gestisci sempre i possibili errori.

## Esempi pratici

```
# Input di una stringa
nome = input("Nome: ")

# Input di un intero con controllo
while True:
    try:
        eta = int(input("Eta': "))
        break
    except ValueError:
        print("Inserisci un numero valido!")
```

```
# Input di una lista di numeri
numeri = list(map(int, input("Numeri separati da spazio: ").split()
    ))
```

### Conclusioni

La funzione `input()` è il metodo standard per acquisire dati dall'utente in Python. È semplice, potente e flessibile, ma richiede attenzione nella conversione dei tipi e nella gestione degli errori. Una corretta gestione dell'input è fondamentale per scrivere programmi interattivi, robusti e user-friendly.

## 27 Formattazione stringhe

La **formattazione delle stringhe** in Python è l'insieme delle tecniche che permettono di inserire valori variabili, numeri, date, oggetti e risultati di espressioni all'interno di una stringa, controllando anche l'aspetto (allineamento, numero di decimali, padding, ecc.). Esistono diversi metodi per formattare le stringhe in Python, ciascuno con caratteristiche e sintassi proprie.

### 1. Concatenazione con +

Il metodo più semplice (ma meno flessibile) consiste nel concatenare stringhe e convertire manualmente i valori:

```
nome = "Anna"
eta = 25
s = "Ciao, mi chiamo " + nome + " e ho " + str(eta) + " anni."
print(s)
```

**Limiti:** poco leggibile, non adatto a tipi non stringa, difficile da mantenere.

### 2. Operatore % (vecchio stile, stile C)

Permette di inserire valori in una stringa usando specificatori di formato simili al C:

```
nome = "Luca"
eta = 30
s = "Ciao, mi chiamo %s e ho %d anni." % (nome, eta)
print(s)
```

**Specificatori principali:**

- `%s`: stringa
- `%d`: intero
- `%f`: float (decimale)
- `%.2f`: float con 2 decimali

- `%x`, `%o`: esadecimale, ottale

**Limiti:** meno flessibile, deprecato nelle nuove versioni.

## 3. Metodo `str.format()`

Metodo moderno e potente, permette di inserire valori tramite `{}` (placeholder):

```
nome = "Giulia"
eta = 28
s = "Ciao, mi chiamo {} e ho {} anni.".format(nome, eta)
print(s)
```

**Placeholder numerati o nominati:**

```
s = "Ciao, mi chiamo {0} e ho {1} anni.".format(nome, eta)
s = "Ciao, mi chiamo {nome} e ho {anni} anni.".format(nome=nome,
    anni=eta)
```

**Formattazione avanzata:**

- `:d`, `:f`, `:.2f`, `:x`, `:>10`, `:^10`, `:<10` per tipo, decimali, padding, allineamento
- `:0>5` padding con zeri a sinistra
- `:,` separatore delle migliaia

```
x = 1234.56789
print("{:.2f}".format(x))       # 1234.57
print("{:10.2f}".format(x))     # '   1234.57'
print("{:0>8.2f}".format(x))    # '01234.57'
print("{:,}".format(1000000))   # '1,000,000'
print("{:^10}".format("ciao")) # '   ciao   '
```

**Accesso a indici, attributi e dizionari:**

```
persona = {"nome": "Anna", "eta": 22}
print("Nome: {0[nome]}, Eta: {0[eta]}".format(persona))
class P: pass
p = P(); p.nome = "Luca"
print("Nome: {0.nome}".format(p))
```

## 4. f-string (formatted string literals, da Python 3.6+)

Il metodo più moderno, leggibile e potente. Anteponi una `f` alla stringa e inserisci espressioni tra `{}`:

```
nome = "Marco"
eta = 35
s = f"Ciao, mi chiamo {nome} e ho {eta} anni."
print(s)
```

**Supporta espressioni arbitrarie:**

```
print(f"Tra 5 anni avro' {eta + 5} anni.")
```

**Formattazione avanzata:**

```
x = 3.14159
print(f"Valore: {x:.2f}")       # Valore: 3.14
print(f"{1000:,}")              # 1,000
print(f"{'ciao':^10}")          # '   ciao   '
```

**Accesso a dizionari, attributi, indici:**

```
d = {"a": 1}
print(f"Valore: {d['a']}")
class P: pass
p = P(); p.x = 10
print(f"x={p.x}")
```

## 5. Template string (modulo `string`)

Per casi in cui serve sicurezza (es. input utente), usa `string.Template`:

```
from string import Template
t = Template("Ciao, $nome!")
print(t.substitute(nome="Anna"))
```

**Limiti:** meno potente, ma più sicuro contro injection.

## 6. Specificatori di formato (mini-language)

La mini-linguaggio di formattazione permette:

- **Tipo**: `d` (intero), `f` (float), `s` (stringa), `x` (esadecimale), `o` (ottale), `b` (binario)
- **Decimali**: `:.2f` (2 decimali)
- **Allineamento**: `<` (sinistra), `>` (destra), `^` (centrato)
- **Larghezza**: `:10` (10 caratteri)
- **Padding**: `:0>5` (zeri a sinistra)
- **Segno**: `:+` (mostra sempre il segno)
- **Separatore migliaia**: `:,`
- **Percentuale**: `:.2%`

```
n = 1234.567
print(f"{n:10.2f}")     # '   1234.57'
print(f"{n:0>10.2f}")   # '0001234.57'
print(f"{n:+.1f}")      # '+1234.6'
print(f"{n:,.2f}")      # '1,234.57'
print(f"{0.123:.2%}")   # '12.30%'
```

## 7. Escape di parentesi graffe

Per stampare una parentesi graffa in una stringa formattata, raddoppiala:

```
print(f"{{Questo e' tra parentesi graffe}}")
```

## 8. Formattazione di date e orari

Puoi formattare oggetti `datetime` direttamente:

```
import datetime
dt = datetime.datetime(2024, 6, 1, 14, 30)
print(f"{dt:%d/%m/%Y %H:%M}")  # 01/06/2024 14:30
```

## 9. Formattazione di numeri complessi, binari, ottali, esadecimali

```
n = 255
print(f"{n:b}")  # binario: 11111111
print(f"{n:o}")  # ottale: 377
print(f"{n:x}")  # esadecimale: ff
```

## 10. Arrotondamento e notazione scientifica

```
x = 12345.6789
print(f"{x:.2e}")  # 1.23e+04
```

## 11. Formattazione con variabili dinamiche

Puoi usare variabili per specificare la larghezza o i decimali:

```
w = 8
d = 3
x = 1.23456
print(f"{x:{w}.{d}f}")  #  '   1.235'
```

## 12. Formattazione di oggetti personalizzati

Puoi personalizzare la formattazione implementando il metodo `__format__` nella tua classe.

## 13. Formattazione multilinea

Le f-string possono essere multilinea:

```
nome = "Anna"
messaggio = f"""
Ciao {nome},
benvenuta!
"""
```

## 14. Formattazione e localizzazione

Per formattare numeri secondo la localizzazione:

```
import locale
locale.setlocale(locale.LC_ALL, "it_IT.UTF-8")
n = 1234567.89
print(locale.format_string("%.2f", n, grouping=True))  #
    '1.234.567,89'
```

## 15. Formattazione e sicurezza

Non usare mai input utente direttamente in f-string o `format()` se può contenere codice malevolo.

## 16. Riepilogo e best practice

- Preferisci le f-string (Python 3.6+) per leggibilità e potenza.
- Usa `str.format()` se devi supportare versioni precedenti.
- L'operatore `%` è obsoleto, usalo solo per compatibilità.
- Per casi di sicurezza, usa `string.Template`.
- Sfrutta la mini-linguaggio di formattazione per controllare decimali, padding, allineamento, ecc.
- Documenta sempre il formato delle stringhe se usato in API o file.

## 17. Documentazione ufficiale

- `https://docs.python.org/3/library/string.html#format-string-syntax`
- `https://docs.python.org/3/reference/lexical_analysis.html#f-strings`

## Conclusioni

La formattazione delle stringhe è fondamentale per produrre output leggibile, generare report, log, messaggi per l'utente, serializzare dati e molto altro. La padronanza delle tecniche di formattazione rende il codice più chiaro, potente e professionale.

# 28 Lavorare con i file

Lavorare con i file in Python è una delle operazioni fondamentali per leggere, scrivere, modificare e gestire dati persistenti. Python offre un supporto completo per la gestione dei file tramite funzioni built-in e moduli della libreria standard. Di seguito una panoramica dettagliata su tutto ciò che riguarda la manipolazione dei file in Python.

## Apertura di un file: `open()`

La funzione built-in `open()` permette di aprire un file e restituire un oggetto file. La sintassi è:

```
f = open("nomefile.txt", "modalita'", encoding="utf-8")
```

**Parametri principali:**

- `file`: percorso del file da aprire (relativo o assoluto)
- `mode`: modalità di apertura (default: `"r"`)
- `encoding`: codifica del file (consigliato: `"utf-8"` per testi)

### Modalità di apertura

- `"r"`: lettura (default), errore se il file non esiste
- `"w"`: scrittura, crea il file o sovrascrive se esiste
- `"a"`: append, scrive in fondo al file (crea se non esiste)
- `"x"`: scrittura esclusiva, errore se il file esiste già
- `"b"`: modalità binaria (es. `"rb"`, `"wb"`)
- `"t"`: modalità testo (default, es. `"rt"`, `"wt"`)
- `"+"`: lettura e scrittura (es. `"r+"`, `"w+"`, `"a+"`)

## Chiusura del file: `close()`

Dopo aver lavorato con un file, è importante chiuderlo per liberare risorse:

```
f.close()
```

**Nota:** Se dimentichi di chiudere un file, potresti perdere dati o bloccare risorse.

### Context manager: `with`

Il modo migliore per lavorare con i file è usare il context manager (`with`), che chiude automaticamente il file anche in caso di errore:

```
with open("file.txt", "r", encoding="utf-8") as f:
    dati = f.read()
# Qui il file e' gia' chiuso
```

### Lettura di file di testo

- `f.read()`: legge tutto il contenuto come stringa
- `f.readline()`: legge una riga alla volta (incluso il carattere di newline)
- `f.readlines()`: restituisce una lista di tutte le righe
- Iterazione diretta: `for riga in f: ...`

```
with open("file.txt", "r", encoding="utf-8") as f:
    contenuto = f.read()
    f.seek(0)  # Torna all'inizio
    prima_riga = f.readline()
    f.seek(0)
    tutte_le_righe = f.readlines()
    f.seek(0)
    for riga in f:
        print(riga.strip())
```

### Scrittura di file di testo

- `f.write(stringa)`: scrive una stringa nel file
- `f.writelines(lista)`: scrive una lista di stringhe (non aggiunge newline automaticamente)

```
with open("output.txt", "w", encoding="utf-8") as f:
    f.write("Ciao mondo!\n")
    f.writelines(["Prima riga\n", "Seconda riga\n"])
```

### Aggiunta (append) a un file

```
with open("log.txt", "a", encoding="utf-8") as f:
    f.write("Nuova riga\n")
```

## Lettura e scrittura di file binari

Per file non di testo (immagini, audio, pdf, ecc.), usa la modalità binaria:

```
with open("immagine.jpg", "rb") as f:
    dati = f.read()

with open("copia.jpg", "wb") as f:
    f.write(dati)
```

## Gestione della posizione nel file

- `f.tell()`: restituisce la posizione corrente (in byte)
- `f.seek(offset, whence)`: sposta il cursore (0: inizio, 1: posizione attuale, 2: fine)

```
with open("file.txt", "r", encoding="utf-8") as f:
    f.seek(5)
    print(f.read())
    f.seek(0)
    print(f.tell())  # 0
```

## Gestione degli errori

Lavorare con i file può generare eccezioni (`FileNotFoundError`, `IOError`, ecc.):

```
try:
    with open("file.txt") as f:
        dati = f.read()
except FileNotFoundError:
    print("File non trovato")
except Exception as e:
    print("Errore:", e)
```

## File temporanei

Per creare file temporanei usa il modulo `tempfile`:

```
import tempfile

with tempfile.NamedTemporaryFile(delete=False) as tmp:
    tmp.write(b"test")
    print(tmp.name)
```

## File e percorsi

Per lavorare con percorsi in modo portabile usa il modulo `os` o `pathlib`:

```
import os
from pathlib import Path

# Verifica esistenza file
if os.path.exists("file.txt"):
    print("Esiste")

# Pathlib (moderno)
p = Path("file.txt")
if p.exists():
    print("Esiste")
```

## Eliminare, rinominare, copiare file

```
import os
import shutil

os.remove("file.txt")              # elimina file
os.rename("vecchio.txt", "nuovo.txt")  # rinomina
shutil.copy("file1.txt", "file2.txt")  # copia file
shutil.move("file1.txt", "cartella/")  # sposta file
```

## Lettura e scrittura di file CSV

Per file CSV usa il modulo `csv`:

```
import csv

# Lettura
with open("dati.csv", newline="", encoding="utf-8") as f:
    reader = csv.reader(f)
    for riga in reader:
        print(riga)

# Scrittura
with open("output.csv", "w", newline="", encoding="utf-8") as f:
    writer = csv.writer(f)
    writer.writerow(["nome", "eta"])
    writer.writerows([["Anna", 25], ["Luca", 30]])
```

## Lettura e scrittura di file JSON

Vedi sezione dedicata, ma in breve:

```
import json
```

```
with open("dati.json", "w", encoding="utf-8") as f:
    json.dump({"a": 1}, f, indent=2)

with open("dati.json", "r", encoding="utf-8") as f:
    dati = json.load(f)
```

## Lettura e scrittura di file XML

Usa il modulo `xml.etree.ElementTree`:

```
import xml.etree.ElementTree as ET

tree = ET.parse("file.xml")
root = tree.getroot()
for elem in root:
    print(elem.tag, elem.text)
```

## File e codifica (encoding)

Specifica sempre l'encoding per file di testo (consigliato: `"utf-8"`). Attenzione a file con codifiche diverse (es. `"latin-1"`, `"cp1252"`).

## File e newline

Su Windows, Linux e Mac i caratteri di newline sono diversi. Python gestisce automaticamente la conversione in modalità testo. Per controllare il comportamento, usa il parametro `newline` in `open()`.

## File e bufferizzazione

`open()` accetta il parametro `buffering` per controllare la bufferizzazione. Di solito non serve modificarlo.

## File e oggetti StringIO/BytesIO

Per simulare file in memoria (utile per test), usa `io.StringIO` (testo) o `io.BytesIO` (binario):

```
from io import StringIO

f = StringIO()
f.write("test")
f.seek(0)
print(f.read())
```

## File e directory

Per lavorare con directory:

```
import os
from pathlib import Path

os.mkdir("nuova_cartella")
os.listdir(".")  # lista file e cartelle

# Pathlib
p = Path(".")
for file in p.iterdir():
    print(file)
```

## File e permessi

Attenzione ai permessi di lettura/scrittura/esecuzione. Su sistemi Unix puoi usare `os.chmod()`.

## File locking (blocco file)

Python non offre un locking portabile nativo. Su Unix puoi usare `fcntl`, su Windows `msvcrt`. Per casi avanzati, usa librerie esterne.

## File e grandi dimensioni

Per file molto grandi, leggi e scrivi a blocchi (chunk) per evitare di caricare tutto in memoria:

```
with open("grande.txt", "r", encoding="utf-8") as f:
    while True:
        blocco = f.read(1024)
        if not blocco:
            break
        # processa blocco
```

## File e compressione

Per file compressi usa i moduli `gzip`, `bz2`, `zipfile`:

```
import gzip

with gzip.open("file.txt.gz", "rt", encoding="utf-8") as f:
    dati = f.read()
```

## File e best practice

- Usa sempre il context manager (`with`) per evitare di dimenticare la chiusura del file.
- Gestisci le eccezioni per evitare crash in caso di file mancanti o permessi insufficienti.
- Specifica sempre l'encoding per file di testo.
- Non leggere mai file enormi tutti in memoria: usa la lettura a blocchi o riga per riga.
- Per dati strutturati (CSV, JSON, XML), usa i moduli dedicati.
- Per test, usa `StringIO`/`BytesIO` per simulare file in memoria.

## Esempi pratici

```
# Copiare un file di testo riga per riga
with open("input.txt", "r", encoding="utf-8") as fin, \
     open("output.txt", "w", encoding="utf-8") as fout:
    for riga in fin:
        fout.write(riga)

# Leggere solo le prime 10 righe di un file
with open("file.txt", "r", encoding="utf-8") as f:
    for i, riga in enumerate(f):
        if i >= 10:
            break
        print(riga.strip())
```

## Documentazione ufficiale

- `https://docs.python.org/3/library/functions.html#open`
- `https://docs.python.org/3/library/io.html`
- `https://docs.python.org/3/library/os.html`
- `https://docs.python.org/3/library/pathlib.html`

## Conclusioni

La gestione dei file in Python è semplice, potente e flessibile. Conoscere tutte le modalità di apertura, le tecniche di lettura e scrittura, la gestione degli errori, i moduli per dati strutturati e le best practice è fondamentale per scrivere programmi robusti, efficienti e portabili.

# 29 Gestione degli ambienti virtuali

La **gestione degli ambienti virtuali** in Python è fondamentale per isolare le dipendenze di progetto, evitare conflitti tra librerie e mantenere un ambiente di sviluppo pulito e riproducibile. Un ambiente virtuale è una directory che contiene una copia isolata dell'interprete Python, delle librerie e degli script installati tramite `pip`.

## Perché usare ambienti virtuali?

- Isolano le dipendenze di ciascun progetto.
- Permettono di lavorare con versioni diverse delle stesse librerie su progetti diversi.
- Evitano conflitti con i pacchetti installati globalmente.
- Facilitano la riproducibilità e la condivisione del progetto (requirements.txt).
- Consentono di testare codice con versioni diverse di Python/librerie.

## Strumenti principali per ambienti virtuali

- `venv`: modulo standard da Python 3.3+.
- `virtualenv`: libreria esterna, compatibile anche con Python 2.
- `pipenv`: gestore di ambienti e dipendenze (combina pip e virtualenv).
- `poetry`: gestore moderno per ambienti, dipendenze e packaging.
- `conda`: gestore di ambienti e pacchetti per Python e altri linguaggi (Anaconda/Miniconda).

## Creazione di un ambiente virtuale con venv (standard)

```
python -m venv nome_ambiente
```

**Nota:** Sostituisci `python` con `python3` se necessario.

## Attivazione dell'ambiente virtuale

- **Windows:**

```
nome_ambiente\Scripts\activate
```

- **Linux/Mac:**

```
source nome_ambiente/bin/activate
```

Dopo l'attivazione, il prompt mostra il nome dell'ambiente.

### Disattivazione dell'ambiente virtuale

```
deactivate
```

### Installazione di pacchetti nell'ambiente virtuale

Dopo l'attivazione, usa `pip` normalmente:

```
pip install nome_pacchetto
```

I pacchetti saranno installati solo nell'ambiente attivo.

### Eliminazione di un ambiente virtuale

Basta cancellare la cartella dell'ambiente:

```
rm -rf nome_ambiente
```

### Verifica dell'ambiente attivo

```
which python      # Linux/Mac
where python      # Windows
```

Il percorso deve puntare all'interno della cartella dell'ambiente.

### Gestione delle dipendenze

- `pip freeze > requirements.txt`: salva le dipendenze.
- `pip install -r requirements.txt`: installa le dipendenze.

### Uso di virtualenv (alternativa a venv)

```
pip install virtualenv
virtualenv nome_ambiente
# Attivazione come per venv
```

`virtualenv` offre più opzioni e compatibilità con Python 2.

### Uso di pipenv

```
pip install pipenv
pipenv install nome_pacchetto
pipenv shell
```

`Pipenv` crea e gestisce automaticamente l'ambiente virtuale e i file `Pipfile`/`Pipfile.lock`.

## Uso di poetry

```
pip install poetry
poetry new mio_progetto
cd mio_progetto
poetry add nome_pacchetto
poetry shell
```

`Poetry` gestisce ambienti, dipendenze e packaging tramite `pyproject.toml`.

## Uso di conda (Anaconda/Miniconda)

```
conda create -n mio_ambiente python=3.11
conda activate mio_ambiente
conda install nome_pacchetto
```

`Conda` gestisce ambienti e pacchetti (anche non Python).

## Ambienti virtuali e IDE

La maggior parte degli IDE (PyCharm, VSCode, ecc.) rileva e permette di selezionare l'ambiente virtuale per il progetto.

## Ambienti virtuali e versioni di Python

Puoi creare ambienti virtuali con versioni diverse di Python, specificando il percorso dell'interprete:

```
python3.10 -m venv venv310
```

## Ambienti virtuali e progetti multipli

Ogni progetto dovrebbe avere il proprio ambiente virtuale, tipicamente nella cartella `venv/` o `.venv/`.

## Ambienti virtuali e repository

- Non includere la cartella dell'ambiente virtuale nel controllo versione (aggiungi `venv/` a `.gitignore`).
- Condividi solo `requirements.txt`, `Pipfile.lock` o `pyproject.toml`.

## Ambienti virtuali e script di attivazione

Puoi automatizzare l'attivazione con script o configurazioni IDE.

## Ambienti virtuali e Jupyter Notebook

Per usare un ambiente virtuale in Jupyter:

```
pip install ipykernel
python -m ipykernel install --user --name nome_ambiente
```

Poi seleziona il kernel corrispondente in Jupyter.

## Ambienti virtuali e sicurezza

- Installa pacchetti solo da fonti affidabili.
- Aggiorna regolarmente le dipendenze.
- Usa ambienti virtuali per evitare conflitti e rischi di sicurezza.

## Ambienti virtuali e automazione

Per automatizzare la creazione e gestione degli ambienti, puoi usare `Makefile`, `tox`, `nox`, script shell, ecc.

## Ambienti virtuali e path

L'attivazione modifica le variabili d'ambiente (`PATH`, `VIRTUAL_ENV`) per puntare all'ambiente attivo.

## Ambienti virtuali e limitazioni

- Non isolano le variabili d'ambiente di sistema.
- Non isolano processi o file di configurazione esterni.
- Non sono adatti per deployment in produzione (usa container, Docker, ecc.).

## Best practice

- Crea sempre un ambiente virtuale per ogni progetto.
- Non committare la cartella dell'ambiente.
- Usa file di dipendenze (`requirements.txt`, `Pipfile.lock`, `pyproject.toml`).
- Documenta come creare/attivare l'ambiente nel README.
- Aggiorna e verifica regolarmente le dipendenze.

### Documentazione ufficiale

- `https://docs.python.org/3/library/venv.html`
- `https://virtualenv.pypa.io/`
- `https://pipenv.pypa.io/`
- `https://python-poetry.org/`
- `https://docs.conda.io/`

### Conclusioni

La gestione degli ambienti virtuali è una pratica essenziale nello sviluppo Python moderno. Permette di lavorare in modo isolato, sicuro e riproducibile, facilitando la collaborazione e la distribuzione dei progetti. La padronanza degli strumenti per la creazione, attivazione e gestione degli ambienti virtuali è fondamentale per ogni sviluppatore Python.

## 30 Debugging e gestione degli errori

Il **debugging** e la **gestione degli errori** sono aspetti fondamentali dello sviluppo software. Permettono di individuare, comprendere e correggere i problemi nei programmi Python, migliorando la qualità, l'affidabilità e la manutenibilità del codice.

### Cos'è il debugging?

Il debugging è il processo di individuazione e correzione degli errori (bug) nel codice. Un bug può essere un errore di sintassi, logica, runtime o di comportamento inatteso.

### Tipi di errori in Python

- **Errori di sintassi** (*SyntaxError*): errori nella struttura del codice (parentesi mancanti, errori di indentazione, ecc.).
- **Errori di runtime** (*Exception*): errori che si verificano durante l'esecuzione (divisione per zero, file non trovato, ecc.).
- **Errori logici**: il programma si esegue senza errori, ma il risultato non è quello atteso.

### Strumenti di debugging in Python

- **Stampa di variabili**: uso di `print()` per visualizzare valori intermedi.
- **Traceback**: messaggio di errore che mostra la sequenza delle chiamate che hanno portato all'errore.

- **Debugger interattivo** (`pdb`): permette di eseguire il codice passo-passo, impostare breakpoint, ispezionare variabili.
- **Debugging negli IDE**: strumenti grafici integrati in PyCharm, VSCode, Thonny, ecc.
- **Logging**: registrazione di messaggi, errori e informazioni tramite il modulo `logging`.
- **Test automatici**: uso di `unittest`, `pytest`, ecc. per individuare regressioni e bug.
- **Profiling**: analisi delle prestazioni e individuazione di colli di bottiglia.

## Uso di print() per il debugging

Il metodo più semplice: inserire `print()` in punti strategici del codice per controllare valori e flusso di esecuzione.

```
def somma(a, b):
    print(f"a={a}, b={b}")
    return a + b
```

**Limiti:** poco scalabile, va rimosso dal codice finale.

## Traceback e messaggi di errore

Quando si verifica un'eccezione non gestita, Python stampa un traceback che mostra:

- Il tipo di errore (es. `ZeroDivisionError`)
- Il messaggio di errore
- La sequenza delle chiamate (stack trace)
- Il file e la riga dove si è verificato l'errore

**Esempio:**

```
Traceback (most recent call last):
  File "main.py", line 2, in <module>
    print(1 / 0)
ZeroDivisionError: division by zero
```

## Il modulo pdb (Python Debugger)

`pdb` è il debugger interattivo standard di Python.

- `import pdb; pdb.set_trace()`: inserisce un breakpoint nel codice.
- `python -m pdb script.py`: esegue uno script in modalità debug.

**Comandi principali di pdb:**

- `n` (*next*): esegui la prossima riga
- `c` (*continue*): continua fino al prossimo breakpoint
- `l` (*list*): mostra il codice sorgente
- `b` (*break*): imposta un breakpoint
- `p` (*print*): stampa il valore di una variabile
- `q` (*quit*): esci dal debugger

**Esempio:**

```
import pdb

def dividi(a, b):
    pdb.set_trace()
    return a / b

dividi(10, 0)
```

## Debugging negli IDE

La maggior parte degli IDE offre strumenti di debugging grafici:

- Breakpoint visivi
- Esecuzione passo-passo
- Ispezione delle variabili
- Stack trace interattivo
- Watch expressions e valutazione di espressioni

**Consigliato** per progetti complessi.

## Il modulo logging

`logging` permette di registrare messaggi di debug, info, warning, error e critical in modo configurabile (console, file, ecc.).

```
import logging

logging.basicConfig(level=logging.DEBUG)
logging.debug("Messaggio di debug")
logging.info("Informazione")
logging.warning("Attenzione")
logging.error("Errore")
logging.critical("Critico")
```

**Vantaggi:** puoi filtrare i messaggi, salvarli su file, disattivarli in produzione.

## Gestione delle eccezioni

Vedi anche la sezione `try-except`. Gestire le eccezioni permette di:

- Prevenire crash del programma
- Fornire messaggi di errore chiari all'utente
- Loggare errori per analisi successive
- Eseguire azioni di recupero o fallback

**Esempio:**

```
try:
    x = int(input("Numero: "))
except ValueError as e:
    logging.error(f"Errore di conversione: {e}")
    print("Devi inserire un numero intero!")
```

## Eccezioni personalizzate

Puoi definire eccezioni specifiche per il tuo dominio applicativo:

```
class ErroreApplicazione(Exception):
    pass

raise ErroreApplicazione("Errore specifico")
```

## Modulo traceback

Per stampare o salvare il traceback di un'eccezione:

```
import traceback

try:
    1 / 0
except Exception:
    traceback.print_exc()
```

## Test automatici e debugging

Scrivere test automatici (`unittest`, `pytest`) aiuta a individuare bug prima che il codice vada in produzione. I test falliti forniscono informazioni utili per il debugging.

## Profiling e performance

Per individuare colli di bottiglia e ottimizzare il codice:

- `cProfile`, `profile`: moduli per il profiling delle prestazioni
- `timeit`: misura il tempo di esecuzione di piccoli blocchi di codice

### Altri strumenti utili

- `faulthandler`: stampa il traceback anche per errori fatali (es. segmentation fault)
- `warnings`: gestisce e controlla i warning (avvisi non fatali)
- `assert`: verifica condizioni durante lo sviluppo, solleva AssertionError se la condizione è falsa

```
assert x > 0, "x deve essere positivo"
```

### Debugging di codice concorrente

Per il debugging di thread e processi multipli, usa strumenti come:

- `threading.settrace()`, `multiprocessing.set_start_method()`
- Debugger avanzati: `pdb++`, `pydevd`, `remote-pdb`

### Debugging remoto

Puoi eseguire il debug di applicazioni remote (es. server, container) usando debugger remoti come `debugpy`, `pydevd`, `remote-pdb`.

### Debugging di memory leak

Per individuare perdite di memoria:

- `gc`: modulo per la gestione del garbage collector
- `tracemalloc`: traccia l'allocazione della memoria
- Strumenti esterni: `objgraph`, `memory-profiler`, `heapy`

```
import tracemalloc
tracemalloc.start()
# ... codice ...
print(tracemalloc.get_traced_memory())
```

### Debugging di codice C/estensioni

Per bug in moduli C o estensioni, usa strumenti come `gdb`, `valgrind`, o il supporto di `faulthandler`.

## Debugging e best practice

- Scrivi codice semplice e leggibile.
- Usa nomi di variabili chiari.
- Commenta il codice complesso.
- Scrivi test automatici.
- Gestisci sempre le eccezioni.
- Usa il logging invece di print() in produzione.
- Rimuovi codice di debug prima di rilasciare.
- Documenta i bug noti e le soluzioni adottate.

## Debugging e documentazione

Consulta sempre la documentazione ufficiale e le risorse della community (Stack Overflow, GitHub Issues, ecc.) per soluzioni a problemi comuni.

## Debugging e versionamento

Usa un sistema di controllo versione (es. git) per poter tornare a versioni funzionanti del codice e isolare le modifiche che introducono bug.

## Debugging e ambienti di test

Esegui il codice in ambienti di test o staging prima di metterlo in produzione.

## Debugging e Continuous Integration

Integra strumenti di test e analisi statica (linting, type checking) nelle pipeline CI/CD per individuare bug automaticamente.

## Risorse utili

- `https://docs.python.org/3/library/pdb.html`
- `https://docs.python.org/3/library/logging.html`
- `https://docs.python.org/3/library/traceback.html`
- `https://realpython.com/python-debugging-pdb/`
- `https://docs.python.org/3/library/unittest.html`

## Conclusioni

Il debugging e la gestione degli errori sono competenze essenziali per ogni sviluppatore Python. Saper individuare, analizzare e risolvere i bug, usare gli strumenti giusti e adottare le best practice permette di scrivere codice più robusto, affidabile e professionale.