

Pausa caffè



Torniamo alle 15:35

Python



Giorno 1

Marius Minia

**Buongiorno,
sono Marius Minia**



Python



Operators	Operation	Example
<code>**</code>	Exponent	<code>`2 ** 3 = 8`</code>
<code>%</code>	Modulus/Remainder	<code>`22 % 8 = 6`</code>
<code>//</code>	Integer division	<code>`22 // 8 = 2`</code>
<code>/</code>	Division	<code>`22 / 8 = 2.75`</code>
<code>*</code>	Multiplication	<code>`3 * 3 = 9`</code>
<code>-</code>	Subtraction	<code>`5 - 2 = 3`</code>
<code>+</code>	Addition	<code>`2 + 2 = 4`</code>

```
>>> 2 + 3 * 6  
# 20

>>> (2 + 3) * 6  
# 30

>>> 2 ** 8  
#256

>>> 23 // 7  
# 3

>>> 23 % 7  
# 2

>>> (5 - 1) * ((7 + 1) / (3 - 1))  
# 16.0
```

Operator	Equivalent
<code>`var += 1`</code>	<code>`var = var + 1`</code>
<code>`var -= 1`</code>	<code>`var = var - 1`</code>
<code>`var *= 1`</code>	<code>`var = var * 1`</code>
<code>`var /= 1`</code>	<code>`var = var / 1`</code>
<code>`var //= 1`</code>	<code>`var = var // 1`</code>
<code>`var %= 1`</code>	<code>`var = var % 1`</code>
<code>`var **= 1`</code>	<code>`var = var ** 1`</code>

```

>>> greeting = 'Hello'
>>> greeting += ' world!'
>>> greeting
# 'Hello world!'

>>> number = 1
>>> number += 1
>>> number
# 2

>>> my_list = ['item']
>>> my_list *= 3
>>> my_list
# ['item', 'item', 'item']

```

Walrus Operator

operatore di assegnazione con espressione

Esempio classico senza walrus

```
python

n = len(my_list)
if n > 10:
    print(f"La lista ha {n} elementi")
```

Con il walrus operator

```
python

if (n := len(my_list)) > 10:
    print(f"La lista ha {n} elementi")
```

In questo modo eviti di calcolare `len(my_list)` due volte o di scrivere una riga separata per salvare il valore

Walrus Operator #2

Un altro esempio



```
while (line := input("Scrivi qualcosa: ")) != "exit":  
    print("Hai scritto:", line)
```

Qui assegni direttamente la stringa letta a **line** dentro la condizione del **while**.

Data Type

```
# Numbers
age = 25                      # int
price = 19.99                   # float
coordinate = 2 + 3j             # complex

# Text
name = "Alice"                 # str

# Boolean
is_student = True               # bool

# None
result = None                  # NoneType

# Collections
scores = [85, 92, 78]           # list
person = {'name': 'Bob', 'age': 30} # dict
coordinates = (10, 20)          # tuple
unique_ids = {1, 2, 3}          # set
```

Liste

Cosa sono

- Collezione **ordinata** e **mutabile** di elementi
- Definite con parentesi quadre `[]`
- Possono contenere tipi diversi (numeri, stringhe, altre liste...)

Caratteristiche principali

- **Ordinata**: gli elementi mantengono la posizione di inserimento
- **Mutabile**: puoi aggiungere, modificare e rimuovere elementi
- Permettono duplicati



```
# Creazione di liste
lista_vuota = []
lista_numeri = [1, 2, 3, 4]
lista_mista = [1, "ciao", 3.14]

# Accesso
print(lista_mista[1])    # "ciao"

# Modifica
lista_numeri[0] = 100    # [100, 2, 3, 4]

# Aggiungere e rimuovere
lista_numeri.append(5)      # [100, 2, 3, 4, 5]
lista_numeri.remove(2)      # [100, 3, 4, 5]
```

Liste

Metodi

```
● ● ●

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

# 1. Aggiungere elementi
numeri.append(9)
print(numeri)  # [3, 1, 4, 1, 5, 9]

numeri.extend([2, 6])
print(numeri)  # [3, 1, 4, 1, 5, 9, 2, 6]

numeri.insert(2, 99)
print(numeri)  # [3, 1, 99, 4, 1, 5, 9, 2, 6]

# 2. Rimuovere elementi
numeri.remove(1)
print(numeri)  # elimina la prima occorrenza di 1

ultimo = numeri.pop()
print(ultimo)  # 6
print(numeri)

numeri.clear()
print(numeri)  # []

# 3. Cercare e contare
frutta = ["mela", "pera", "mela", "banana"]
print(frutta.index("mela"))  # 0
print(frutta.count("mela"))  # 2

# 4. Ordinare e invertire
valori = [10, 3, 7, 1]
valori.sort()
print(valori)  # [1, 3, 7, 10]

valori.sort(reverse=True)
print(valori)  # [10, 7, 3, 1]

valori.reverse()
print(valori)  # [1, 3, 7, 10] (invertito)

# 5. Copiare
copia = valori.copy()
print(copia)  # [1, 3, 7, 10]
```

Tuple

Cosa sono

- Collezione **ordinata** e **immutabile** di elementi
- Definite con parentesi tonde `()`
- Possono contenere tipi diversi (numeri, stringhe, liste, altre tuple...)

Caratteristiche principali

- **Ordinata**: gli elementi hanno una posizione precisa
- **Immutabile**: non puoi modificare, aggiungere o rimuovere elementi
- Possono essere usate come **chiavi di dizionari** (a differenza delle liste)

Vantaggi

- Più leggere e veloci delle liste
- Sicurezza: non puoi alterarne il contenuto
- Perfette per dati costanti o "record" (es. coordinate, date, impostazioni)



```
# Creazione di tuple
tupla_vuota = ()
tupla_uno = (42,)
tupla_mista = (1, "ciao", 3.14)

# Accesso
print(tupla_mista[1]) # "ciao"

# Slicing
print(tupla_mista[:2]) # (1,
"ciao")
```

Tuple



```
tupla = (1, 2, 3, 2, 4, 2)

# Conta quante volte compare un elemento
print(tupla.count(2)) # 3

# Restituisce la posizione della prima occorrenza
print(tupla.index(3)) # 2
```

Set

Cosa sono

- Collezione **non ordinata** e **mutabile** di elementi **unici**
- Definiti con parentesi graffe `{ }` o con `set()`
- Non permettono duplicati

Caratteristiche principali

- **Non ordinati:** niente indici, non c'è garanzia sull'ordine
- **Elementi unici:** i duplicati vengono rimossi automaticamente
- **Mutabili:** puoi aggiungere o rimuovere elementi
- Possono contenere solo elementi **immutabili** (int, str, tuple...)



```
# Creazione di set
s1 = {1, 2, 3, 3, 4}
print(s1) # {1, 2, 3, 4} (niente
duplicati)
s2 = set([1, 2, 2, 3])
print(s2) # {1, 2, 3}

# Aggiunta e rimozione
s1.add(5)      # {1, 2, 3, 4, 5}
s1.remove(3)   # {1, 2, 4, 5}
```

Set

Operazioni insiemistiche



```
a = {1, 2, 3}
```

```
b = {3, 4, 5}
```

```
print(a | b)    # {1, 2, 3, 4, 5}      (unione)
```

```
print(a & b)    # {3}                      (intersezione)
```

```
print(a - b)    # {1, 2}                  (differenza)
```

```
print(a ^ b)    # {1, 2, 4, 5}      (differenza
```

```
simmetrica)
```

Dizionari

Cosa sono

- Collezione **non ordinata, mutabile e indicizzata**
- Ogni elemento è una **coppia chiave → valore**
- Definiti con parentesi graffe `{chiave: valore}`

Caratteristiche principali

- **Chiavi uniche:** non possono ripetersi
- **Valori qualsiasi:** numeri, stringhe, liste, altri dizionari...
- **Mutabili:** puoi aggiungere, modificare, rimuovere coppie
- **Accesso veloce** ai valori tramite chiave



```
# Creazione
studente = {
    "nome": "Anna",
    "età": 22,
    "corso": "Informatica"
}
```

```
# Accesso
print(studente["nome"]) # Anna
# Aggiungere/modificare
studente["età"] = 23
studente["città"] = "Roma"
```

```
# Rimuovere
del studente["corso"]
```

Dizionari



```
d = {"a": 1, "b": 2, "c": 3}

print(d.keys())      # dict_keys(['a', 'b', 'c'])
print(d.values())    # dict_values([1, 2, 3])
print(d.items())     # dict_items([('a', 1), ('b', 2), ('c', 3)])
d.pop("b")          # rimuove la chiave 'b'
d.update({"d": 4})  # aggiunge {"d": 4}
```

Vantaggi

- Molto usati per rappresentare **dati strutturati**
- Base per **JSON**, API e configurazioni
- Accesso rapido per chiave

Scegliere la struttura adatta

Usare il tipo di dato corretto semplifica il codice e migliora le performance.

Ad esempio, per cercare se un elemento esiste, un set è più efficiente di una lista (lookup in $O(1)$ invece di $O(n)$).

Per accedere a coppie chiave-valore, meglio un dict che due liste parallele.

Inoltre, essere consapevoli della mutabilità evita bug: ad esempio, evitare di usare oggetti mutabili come chiavi di dizionario (perché devono essere hashable, quindi immutabili).

Pitfall comune:

La mutabilità nelle liste.

Se assegniamo una lista a un'altra variabile, entrambe puntano alla stessa lista in memoria.

Modificandola da una parte, vedremo il cambiamento riflesso sull'altra.

```
● ● ●  
config1 = {"tema": "scuro", "lingua": "it"}  
config2 = config1      # config2 fa riferimento allo stesso dict di  
config2["tema"] = "chiaro"  # modifica attraverso config2  
print(config1["tema"])    # scendo su config1, riflette la modifica:  
"chiaro"
```

In questo caso **config1** e **config2** riferiscono lo stesso dizionario. Best practice: se serve una copia indipendente, usare **config1.copy()** (per copie superficiali) o il modulo **copy** per copie profonde, evitando effetti indesiderati.

Stringhe

Creazione e base

```
python  
  
s = " Python è Fantastico! "
```

Conversione

- `.upper()` → maiuscolo → "PYTHON È FANTASTICO!"
- `.lower()` → minuscolo → "python è fantastico!"
- `.title()` → iniziali maiuscole → "Python È Fantastico!"
- `.capitalize()` → prima maiuscola → "Python è fantastico!"

✂ Spazi e sostituzioni

- `.strip()` → rimuove spazi ai lati → "Python è Fantastico!"
- `.replace("Fantastico", "potente")` → sostituisce → "Python è potente!"

Ricerca

- `.find("è")` → posizione della prima occorrenza
- `.count("o")` → quante volte compare "o"

🔗 Suddivisione e unione

```
python  
  
parole = s.split()      # ['Python', 'è', 'Fantastico!']  
frase = "-".join(parole) # "Python-è-Fantastico!"
```

✓ Verifiche

- `.startswith("Py")` → True
- `.endswith("!")` → True
- `.isdigit()` → True se solo cifre
- `.isalpha()` → True se solo lettere

✳ Formattazione

```
python  
  
nome = "Anna"  
eta = 25  
print(f"Ciao {nome}, hai {eta} anni")  
# "Ciao Anna, hai 25 anni"
```

Stringhe

◆ f-string (Python 3.6+)

python

```
nome = "Anna"  
msg = f"Ciao {nome}, oggi è {2025}!"  
print(msg) # "Ciao Anna, oggi è 2025!"
```

- Più leggibili e veloci
- Puoi usare espressioni direttamente:

python

```
print(f"3 + 5 = {3+5}") # 3 + 5 = 8
```

◆ Metodo `.format()`

python

```
s = "Ciao {0}, hai {1} messaggi".format("Anna", 5)  
# "Ciao Anna, hai 5 messaggi"  
  
s = "Ciao {nome}, hai {n} messaggi".format(nome="Anna", n=5)  
# "Ciao Anna, hai 5 messaggi"
```

◆ `format_map()` con dizionario

python

```
d = {"nome": "Anna", "n": 5}  
s = "Ciao {nome}, hai {n} messaggi".format_map(d)  
# "Ciao Anna, hai 5 messaggi"
```

◆ Gestione chiavi mancanti

Se una chiave non esiste → `KeyError`.

Possiamo usare un **dizionario personalizzato** con `__missing__`:

python

```
class Default(dict):  
    def __missing__(self, key):  
        return "?"
```

```
d = {"nome": "Anna"}  
s = "Ciao {nome}, hai {n} messaggi".format_map(Default(d))  
# "Ciao Anna, hai ? messaggi"
```

✓ Quando usarla

- f-string → quando i valori sono già noti nel codice
- `.format()` / `format_map()` → quando i valori arrivano da dizionari o input dinamici
- `dict` personalizzato → per **default sicuri** invece di errori

print()

- Stampa a schermo un valore o più valori
- Accetta argomenti multipli, separati da virgola

python

```
print("Ciao", "mondo!")      # Ciao mondo!
print(1, 2, 3, sep=" - ")    # 1 - 2 - 3
print("Fine", end="!\n")      # Fine!
```

type()

- Restituisce il tipo di un oggetto

python

```
print(type(42))      # <class 'int'>
print(type("ciao"))   # <class 'str'>
```

len()

- Restituisce la lunghezza (numero di elementi)

python

```
print(len("Python"))   # 6
print(len([1, 2, 3])) # 3
```

input()

- Legge un valore da tastiera (sempre come stringa)

python

```
nome = input("Come ti chiami? ")
print("Ciao", nome)
```

str(), int(), float()

- Conversione di tipi

python

```
numero = "42"
print(int(numero) + 1)    # 43
```

🔍 Ispezione e conversione

- `print(x, ...)` → stampa valori
- `type(x)` → tipo dell'oggetto
- `id(x)` → indirizzo in memoria
- `str(x)`, `int(x)`, `float(x)`, `bool(x)` → conversioni

12 34 Numeri

- `abs(x)` → valore assoluto
- `round(x, ndigits)` → arrotondamento
- `pow(x, y)` → potenza x^y
- `max(iterable)`, `min(iterable)` → massimo e minimo
- `sum(iterable)` → somma

↙ Sequenze

- `len(x)` → lunghezza
- `sorted(x)` → ritorna lista ordinata (non in place)
- `reversed(x)` → iteratore invertito
- `enumerate(x)` → coppie `(indice, valore)`
- `zip(a, b, ...)` → combina più sequenze

⌚ Iterazione

- `range(start, stop, step)` → intervallo numerico
- `map(func, iterable)` → applica funzione a ogni elemento
- `filter(func, iterable)` → filtra elementi
- `any(iterable)` → True se almeno un elemento è vero
- `all(iterable)` → True se tutti gli elementi sono veri

📁 Input/Output

- `input(prompt)` → leggi da tastiera
- `open(file, mode)` → apre un file

Comparison Operators

Operator	Meaning
<code>'=='</code>	Equal to
<code>'!='</code>	Not equal to
<code>'<'</code>	Less than
<code>'>'</code>	Greater Than
<code>'<='</code>	Less than or Equal to
<code>'>='</code>	Greater than or Equal to

```
>>> 42 == 42
True

>>> 40 == 42
False

>>> 'hello' == 'hello'
True

>>> 'hello' == 'Hello'
False

>>> 'dog' != 'cat'
True

>>> 42 == 42.0
True

>>> 42 == '42'
False
```

Sintassi base

If

```
python

if condizione:
    # blocco eseguito se la condizione è vera
```

Con else

```
python

x = 5
if x > 10:
    print("Maggiore di 10")
else:
    print("Minore o uguale a 10")
```

Con elif (else if)

```
python

x = 7
if x > 10:
    print("Grande")
elif x == 10:
    print("Uguale a 10")
else:
    print("Piccolo")
```

Condizioni multiple

- Operatori logici: and , or , not

```
python

eta = 20
patente = True

if eta >= 18 and patente:
    print("Puoi guidare")
```

Sintassi compatta (if inline)

```
python

x = 3
parita = "pari" if x % 2 == 0 else "dispari"
print(parita) # "dispari"
```

Switch

1. Con i dizionari (metodo classico)

```
python

def azione(comando):
    switch = {
        "start": "Avvio del programma",
        "stop": "Arresto del programma",
        "pause": "Pausa"
    }
    return switch.get(comando, "Comando sconosciuto")

print(azione("start")) # Avvio del programma
print(azione("exit")) # Comando sconosciuto
```

💡 `dict.get()` permette di dare un valore di default se la chiave non esiste.

2. Con `match-case` (introdotto in Python 3.10)

Questo è il vero equivalente del `switch` moderno.

```
python

comando = "start"

match comando:
    case "start":
        print("Avvio del programma")
    case "stop":
        print("Arresto del programma")
    case "pause":
        print("Pausa")
    case _:
        print("Comando sconosciuto")
```

💡 `case _:` è il “default” (corrisponde al `else` del `if`).

For



```
# Lista
frutta = ["mela", "pera", "banana"]
for item in frutta:
    print(item)

# Stringa
for lettera in "ciao":
    print(lettera)

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

range()

```
# range(start, stop, step)
>>> for i in range(0, 10, 2):
...     print(i)
...
# 0
# 2
# 4
# 6
# 8
```

For con indice

```
animali = ["cane", "gatto", "pesce"]
for i, animale in enumerate(animali):
    print(i, animale)
# 0 cane, 1 gatto, 2 pesce
```

Iterare su un dizionario

```
studente = {"nome": "Anna", "eta": 22,  
"corso": "Informatica"}  
for chiave, valore in studente.items():  
    print(chiave, "->", valore)
```

While

```
● ● ●  
x = 0  
while x < 5:  
    print(x)  
    x += 1  
# stampa 0,1,2,3,4
```

While con break e continue

```
● ● ●

# break → interrompe il ciclo
x = 0
while True:
    if x == 3:
        break
    print(x)
    x += 1

# continue → salta alla prossima iterazione
x = 0
while x < 5:
    x += 1
    if x % 2 == 0:
        continue
    print(x) # stampa solo numeri dispari
```

Funzioni

Cosa sono

- Blocchi di codice riutilizzabili
- Ricevono **parametri** e possono restituire un **valore**
- Definite con la parola chiave `def`

Sintassi base

```
python

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

saluta() # chiamata
```

Parametri e ritorno

```
python

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

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

Valori di default

```
python

def presentazione(nome, lingua="it"):
    if lingua == "it":
        print(f"Ciao {nome}")
    else:
        print(f"Hello {nome}")

presentazione("Anna")           # Ciao Anna
presentazione("Anna", "en")     # Hello Anna
```

Argomenti variabili

```
python

def totale(*args):
    return sum(args)

print(totale(1, 2, 3, 4)) # 10

def profilo(**kwargs):
    print(kwargs)

profilo(nome="Anna", età=22) # {'nome': 'Anna', 'età': 22}
```

Funzioni anonime (lambda)

```
python

quadrato = lambda x: x**2
print(quadrato(5)) # 25
```

Funzioni



```
def aggiungi_elemento(elemento, lista=[]):
    lista.append(elemento)
    return lista

# Uso della funzione
print(aggiungi_elemento(1)) # Output: [1]
print(aggiungi_elemento(2)) # Ci si potrebbe aspettare [2], invece...
print(aggiungi_elemento(3)) # ...l'elenco cresce cumulativamente!
```

Parametri di default e mutabilità: attenzione ai valori di default mutabili.

In Python, i valori di default dei parametri sono valutati una volta sola quando la funzione viene definita, non a ogni chiamata. Ciò significa che usare un oggetto mutabile come default può portare a comportamenti imprevisti.

L'output reale sarà [1], poi [1, 2], poi [1, 2, 3] perché la lista di default [] viene creata una volta e riutilizzata tra le chiamate.

Funzioni

Best practice: usare `None` come default e all'interno della funzione assegnare una nuova lista se il parametro è `None`:



```
def aggiungi_elemento(elemento, lista=None):
    if lista is None:
        lista = []
    lista.append(elemento)
    return lista
```

In questo modo ogni chiamata senza lista esplicita utilizza una nuova lista vuota.

Funzioni

`*args` e `**kwargs`: in funzioni avanzate possiamo usare *argomenti variabili*. `*args` raccoglie argomenti posizionali extra in una tupla, `**kwargs` raccoglie argomenti nominali extra in un dizionario. Questi sono utili per funzioni che devono accettare un numero variabile di parametri (ad esempio una funzione di logging personalizzato che accetta qualsiasi numero di dettagli). Esempio:

python

```
def stampa_report(nome, *valori, **opzioni):
    """Stampa un report semplice con i valori forniti."""
    print(f"Report: {nome}")
    for v in valori:
        print(f"- {v}")
    if opzioni.get("footer"):
        print(opzioni["footer"])

# Chiamata della funzione con vari argomenti
stampa_report("Vendite Q1", 1500, 2000, 1700, footer="Fine Report")
```

Qui `*valori` permetterà di passare qualsiasi numero di valori di vendita, e `**opzioni` può gestire parametri extra come un footer.

Funzioni

Valore di ritorno: è buona norma far restituire alle funzioni il risultato del loro calcolo invece di stamparlo direttamente, a meno che la funzione stessa non sia pensata per l'output. Ad esempio, una funzione

`calcola_media()` dovrebbe restituire la media calcolata, lasciando a chi la chiama la decisione di come usarla (stamparla, salvarla, ecc.). Questo aumenta la riusabilità (lo stesso calcolo può essere usato in contesti diversi).

Type hints (annotazioni di tipo): Python permette di annotare i tipi dei parametri e del ritorno delle funzioni. Ad esempio:

python

```
def calcola_media(valori: list[float]) -> float:  
    ...
```

Anche se non obbligatorio (Python non le applica a runtime), è *best practice* in progetti di una certa dimensione aggiungere queste annotazioni. Aiutano strumenti esterni (linters, editor, mypy) a rilevare incompatibilità e migliorano la comprensibilità del codice per gli umani, documentando cosa ci si aspetta come input/output.

PEP 8

Leggibilità e convenzioni (PEP 8)

Una parte essenziale del codice di qualità è aderire a uno stile uniforme. In Python esiste una guida ufficiale, **PEP 8**, che fornisce convenzioni di stile ampiamente accettate. Ecco alcuni punti chiave:

- **Nomi significativi:** Scegliere nomi descrittivi per variabili, funzioni e classi. Ad esempio, invece di `x` o `var1`, usare nomi come `quantita_prodotti` o `calcola_media`. I nomi chiari rendono il codice auto-esplicativo.
- **Naming convention:** usare `snake_case` (tutto minuscolo con underscore) per funzioni e variabili (`calcola_valore_totale`), e il `Came/Case` con iniziali maiuscole per i nomi delle classi (`GestioneClienti`). Le costanti (valori fissi) si scrivono in MAIUSCOLO (`MAX_ITERAZIONI`). Evitare di usare come nomi quelli built-in di Python (ad esempio, non usare `list` o `str` come nomi di variabili).
- **Indentazione:** Python richiede indentazione consistente del codice. La regola standard è usare **4 spazi** per ogni livello di indentazione (meglio non usare tab per evitare confusione). Un rientro corretto migliora la leggibilità e previene errori di sintassi. Esempio:

```
python

# Esempio di indentazione corretta in una struttura di controllo
for cliente in lista_clienti:
    if cliente.attivo:
        invia_email(cliente)
    else:
        registra_log(cliente)
```

In questo esempio ogni blocco (`if/else`) è indentato di 4 spazi all'interno del `for`.

- **Lunghezza delle linee:** Evitare linee di codice eccessivamente lunghe. PEP 8 consiglia di non superare circa **79 caratteri per riga** (peps.python.org), così il codice è visibile senza scroll orizzontale ed è più facile da leggere in affiancamento. Se una linea è troppo lunga, meglio spezzarla su più righe usando parentesi tonde o barre invertite per continuare.
- **Spazi bianchi:** Usare gli spazi con criterio. Ad esempio, mettere uno spazio dopo le virgolette in liste o argomenti di funzione (`funzione(x, y, z)`), e attorno agli operatori (`a = b + c`). Evitare spazi inutili prima di virgolette o all'interno di parentesi. Non lasciare spazi alla fine delle righe. Queste piccole attenzioni rendono il codice più pulito.
- **Commenti e docstring:** I commenti dovrebbero spiegare perché il codice fa qualcosa, se non è ovvio, più che descrivere passo passo cosa fa (che spesso è evidente leggendo il codice stesso). Manteniamoli sintetici e aggiornati: un commento sbagliato è peggio di nessun commento. Per documentare funzioni e classi, usare le **docstring** (stringhe racchiuse in `""" """` all'inizio di funzione/classe) per spiegare lo scopo e l'uso. Ad esempio:

```
python

def calcola_media(valori):
    """Calcola la media aritmetica di una lista di numeri."""
    return sum(valori) / len(valori)
```

Una docstring come quella sopra permette ad altri (o a `help()` in Python) di capire facilmente cosa fa la funzione.

- **Strumenti automatici:** Nello sviluppo reale, possiamo utilizzare **linters** (come `flake8` o `pylint`) e **formatter** automatici (come `black`) per verificare e formattare il codice secondo le convenzioni. Questi strumenti aiutano a mantenere uno stile consistente in team numerosi.

Pattern Pythonici per loop e condizioni

Python incoraggia uno stile di codice più conciso e leggibile rispetto ad altri linguaggi, specialmente nelle iterazioni e nelle condizioni. Alcune best practice in questo ambito:

- **Loop su elementi, non su indici:** invece di iterare su indici numerici per accedere a elementi di una lista, è più *Pythonico* iterare direttamente sugli elementi.

```
python

# Modo meno Pythonico – iterare con indice
colori = ["rosso", "verde", "blu"]
for i in range(len(colori)):
    print(colori[i].upper())

# Modo Pythonico – iterare direttamente sugli elementi
for colore in colori:
    print(colore.upper())
```

Il secondo approccio è più leggibile e meno soggetto a errori (non si rischia di andare fuori indice). Se serve anche l'indice (ad esempio per numerare gli elementi), usare `enumerate`:

```
python

for index, colore in enumerate(colori, start=1):
    print(f"{index} {colore}")
```

- **Compreensioni di lista:** per trasformare o filtrare collezioni, le *list comprehension* rendono il codice compatto e chiaro. Esempio, dato un elenco di numeri, vogliamo i quadrati dei numeri pari:

```
python

numeri = [1, 2, 3, 4, 5, 6]
quadrati_pari = [n**2 for n in numeri if n % 2 == 0]
print(quadrati_pari) # Output: [4, 16, 36]
```

Questa singola linea crea la lista dei quadrati solo per i numeri che soddisfano la condizione (pari). È equivalente a un loop `for` con dentro un `if`, ma più concisa. **Best practice:** usare le comprensioni per semplici operazioni di mapping o filtro. Se però la logica diventa troppo complessa, meglio usare un ciclo normale per mantenere la leggibilità.

- **Utilizzare funzioni built-in:** Python ha molte funzioni integrate che semplificano le operazioni comuni. Ad esempio, per sommare una lista di numeri si può usare `sum(lista)` invece di un loop manuale; per ottenere il massimo `max(lista)` e così via. Queste funzioni sono scritte in C e ottimizzate, quindi spesso più efficienti oltre che più concise. Esempio:

```
python

valori = [10, 8, 22, 5]
totale = sum(valori)           # Somma degli elementi
minimo = min(valori)          # Minimo
media = sum(valori) / len(valori) # Media (usando sum di nuovo)
```

In un contesto di **analisi dati**, queste funzioni accelerate possono far la differenza quando si lavora con grandi dataset.

Uso di `with` e context manager

In ambiente professionale, è frequente dover leggere da o scrivere su file, oppure gestire risorse come connessioni a database. In Python, la best practice per gestire queste operazioni è utilizzare i **context manager** tramite l'istruzione `with`. Questo approccio assicura che le risorse vengano correttamente rilasciate (file chiusi, connessioni terminate, etc.), anche se durante l'operazione avviene un errore.

- **Apertura di file:** Consideriamo un caso semplice di gestione configurazioni: un file JSON contenente parametri. Possiamo leggerlo così:

```
python
import json

# Apro e leggo un file di configurazione JSON in modo sicuro
with open("config.json", mode="r", encoding="utf-8") as f:
    config_data = json.load(f) # carica il contenuto JSON in un dizionario

# Qui il file è già stato chiuso automaticamente dall'uscita dal blocco with
valore = config_data.get("chiave_interessante")
```

All'uscita dal blocco `with`, il file viene **automaticamente chiuso**, indipendentemente dal fatto che la lettura abbia avuto successo o meno. Senza `with`, avremmo dovuto chiamare manualmente `f.close()`, col rischio di dimenticarlo (soprattutto se un'eccezione interrompe il flusso).

- **Altri context manager:** `with` non è solo per i file. Molte librerie definiscono context manager per gestire risorse. Ad esempio, in gestione database si può usare un context manager per le transazioni; nel threading per acquisire e rilasciare lock; in automazione, potrebbe esistere un context manager per temporaneamente cambiare directory e poi tornare indietro. Quando una libreria fornisce un context manager, è indice che è la strada preferibile per gestire quella risorsa.
- **Creare context manager personalizzati** (cenno avanzato): in Python possiamo definire le nostre classi come context manager implementando i metodi speciali `__enter__` e `__exit__`. Questo è utile in scenari avanzati di *automazione* o gestione risorse proprie dell'applicazione, ma ne parleremo eventualmente più avanti nel corso.

Esempio realistico: immaginiamo un semplice script di automazione che legga un elenco di task da un file di testo e li esegua. Usando `with open(...)` as `f`: leggiamo le righe in modo sicuro. Possiamo poi processarle sapendo che il file è già stato chiuso correttamente dal context manager. Inoltre, se il file non esiste, possiamo gestire l'eccezione con un `try/except` (vedremo tra poco la gestione errori), ma comunque il context manager garantisce il rilascio di risorse.

Lettura e Scrittura su file

◆ Apertura file

```
python

f = open("dati.txt", "r", encoding="utf-8") # modalità lettura
contenuto = f.read()
f.close()
```

⚠ Ricorda sempre di chiudere il file dopo l'uso.

◆ Uso di `with` (più sicuro)

```
python

with open("dati.txt", "r", encoding="utf-8") as f:
    for riga in f:
        print(riga.strip()) # .strip() rimuove \n
```

👉 `with` chiude il file automaticamente.

◆ Modalità di apertura

- `"r"` → lettura
- `"w"` → scrittura (sovrascrive)
- `"a"` → append (aggiunge in fondo)
- `"x"` → crea nuovo file, errore se esiste

◆ Scrittura

```
python

with open("output.txt", "w", encoding="utf-8") as f:
    f.write("Prima riga\n")
    f.write("Seconda riga\n")
```

◆ Lettura di tutte le righe

```
python

with open("dati.txt", "r", encoding="utf-8") as f:
    righe = f.readlines()
print(righe) # lista di stringhe
```

✓ Esempio pratico (Log Levels)

```
python

with open("log.txt", "r", encoding="utf-8") as f:
    count = {"INFO": 0, "WARN": 0, "ERROR": 0}
    for line in f:
        parola = line.strip().split()[0].upper()
        if parola in count:
            count[parola] += 1
print(count)
```

Python



Giorno 2

Marius Minia

Python — Fondamentali (Giorno 1)

◆ Tipi di Dato

- `int, float, bool, str` → tipi base
- `list` → mutabile, ordinata, duplicati ammessi
- `tuple` → immutabile, ordinata
- `set` → mutabile, non ordinata, elementi unici
- `dict` → chiavi uniche → valori

◆ Stringhe

- Metodi principali: `.strip()`, `.lower()`, `.upper()`, `.replace()`, `.split()`, `.join()`
- F-string: `f"Ciao {nome}"`

◆ Strutture di controllo

- `if / elif / else` → ramificazioni
- `match / case` → switch moderno (da Python 3.10)
- `for` → iterazione su sequenze (`for x in lista`)
- `while` → ciclo finché la condizione è vera
- `break / continue` → gestione flusso

◆ Funzioni

- Definizione: `def nome_funzione(parametri): ...`
- Parametri di default: attenzione ai **mutabili**
- Argomenti variabili: `*args`, `**kwargs`
- Funzioni anonime: `lambda`

◆ Concetti speciali

- **Walrus operator** (`:=`) → assegna dentro un'espressione
- **Slicing**: `seq[start:stop:step]` → sottosequenze, indici negativi ammessi

◆ Best practices

- Copie indipendenti: `.copy()` o `copy.deepcopy`
- Seguire **PEP 8** per leggibilità e coerenza
- Usare `with open(...)` per i file (chiusura automatica)

Gestione degli errori

◆ Eccezioni

Quando qualcosa va storto, Python genera un'**eccezione**:

```
python

print(10 / 0)    # ZeroDivisionError
x = int("ciao") # ValueError
```

◆ Try / Except

```
python

try:
    numero = int("123")
    print("Conversione riuscita:", numero)
except ValueError:
    print("Errore: non è un numero valido")
```

◆ Catch multipli

```
python

try:
    x = 10 / 0
except ZeroDivisionError:
    print("Divisione per zero!")
except Exception as e:
    print("Errore generico:", e)
```

◆ Else e Finally

```
python

try:
    f = open("dati.txt")
except FileNotFoundError:
    print("File non trovato")
else:
    print("File aperto correttamente")
finally:
    print("Operazione conclusa (successo o errore)")
```

◆ Uso pratico nei nostri esercizi

- **Esercizio log levels** → file potrebbe non esistere → `FileNotFoundException`
- **Conversioni con int/float** → `ValueError` se input non valido
- **Dizionari** → evitare `KeyError` usando `.get()` o `defaultdict`

Gestione degli errori

◆ IndexError

Accedi a un indice che non esiste in una lista.

```
python  
  
numeri = [1, 2, 3]  
print(numeri[5]) # IndexError
```

◆ TypeError

Usi un tipo sbagliato in un'operazione.

```
python  
  
print("ciao" + 5) # TypeError
```

◆ KeyError

Cerchi una chiave inesistente in un dizionario.

```
python  
  
d = {"nome": "Anna"}  
print(d["eta"]) # KeyError
```

👉 Soluzione: usare `d.get("eta", "sconosciuta")`

◆ ValueError

Conversione o operazione con valore non valido.

```
python  
  
int("ciao") # ValueError
```

◆ ZeroDivisionError

Divisione per zero.

```
python  
  
print(10 / 0) # ZeroDivisionError
```

✓ Consigli

- Leggere bene il messaggio dell'errore
- Usare `try/except` solo quando serve
- Prevenire con controlli (`if`, `in`, `len`)

```
● ● ●

# conteggio accessi

results = {}
results_filtered = {}
for nome, valore in log:
    if nome not in results:
        results[nome] = 0
    if valore == 200:
        results[nome] += 1
for nome, valore in results.items():
    if valore != 0:
        results_filtered[nome] = valore
return(results_filtered)
```

```
● ● ●

# La variabile results_filtered è superflua
#puoi filtrare direttamente nel return usando una dictionary comprehension

# Non è necessario usare le parentesi tonde in return(results_filtered)
# puoi semplicemente scrivere return results_filtered.

results = {}
for nome, valore in log:
    if nome not in results:
        results[nome] = 0
    if valore == 200:
        results[nome] += 1
return {nome: count for nome, count in results.items() if count != 0}
# raise NotImplementedError
```

```
#ordina task per priorità

# Dizionario per convertire le priorità in numeri
priorita_valore = {
    "critica": 4,
    "alta": 3,
    "media": 2,
    "bassa": 1
}

# Lunghezza della lista
n = len(task)

# Bubble sort
for i in range(n):
    for j in range(0, n - i - 1):
        titolo1, etichetta1 = task[j]
        titolo2, etichetta2 = task[j + 1]

        # Valore numerico delle priorità
        val1 = priorita_valore.get(etichetta1, 0)
        val2 = priorita_valore.get(etichetta2, 0)

        # Confronto: prima priorità decrescente, poi titolo alfabetico
        if val1 < val2 or (val1 == val2 and titolo1 > titolo2):
            # Scambia i task
            task[j], task[j + 1] = task[j + 1], task[j]

return task
```

```
# Usa sorted con una key che ordina per priorità decrescente e poi per titolo alfabetico
return sorted(
    task,
    key=lambda x: (-punteggio_priorita(x[1]), x[0])
)
```

Schema riassuntivo

- **Uso base**

```
python
```

```
numeri = [5, 2, 9, 1]
ordinati = sorted(numeri)      # [1, 2, 5, 9]
```

- **Ordine inverso**

```
python
```

```
sorted(numeri, reverse=True)  # [9, 5, 2, 1]
```

- **Ordinamento per chiave (key)**

```
python
```

```
parole = ["cane", "Gatto", "elefante"]
sorted(parole, key=str.lower)  # ['cane', 'elefante', 'Gatto']
```

- **Ordina lista di tuple o dict**

```
python
```

```
dati = [("Luca", 3), ("Anna", 5), ("Paolo", 2)]
sorted(dati, key=lambda x: x[1])  # per il secondo elemento
# [('Paolo', 2), ('Luca', 3), ('Anna', 5)]
```

- **Differenza con `.sort()`**

- `sorted()` → ritorna una nuova lista
- `.sort()` → ordina **in place** (modifica la lista originale)

Moduli e Pacchetti

◆ Cos'è un modulo?

- Un **file Python** (`.py`) che contiene funzioni, classi, variabili.
- Serve per **riutilizzare il codice**.

```
python

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

```
python

# file principale
import matematica
print(matematica.somma(2, 3)) # 5
```

◆ Importare da un modulo

```
python

from matematica import somma
print(somma(4, 6)) # 10
```

👉 Puoi anche dare un alias:

```
python

import matematica as m
print(m.somma(1, 2))
```

◆ Cos'è un pacchetto?

- Una **cartella** che contiene moduli + un file speciale `__init__.py`
- Serve per organizzare moduli correlati

markdown

```
mio_pacchetto/
|
└── __init__.py
    ├── matematica.py
    └── geometria.py
```

Uso:

```
python

from mio_pacchetto.matematica import somma
```

◆ Moduli della libreria standard

Python ha centinaia di moduli pronti:

- `math` → funzioni matematiche (`sqrt`, `pi`, `factorial`)
- `random` → numeri casuali
- `os` → interazione con il sistema operativo
- `datetime` → date e orari
- `collections` → strutture dati avanzate (`Counter`, `defaultdict`)

```
python

import math, random
print(math.sqrt(16))          # 4.0
print(random.randint(1,10))    # numero casuale
```

Virtual Environment

◆ Cos'è

- Un **ambiente isolato** dove installi librerie senza toccare quelle di sistema
- Ogni progetto può avere le proprie dipendenze e versioni di pacchetti

◆ Creazione

Da terminale, nella cartella del progetto:

```
bash  
  
python -m venv venv
```

👉 Questo crea una cartella `venv/` con tutto l'ambiente virtuale

◆ Attivazione

- Linux/Mac

```
bash  
  
source venv/bin/activate
```

- Windows (cmd)

```
bash  
  
venv\Scripts\activate
```

Vedrai il prefisso `(venv)` nel terminale

◆ Installare pacchetti

Con ambiente attivo:

```
bash  
  
pip install requests
```

👉 I pacchetti finiscono dentro `venv/` e non nel sistema globale

◆ Disattivazione

```
bash  
  
deactivate
```

◆ Congelare dipendenze

```
bash  
  
pip freeze > requirements.txt
```

Per ricreare l'ambiente:

```
bash  
  
pip install -r requirements.txt
```

✓ Perché è importante

- Evita conflitti tra versioni di librerie
- Mantiene i progetti **puliti e riproducibili**
- Essenziale per il lavoro in team e la distribuzione

Slicing

◆ Cosa significa

Lo **slicing** permette di ottenere una sottosequenza (porzione) da:

- **stringhe**
- **liste**
- **tuple**

Sintassi:

```
python
sequenza[start:stop:step]
• start → indice di inizio (incluso)
• stop → indice di fine (escluso)
• step → passo
```

◆ Esempi con lista

```
python
numeri = [0, 1, 2, 3, 4, 5]

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

◆ Esempi con stringa

```
python
s = "Python"
print(s[0:2])      # "Py"
print(s[-3:])      # "hon"
print(s[::-1])     # "nohtyP"
```

✓ Note importanti

- `start` e `stop` sono opzionali
- Indici negativi contano da destra (`-1` = ultimo)
- Lo slicing crea **una nuova sequenza** (non modifica l'originale)

Generatori

⚡ Generatori in Python

◆ Cosa sono

- Oggetti che **producono valori uno alla volta**
- Usano `yield` al posto di `return`
- Sono **iteratori** → puoi usarli nei `for`, con `next()`

👉 Vantaggio: **non memorizzano tutta la sequenza in memoria**, ma la calcolano su richiesta (lazy evaluation).

◆ Esempio base

```
python

def contatore(n):
    i = 0
    while i < n:
        yield i
        i += 1

for x in contatore(5):
    print(x)
# 0 1 2 3 4
```

◆ Uso di `next()`

python

```
gen = contatore(3)
print(next(gen)) # 0
print(next(gen)) # 1
print(next(gen)) # 2
# StopIteration se esaurito
```

◆ Generator Expression

Come le list comprehension, ma con `()` → più leggeri

python

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

✓ Quando usarli

- Sequenze molto grandi (file, stream, dati infiniti)
- Pipeline di elaborazione dati
- Risparmio memoria e maggiore efficienza

Liste e Cicli



python

```
numeri = [1, 2, 3, 4, 5]
quadrati = []
for n in numeri:
    quadrati.append(n * n)
```



python

```
numeri = [1, 2, 3, 4, 5]
quadrati = [n*n for n in numeri]
```

Funzioni e Somma



python

```
def somma_lista(lista):
    totale = 0
    for x in lista:
        totale += x
    return totale
```



python

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

Slicing Lista



python

```
seq = [1, 2, 3, 4, 5]
invertita = []
i = len(seq) - 1
while i >= 0:
    invertita.append(seq[i])
    i -= 1
```



python

```
seq = [1, 2, 3, 4, 5]
invertita = seq[::-1]
```

Ricerca in Lista



python

```
trovato = False
for x in numeri:
    if x == 3:
        trovato = True
        break
```



python

```
trovato = 3 in numeri
```

Conteggio Frequenze



python

```
conteggio = {}
for parola in parole:
    if parola in conteggio:
        conteggio[parola] += 1
    else:
        conteggio[parola] = 1
```



python

```
from collections import Counter
conteggio = Counter(parole)
```

Generatore Quadrati



python

```
def quadrati(n):
    result = []
    for i in range(n):
        result.append(i*i)
    return result
```



python

```
def quadrati(n):
    for i in range(n):
        yield i*i
```

Costruzione Dizionario



python

```
dati = [("a",1),("b",2)]  
mappa = {}  
for k,v in dati:  
    mappa[k] = v
```



python

```
dati = [("a",1),("b",2)]  
mappa = dict(dati)
```

Filtraggio Lista



python

```
pari = []
for n in numeri:
    if n % 2 == 0:
        pari.append(n)
```



python

```
pari = [n for n in numeri if n % 2 == 0]
```

Apertura File



python

```
f = open("dati.txt", "r", encoding="utf-8")
righe = f.readlines()
f.close()
```



python

```
with open("dati.txt", "r", encoding="utf-8") as f:
    righe = f.readlines()
```

Iterare su Dizionario



python

```
d = {"a":1, "b":2}  
for k in d.keys():  
    print(k, d[k])
```



python

```
d = {"a":1, "b":2}  
for k, v in d.items():  
    print(k, v)
```

Python



Giorno 4

Marius Minia

Parte 1 – Fondamenti: OOP vs Procedurale, Classi e Oggetti (Slide 1–7)

1. Procedurale vs OOP

Approccio	Caratteristiche
Procedurale	Funzioni e dati separati, ordine di esecuzione sequenziale
OOP	Oggetti combinano dati (attributi) e comportamenti (metodi)

|  L'OOP favorisce riuso, modularità, manutenibilità

2. Cos'è una Classe?

- È un **modello** per creare oggetti (istanze).
- Definisce:
 - **Attributi:** dati che ogni oggetto possiede
 - **Metodi:** funzioni che operano sugli attributi

3. Cos'è un Oggetto?

- È un'**istanza concreta** di una classe
- Ogni oggetto ha:
 - Un suo stato (valori dei suoi attributi)
 - Un'identità unica (`id(obj)`)
 - Un comportamento (metodi della classe)

4. 📄 Definizione base

python

```
class Persona:  
    def saluta(self):  
        print("Ciao!")
```

5. ✎ Istanziazione

python

```
p = Persona()  
p.saluta() # Ciao!
```

6. 🔒 Attributi di istanza vs di classe

Tipo	Dove si definisce	Accesso	Esempio
Istanza	Dentro <code>__init__</code>	<code>self.attr</code>	<code>self.nome = "Mario"</code>
Classe	Fuori dai metodi	<code>Classe.attr</code>	<code>contatore = 0</code> (condiviso)

python

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

7.  Metodi di Istanza

- Il primo parametro è `self`, riferimento all'oggetto stesso
- Possono accedere/modificare `self.attributo`

python

```
class Conto:  
    def deposita(self, val):  
        self.saldo += val
```

8.  Metodi di Classe

- Decorati con `@classmethod`
- Primo parametro è `cls`, la classe stessa
- Usati per costruttori alternativi o logica condivisa

python

```
class Persona:  
    specie = "Homo sapiens"  
    @classmethod  
    def info_specie(cls):  
        return cls.specie
```

9.  Metodi Statici

- Decorati con `@staticmethod`
- Non accedono a `self` né a `cls`
- Sono funzioni utili legate logicamente alla classe

python

```
class Matematica:  
    @staticmethod  
    def quadrato(x):  
        return x * x
```

10. 🎬 Il Costruttore `__init__`

- Viene chiamato automaticamente alla creazione di un oggetto
- Serve per inizializzare gli attributi di istanza

python

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

11. 🌐 `__str__` vs `__repr__`

Metodo	Scopo	Output tipico
<code>__str__</code>	Umanamente leggibile	"Mario (25 anni)"
<code>__repr__</code>	Debug, rappresentazione precisa	"Persona('Mario', 25)"

python

```
def __str__(self):  
    return f"{self.nome} ({self.eta} anni)"  
def __repr__(self):  
    return f"Persona('{self.nome}', {self.eta})"
```

🛡️ Parte 3 – Incapsulamento, Attributi privati, Property (Slide 17–21)

12. 🔒 Incapsulamento

- Protezione dello stato interno di un oggetto
 - Separazione tra **interfaccia pubblica** e **implementazione privata**
 - Python usa **convenzioni**, non restrizioni reali
-

13. ↗ Attributi privati (convenzioni)

Notazione	Significato
_attributo	Uso interno (convenzione)
__attributo	Name mangling (più difficile da accedere)

```
python

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

14. 📁 Getter & Setter tradizionali

python

```
def get_nome(self):
    return self._nome
def set_nome(self, nuovo_nome):
    self._nome = nuovo_nome
```

🔴 Non molto "pythonic"

15. ✅ @property (getter elegante)

python

```
class Persona:
    @property
    def nome(self):
        return self._nome
```

16. 🖊️ @setter per scrittura controllata

python

```
@nome.setter
def nome(self, valore):
    if len(valore) >= 2:
        self._nome = valore
    else:
        print("Nome troppo corto")
```

✅ Accesso semplice ma con controllo interno:

python

```
p.nome = "Luigi" # chiama il setter
print(p.nome)     # chiama il getter
```

⌚ Metodi speciali comuni

Metodo	Scopo	Esempio
<code>__init__</code>	Costruttore	<code>x = Classe()</code>
<code>__str__</code>	Rappresentazione leggibile	<code>print(obj)</code>
<code>__repr__</code>	Rappresentazione tecnica/debug	<code>repr(obj)</code>
<code>__len__</code>	Lunghezza oggetto	<code>len(obj)</code>
<code>__eq__</code>	Uguaglianza	<code>obj1 == obj2</code>
<code>__lt__</code>	Confronto "minore di"	<code>obj1 < obj2</code>
<code>__call__</code>	Oggetto chiamabile	<code>obj()</code>
<code>__getitem__</code>	Accesso per indice o chiave	<code>obj[0]</code>
<code>__setitem__</code>	Assegnazione per indice o chiave	<code>obj[0] = val</code>
<code>__iter__</code>	Iterazione	<code>for x in obj:</code>

✍ Esempio: `__len__`, `__eq__`, `__call__`

python

```

class Parola:
    def __init__(self, testo):
        self.testo = testo

    def __len__(self):
        return len(self.testo)

    def __eq__(self, other):
        return self.testo.lower() == other.testo.lower()

    def __call__(self):
        print(self.testo.upper())

p1 = Parola("Python")
p2 = Parola("PYTHON")
print(len(p1))      # 6
print(p1 == p2)     # True
p1()                # PYTHON

```

Mixin in Python

Cosa sono

- Una **classe riutilizzabile** progettata per essere ereditata insieme ad altre
- Non rappresenta un concetto autonomo (non ha senso da sola)
- Serve per **aggiungere funzionalità** a più classi diverse senza duplicare codice

Caratteristiche

- Usano **ereditarietà multipla**
- Convenzione: il nome termina con `Mixin` (`LoggerMixin`, `PrintableMixin` ...)
- Non definiscono un'interfaccia completa, ma **comportamenti extra**
- Evitano duplicazione di codice orizzontale tra classi non collegate

Esempio pratico

python

```
class LoggerMixin:  
    def log(self, msg):  
        print(f"[LOG] {msg}")  
  
class Service(LoggerMixin):  
    def run(self):  
        self.log("Servizio in esecuzione")
```

```
s = Service()  
s.run()  
# [LOG] Servizio in esecuzione
```

Vantaggi

- Riutilizzo del codice senza forzare gerarchie innaturali
- Codice più modulare e leggibile
- Combinazione flessibile di funzionalità (puoi ereditare più mixin insieme)

Python



Giorno 7

Marius Minia

Principi SOLID in Python

Cos'è SOLID?

- Acronimo di 5 principi per scrivere codice orientato agli oggetti **pulito, estensibile e manutenibile**.
- Ogni lettera rappresenta un principio chiave:
 - **S** – Single Responsibility Principle
 - **O** – Open/Closed Principle
 - **L** – Liskov Substitution Principle
 - **I** – Interface Segregation Principle
 - **D** – Dependency Inversion Principle

■ S – Single Responsibility Principle (SRP)

Una classe deve avere **una sola ragione di cambiare**

 Ogni classe deve fare **una cosa sola**.

 **Violazione:**

```
python

class Report:
    def generate(self): ...
    def save_to_file(self): ...
```

 **Corretto (divisione):**

```
python

class ReportGenerator: ...
class FileSaver: ...
```

■ O – Open/Closed Principle (OCP)

Il codice deve essere **aperto all'estensione**, ma **chiuso alla modifica**

- ✓ Aggiungi nuovi comportamenti senza modificare il codice esistente.

Esempio con strategia:

```
python

class Shape:
    def area(self): ...

class Circle(Shape): ...
class Square(Shape): ...

def total_area(shapes: list[Shape]):
    return sum(s.area() for s in shapes)
```

■ L – Liskov Substitution Principle (LSP)

Ogni sottoclasse deve poter **sostituire** la sua superclasse **senza rompere il codice**

✗ **Violazione:**

```
python

class Bird:
    def fly(self): ...
class Ostrich(Bird): # non può volare
    def fly(self): raise NotImplementedError
```

✓ **Meglio:**

```
python

class Bird: ...
class FlyingBird(Bird): def fly(self): ...
class Ostrich(Bird): ...
```

■ I – Interface Segregation Principle (ISP)

| Non costringere le classi a implementare **metodi che non usano**

Python non ha interfacce rigide, ma vale anche per ABC e mixin.

✗ Interfaccia troppo ampia:

```
python

class Animal:
    def fly(self): ...
    def swim(self): ...
```

✓ Meglio usare mixin:

```
python

class Flying:
    def fly(self): ...
class Swimming:
    def swim(self): ...
```

■ D – Dependency Inversion Principle (DIP)

| Dipendi da **astrazioni**, non da classi concrete

✗ Codice rigido:

```
python

class MySQLDB:
    def connect(self): ...
class UserRepo:
    def __init__(self):
        self.db = MySQLDB()
```

✓ Uso di astrazioni:

```
python

class DBConnection: ...
class MySQLDB(DBConnection): ...
class UserRepo:
    def __init__(self, db: DBConnection):
        self.db = db
```

Compreseioni (List, Dict, Set) in Python

◆ Cos'è

Un modo **compatto e leggibile** per creare nuove collezioni partendo da sequenze o iterabili, evitando cicli esplicativi.

◆ Sintassi

- **Lista:** `[espressione for elemento in sequenza if condizione]`
- **Set:** `{espressione for elemento in sequenza if condizione}`
- **Dizionario:** `{chiave: valore for elemento in sequenza if condizione}`

◆ Esempi

```
python

# Lista: quadrati dei numeri pari
squares = [x**2 for x in range(10) if x % 2 == 0]
print(squares) # [0, 4, 16, 36, 64]

# Set: lettere uniche in una parola
unique = {ch for ch in "programmazione"}
print(unique) # {'p', 'r', 'o', 'g', 'a', 'm', 'z', 'i', 'n', 'e'}

# Dizionario: mappa numero→quadrato
mapping = {x: x**2 for x in range(5)}
print(mapping) # {0:0, 1:1, 2:4, 3:9, 4:16}
```

◆ Vantaggi

- Più **compatti** dei cicli tradizionali
- Spesso più **veloci**
- Facili da leggere (se usati bene)

◆ Errori comuni

- ✗ Usare compreseioni troppo complesse → codice meno leggibile
- ✗ Nidificazioni profonde → meglio usare cicli esplicativi

✓ Best practice

- ✓ Usare per trasformazioni semplici e filtri
- ✓ Limitare a **una sola condizione**
- ✓ Per logica complessa → tornare a `for` tradizionale

⚡ Lambda & Higher-Order Functions

◆ Funzioni Lambda

- Funzioni **anonime** e compatte
- Sintassi: `lambda argomenti: espressione`

```
python
```

```
quadrato = lambda x: x**2
print(quadrato(5)) # 25
```

◆ Funzioni di ordine superiore

Funzioni che prendono altre funzioni come argomento.

```
map(func, iterable)
```

Applica la funzione a ogni elemento.

```
python
```

```
nums = [1, 2, 3, 4]
doppio = list(map(lambda x: x*2, nums))
print(doppio) # [2, 4, 6, 8]
```

```
filter(func, iterable)
```

Seleziona gli elementi che rispettano la condizione.

```
python
```

```
pari = list(filter(lambda x: x % 2 == 0, nums))
print(pari) # [2, 4]
```

```
reduce(func, iterable) (serve import)
```

Combinà gli elementi riducendoli a un singolo valore.

```
python
```

```
from functools import reduce
somma = reduce(lambda a, b: a + b, nums)
print(somma) # 10
```

◆ Vantaggi

- ✓ Codice conciso
- ✓ Utile con trasformazioni semplici
- ✓ Ottimo con funzioni matematiche o di aggregazione

◆ Limiti

- ✗ Leggibilità bassa con logica complessa
- ✗ Spesso meglio usare **list comprehension**

Itertools & Built-in Avanzati

◆ Built-in utili

- `enumerate(iterable)` → coppie (indice, valore)

```
python
```

```
animali = ["cane", "gatto", "pesce"]
for i, a in enumerate(animali):
    print(i, a) # 0 cane ...
```

- `zip(a, b, ...)` → combina più sequenze

```
python
```

```
nomi = ["Anna", "Marco"]
eta = [25, 30]
print(list(zip(nomi, eta))) # [('Anna', 25), ('Marco', 30)]
```

- `any(iterable)` → True se almeno un elemento è vero
- `all(iterable)` → True se tutti sono veri

```
python
```

```
print(any([0, 1, 0])) # True
print(all([1, 2, 3])) # True
```

◆ Modulo `itertools` (standard library)

Offre strumenti potenti per iterazioni avanzate.

`count(start=0, step=1)`

Iteratore infinito di numeri.

```
python
```

```
from itertools import count
for i in count(10, 2):
    if i > 20: break
    print(i) # 10, 12, 14, 16, 18, 20
```

`cycle(iterable)`

Ripete all'infinito una sequenza.

`product(a, b)`

Prodotto cartesiano.

```
python
```

```
from itertools import product
print(list(product([1,2], ['a','b'])))
# [(1,'a'), (1,'b'), (2,'a'), (2,'b')]
```

`permutations(iterable, r)`

Tutte le permutazioni possibili.

```
python
```

```
from itertools import permutations
print(list(permutations([1,2,3], 2)))
# [(1,2), (1,3), (2,1), (2,3), (3,1), (3,2)]
```

✓ Quando usarli

✓ `zip`, `enumerate` → codice più leggibile

✓ `any`, `all` → validazioni rapide

✓ `itertools` → problemi combinatori, generazione sequenze infinite, simulazioni

Type Hints in Python

◆ Cosa sono

Annotazioni opzionali che descrivono i **tipi di variabili, parametri e ritorni**.

- Non cambiano l'esecuzione
- Aiutano leggibilità, debugging e strumenti come **mypy** o IDE

◆ Sintassi base

```
python

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

x: str = "ciao"
y: list[int] = [1, 2, 3]
```

◆ Collezioni tipizzate (Python 3.9+)

- `list[int]`
- `dict[str, float]`
- `set[str]`
- `tuple[int, int, int]`

```
python

rubrica: dict[str, str] = {"Anna": "123", "Marco": "456"}
```

◆ Tipi speciali (`typing`)

```
python

from typing import Optional, Union, Callable

def greet(name: Optional[str]) -> str:
    return f"Ciao {name or 'ospite'}"

def parse(x: Union[int, float]) -> float:
    return float(x)

Funzione = Callable[[int, int], int]
```

◆ Vantaggi

- ✓ Codice più **chiaro e mantenibile**
- ✓ Supporto dagli editor (autocompletamento, linting)
- ✓ Documentazione implicita

◆ Limiti

- ✗ Non obbligatori → Python resta dinamico
- ✗ Serve disciplina per mantenerli aggiornati

Python



Giorno 8

Marius Minia

Metodo `group_by`

python

```
def group_by(self, key_or_fn: Union[str, Callable[[Dict[str, Any]], Any]])>
    groups: Dict[Any, List[Dict[str, Any]]] = {}
    for r in self._rows:
        if isinstance(key_or_fn, str):
            if key_or_fn not in r:
                raise ValueError(f"Campo {key_or_fn} mancante")
            k = r[key_or_fn]
        else:
            k = key_or_fn(r)
        groups.setdefault(k, []).append(r)
    return {k: DataSet(v) for k, v in groups.items()}
```

Funzione

- Permette di **raggruppare i record** in base a:
 - un campo (stringa) → esempio: "city",
 - oppure una funzione che calcola la chiave → esempio: `lambda r: r["age"] // 10` (fascia d'età).

Meccanismo

1. Si crea un dizionario vuoto `groups`.
 2. Si itera su tutti i record:
 - Se il parametro è una stringa:
 - si controlla che il campo esista, altrimenti si solleva `ValueError`;
 - si estrae il valore di quella chiave.
 - Se il parametro è una funzione:
 - si calcola la chiave chiamando la funzione sul record.
 3. Ogni record viene inserito nel gruppo corrispondente (`groups.setdefault(...).append(r)`).
 4. Alla fine, per ogni gruppo si costruisce un **nuovo** `DataSet`.
- Risultato: un dizionario che mappa ogni chiave → sotto-dataset.

Metodo `_numeric_series`

python

```
def _numeric_series(self, field: str) -> List[float]:
    values = []
    for r in self._rows:
        if field not in r:
            raise ValueError(f"Campo {field} mancante in un record")
        v = r[field]
        if not isinstance(v, (int, float)):
            raise TypeError(f"Valore non numerico trovato: {v}")
        values.append(float(v))
    return values
```

Funzione

- Estraie i valori numerici di un campo specifico (es. `"salary"`) da tutti i record.
- Serve come **metodo di supporto** per le aggregazioni (`sum`, `mean`, `min`, `max`).

Meccanismo

- Controlla che ogni record contenga il campo.
- Controlla che i valori siano numerici (`int` o `float`).
- Solleva errori (`ValueError` o `TypeError`) in caso di dati invalidi.
- Restituisce una lista di `float`.

Metodi di aggregazione

sum

python

```
def sum(self, field: str) -> float:  
    vals = self._numeric_series(field)  
    return sum(vals)
```

- Somma tutti i valori numerici di un campo.

min

python

```
def min(self, field: str) -> float:  
    vals = self._numeric_series(field)  
    return min(vals)
```

- Restituisce il valore minimo del campo.

mean

python

```
def mean(self, field: str) -> float:  
    vals = self._numeric_series(field)  
    if not vals:  
        raise ValueError("Dataset vuoto")  
    return sum(vals) / len(vals)
```

- Calcola la media dei valori numerici.
- Gestisce il caso del dataset vuoto con `ValueError`.

max

python

```
def max(self, field: str) -> float:  
    vals = self._numeric_series(field)  
    return max(vals)
```

- Restituisce il valore massimo del campo.