Corso Pratico Python

[Modulo 1- 2026]

2.2. Tipizzazione

2.2.1. Strutture dati del linguaggio

2.2.2. Array

2.2.3. Oggetti di sistema e convenzioni delle librerie

Richiedente: Dott. Roberto Felici [CNR-ISM] • Insegnante: Dott. Scigè J. Liu [INAF-IAPS]

2.2. Tipizzazione

2.2. Tipizzazione in Python

Python è un linguaggio a **tipizzazione dinamica** e **forte**.

Dinamica: Non è necessario dichiarare il tipo di una variabile; il tipo è legato al valore, non al nome della variabile.

Forte: Il linguaggio non effettua conversioni implicite tra tipi incompatibili (es. non puoi sommare una stringa e un intero senza un cast esplicito).

Nota Bene: Sebbene dinamico, Python supporta i **Type Hints** (es. x: int = 5) per migliorare la leggibilità e facilitare il debugging tramite tool esterni.

2.2.1. Strutture dati del linguaggio

Python offre quattro strutture dati "built-in" principali, ognuna con caratteristiche specifiche di mutabilità e ordinamento:

Struttura	Sintassi	Caratteristiche
Lista	[a, b, c]	Ordinata, **mutabile**, permette duplicati.
Tupla	(a, b, c)	Ordinata, **immutabile**, più veloce delle liste.
Set	{a, b, c}	Non ordinata, **mutabile**, valori univoci.
Dizionario	{"k": "v"}	Coppie Chiave-Valore, mutabile, accesso rapido.

```python
numeri = [5, 2, 9, 1]
numeri.sort() # Ora numeri è [1, 2, 5, 9]

# Ordinamento personalizzato (per lunghezza stringa)
nomi = ["Marco", "Anna", "Alessandro"]
nomi.sort(key=len) # ["Anna", "Marco", "Alessandro"]
```

```python
tupla_orig = (10, 1, 5)
nuova_lista = sorted(tupla_orig)
# Risultato: [1, 5, 10] (diventa una lista)

tupla_ordinata = tuple(sorted(tupla_orig))
# Risultato: (1, 5, 10)
```

2.2.1. Strutture dati del linguaggio

Python offre quattro strutture dati "built-in" principali, ognuna con caratteristiche specifiche di mutabilità e ordinamento:

Struttura	Sintassi	Caratteristiche
Lista	[a, b, c]	Ordinata, **mutabile**, permette duplicati.
Tupla	(a, b, c)	Ordinata, **immutabile**, più veloce delle liste.
Set	{a, b, c}	Non ordinata, **mutabile**, valori univoci.
Dizionario	{"k": "v"}	Coppie Chiave-Valore, mutabile, accesso rapido.

```python
mio_set = {3, 1, 4, 2}
lista_da_set = sorted(mio_set) # [1, 2, 3, 4]
```

```python
diz = {"b": 2, "a": 5, "c": 1}
ordinato_chiavi = dict(sorted(diz.items()))
# {'a': 5, 'b': 2, 'c': 1}
```

```python
ordinato_valori = dict(sorted(diz.items(), key=lambda x: x[1]))
# {'c': 1, 'b': 2, 'a': 5}
```

Struttura	Metodo in-place	Ritorna nuova lista	Note
Lista	✅ `sort()`	✅ `sorted()`	La più flessibile.
Tupla	❌	✅ `sorted()`	Il risultato è sempre una lista.
Set	❌	✅ `sorted()`	Perde la natura di "set" se ordinato.
Dizionario	❌	✅ `sorted()`	Si ordinano le tuple `(key, value)`.

2.2.2. Array

In Python standard, le "liste" fungono spesso da array, ma per calcoli scientifici o gestione efficiente della memoria esistono alternative:

1. Modulo array (Standard Library)

Contiene elementi dello **stesso tipo** (tipizzazione rigida).

Più efficiente delle liste per grandi volumi di dati numerici semplici.

2. NumPy Array (Standard de facto)

Performance: Scritto in C, permette operazioni vettoriali.

Multidimensionalità: Supporta matrici e tensori complessi.

```python
import array

# Creazione: il carattere 'i' indica il tipo "signed integer" (intero
numeri = array.array('i', [10, 20, 30, 40])

# 1. Aggiunta di un elemento
numeri.append(50)

# 2. Accesso tramite indice (come una lista)
primo = numeri[0]  # 10

# 3. Tentativo di errore (Tipizzazione Forte)
# numeri.append("ciao")  # Errore! Si possono aggiungere solo interi.

print(f"Array standard: {numeri}")
```

2.2.2. Array

In Python standard, le "liste" fungono spesso da array, ma per calcoli scientifici o gestione efficiente della memoria esistono alternative:

1. Modulo array (Standard Library)

Contiene elementi dello **stesso tipo** (tipizzazione rigida).

Più efficiente delle liste per grandi volumi di dati numerici semplici.

2. NumPy Array (Standard de facto)

Performance: Scritto in C, permette operazioni vettoriali.

Multidimensionalità: Supporta matrici e tensori complessi.

```python
import array

# Creazione: il carattere 'i' indica il tipo "signed integer" (intero
numeri = array.array('i', [10, 20, 30, 40])

# 1. Aggiunta di un elemento
numeri.append(5

# 2. Accesso tr
primo = numeri[

# 3. Tentativo
# numeri.append

print(f"Array s
```

```python
import numpy as np

# Creazione di un array NumPy
vettore = np.array([1, 2, 3, 4])

# 1. Operazione vettoriale (Broadcasting)
# In una lista normale dovresti fare un ciclo for.
raddoppiato = vettore * 2  # [2, 4, 6, 8]

# 2. Array multidimensionale (Matrice)
matrice = np.array([[1, 2], [3, 4]])

# 3. Funzioni statistiche integrate
media = np.mean(vettore)  # 2.5

print(f"NumPy Array:\n{vettore}")
print(f"Matrice NumPy:\n{matrice}")
```

2.2.2.1 Persistenza di Strutture Dati Complesse [Digressione sul tema : condivisione / serializzazione]

In Python manipoliamo oggetti nidificati.
Salvare e ricaricare queste strutture
richiede la **serializzazione**
(conversione in un formato memorizzabile)

```python
import json
import bson  # Richiede: pip install pymongo

# Struttura dati complessa: Info su un corso
dati_corso = {
    "titolo": "Sviluppo Python Advanced",
    "codice": 102,
    "completo": False,
    "studenti": [
        {"id": 1, "nome": "Alice", "voti": [28, 30, 27]},
        {"id": 2, "nome": "Bob",   "voti": [24, 25]}
    ],
    "config": ("standard", 2024) # Una tupla (immutabile)
}
```

2.2.2.1 Persistenza di Strutture Dati Complesse [Digressione sul tema : condivisione / serializzazione]

In Python manipoliamo oggetti nidificati.
Salvare e ricaricare queste strutture richiede la **serializzazione**
(conversione in un formato memorizzabile)

```python
import json
import bson  # Richiede: pip install pymongo

# Struttura dati complessa: Info su un corso
dati_corso = {
    "titolo": "Sviluppo Python Advanced",
    "codice": 102,
    "completo": False,
    "studenti": [
        {"id": 1, "nome": "Alice", "voti": [28, 30, 27]},
        {"id": 2, "nome": "Bob",   "voti": [24, 25]}
    ],
    "config": ("standard", 2024) # Una tupla (immutabile)
}
```

```python
# --- SALVATAGGIO ---
# Convertiamo l'intero oggetto in stringa
with open("corso.txt", "w") as f:
    f.write(str(dati_corso))

# --- RICARICAMENTO ---
with open("corso.txt", "r") as f:
    contenuto = f.read()
    # ATTENZIONE: eval è pericoloso se non controlli la sorgente
    dati_testo = eval(contenuto)

print(dati_testo['studenti'][0]['nome']) # Alice
```

2.2.2.1 Persistenza di Strutture Dati Complesse [Digressione sul tema : condivisione / serializzazione]

In Python manipoliamo oggetti nidificati.
Salvare e ricaricare queste strutture richiede la **serializzazione**
(conversione in un formato memorizzabile)

```python
import json
import bson  # Richiede: pip install pymongo

# Struttura dati complessa: Info su un corso
dati_corso = {
    "titolo": "Sviluppo Python Advanced",
    "codice": 102,
    "completo": False,
    "studenti": [
        {"id": 1, "nome": "Alice", "voti": [28, 30, 27]},
        {"id": 2, "nome": "Bob",   "voti": [24, 25]}
    ],
    "config": ("standard", 2024) # Una tupla (immutabile)
}
```

```python
# --- SALVATAGGIO COMPATTO (Una sola riga) ---
# separators=(',', ':') rimuove gli spazi inutili
with open("corso_compatto.json", "w") as f:
    json.dump(dati_corso, f, separators=(',', ':'))

# --- SALVATAGGIO INDENTATO (Leggibile) ---
# indent=2 usa due spazi per ogni livello
with open("corso_human.json", "w") as f:
    json.dump(dati_corso, f, indent=2)
```

2.2.2.1 Persistenza di Strutture Dati Complesse [Digressione sul tema : condivisione / serializzazione]

In Python manipoliamo oggetti nidificati.
Salvare e ricaricare queste strutture richiede la **serializzazione** (conversione in un formato memorizzabile)

```python
import json
import bson  # Richiede: pip install pymongo

# Struttura dati complessa: Info su un corso
dati_corso = {
    "titolo": "Sviluppo Python Advanced",
    "codice": 102,
    "completo": False,
    "studenti": [
        {"id": 1, "nome": "Alice", "voti": [28, 30, 27]},
        {"id": 2, "nome": "Bob",   "voti": [24, 25]}
    ],
    "config": ("standard", 2024) # Una tupla (immutabile)
}
```

```python
# --- SALVATAGGIO COMPATTO (Una sola riga) ---
# separators=(',', ':') rimuove gli spazi inutili
with open("corso_compatto.json", "w") as f:
    json.dump(dati_corso, f, separators=(',', ':'))

# --- SALVATAGGIO INDENTATO (Leggibile) ---
```

```python
# --- SALVATAGGIO BINARIO ---
# serializza i dati in formato bson (binario)
with open("corso.bson", "wb") as f: # Notare 'wb' per write-binary
    f.write(bson.dumps(dati_corso))
```

2.2.2.1 Persistenza di Strutture Dati Complesse [Digressione sul tema : condivisione / serializzazione]

In Python manipoliamo oggetti nidificati.
Salvare e ricaricare queste strutture richiede la **serializzazione**
(conversione in un formato memorizzabile)

```python
import json
import bson  # Richiede: pip install pymongo

# Struttura dati complessa: Info su un corso
dati_corso = {
    "titolo": "Sviluppo Python Advanced",
    "codice": 102,
    "completo": False,
    "studenti": [
        {"id": 1, "nome": "Alice", "voti": [28, 30, 27]},
        {"id": 2, "nome": "Bob",   "voti": [24, 25]}
    ],
    "config": ("standard", 2024) # Una tupla (immutabile)
}
```

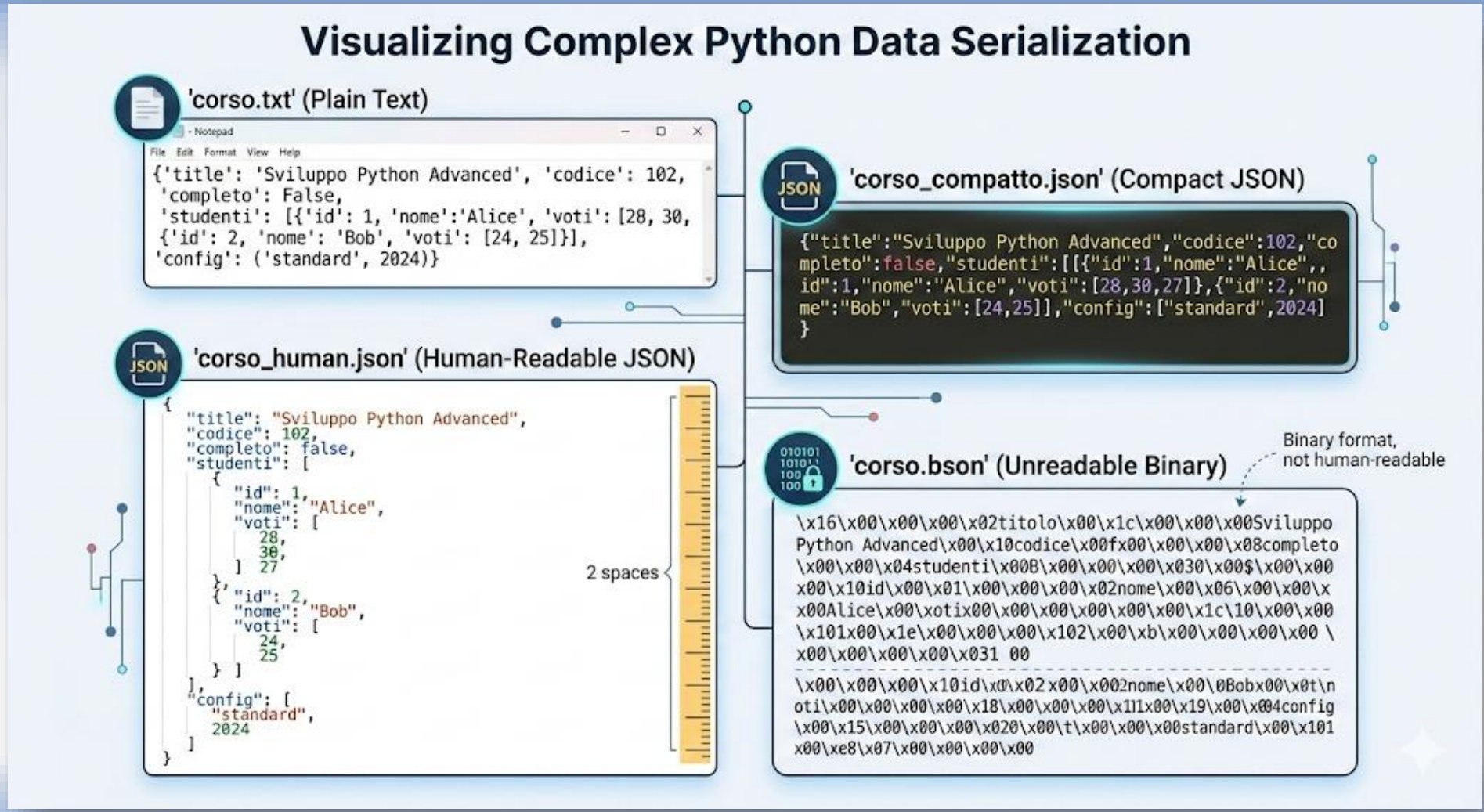
```python
# --- SALVATAGGIO COMPATTO (Una sola riga) ---
# separators=(',', ':') rimuove gli spazi inutili
with open("corso_compatto.json", "w") as f:
    json.dump(dati_corso, f, separators=(',', ':'))

# --- SALVATAGGIO INDENTATO (Leggibile) ---
```

```python
# --- SALVATAGGIO BINARIO ---
# serializza i dati in formato bson (binario)
with open("corso.bson", "wb") as f: # Notare 'wb' per write-binary
    f.write(bson.dumps(dati_corso))
```

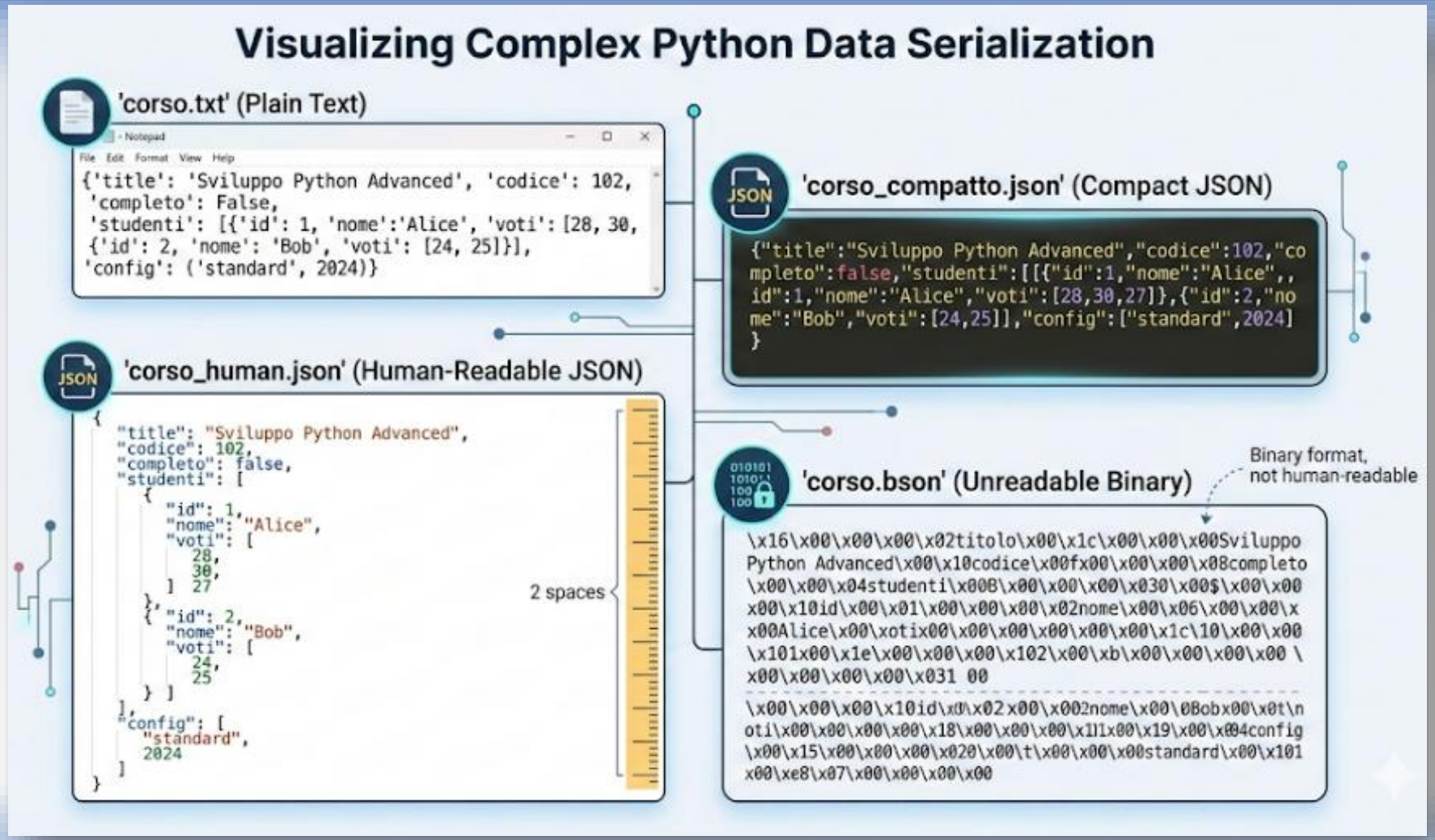
? ALTRI FORMATI ?

2.2.2.1 Persistenza di Strutture Dati Complesse RECAP 1



2.2.2.1 Persistenza di Strutture Dati Complesse
RECAP 2

1.`corso.txt`: semplice testo normale, struttura dizionario Python, leggibile, non per l'interoperabilità.
2.`corso_compatto.json`: visualizzazione JSON compatta, tutti gli spazi e i ritorni sono rimossi, risparmiare spazio. Una riga di testo [buono in rete].
3.`corso_human.json`: stessa JSON, formattata ("beautified"). facilmente leggibile dagli esseri umani.
4.`corso.bson`: anteprima binaria. BSON è un binario serializzato, mostrato come dati esadecimali illeggibili a occhio nudo, ma ottimizzato per l'archiviazione e la velocità [tipo MongoDB]



2.2.3. Oggetti di Sistema: sys vs os

Il modulo sys (System-specific parameters)

Il modulo sys fornisce l'accesso a variabili e funzioni che interagiscono strettamente con l'**interprete Python**.

- **sys.argv**: Una lista che contiene gli argomenti passati allo script da riga di comando.
- **sys.path**: Una lista di stringhe che specifica i percorsi di ricerca per i moduli (fondamentale per risolvere errori di import).
- **sys.exit()**: Permette di uscire dal programma con un codice di stato specifico.
- **sys.platform**: Identifica il sistema operativo su cui gira l'interprete (es. 'linux', 'win32', 'darwin').

```python
import sys

print(f"Nome dello script: {sys.argv[0]}")
if len(sys.argv) > 1:
    print(f"Primo argomento ricevuto: {sys.argv[1]}")
```

2.2.3. Oggetti di Sistema: sys vs os

Il modulo os (Operating System interface)

Il modulo os permette a Python di dialogare con il **sistema operativo** ospite, indipendentemente dal fatto che sia Windows, macOS o Linux.

- **os.environ**: Un dizionario che contiene le variabili d'ambiente del sistema.
- **os.getcwd()**: Restituisce la "Current Working Directory" (la cartella in cui ti trovi).
- **os.listdir()**: Elenca i file e le cartelle in un percorso.
- **os.mkdir() / os.remove()**: Per creare o eliminare file e directory.

```python
import sys

print(f"Nome dello script: {sys.argv[0]}")
if len(sys.argv) > 1:
    print(f"Primo argomento ricevuto: {sys.argv[1]}")
```

```python
import os

# Ottenere il valore di una variabile d'ambiente (es. USER o PATH)
utente = os.environ.get('USER') or os.environ.get('USERNAME')
print(f"L'utente corrente è: {utente}")

# Verificare se un file esiste prima di aprirlo
if os.path.exists("dati.json"):
    print("File trovato!")
```

2.2.3. Oggetti di sistema e convenzioni delle librerie

Convenzioni delle Librerie e "Dunder"

La gestione degli oggetti di sistema segue convenzioni rigide per evitare conflitti di nomi:

Attributi Public: oggetto.nome (accessibile a tutti).

Attributi Protected: ***_nome***
(suggerisce un uso interno alla classe/modulo).

Attributi Private: ***__nome***
(attiva il *name mangling,*
evitare collisioni nelle sottoclassi).

Dunder Methods (Double Under):
__init__, __str__, __main__.
Sono metodi "magici" invocati dall'interprete.

Esempio di Sistema:

`if __name__ == "__main__"`: assicura che il codic
venga eseguito solo se lo script è lanciato
direttamente, e non se viene importato come modulo
in un altro file.

```python
import sys

print(f"Nome dello script: {sys.argv[0]}")
if len(sys.argv) > 1:
    print(f"Primo argomento ricevuto: {sys.argv[1]}")
```

```python
import os

# Ottenere il valore di una variabile d'ambiente (es. USER o PATH)
utente = os.environ.get('USER') or os.environ.get('USERNAME')
print(f"L'utente corrente è: {utente}")

# Verificare se un file esiste prima di aprirlo
if os.path.exists("dati.json"):
    print("File trovato!")
```

2.2.3. Oggetti di sistema e convenzioni delle librerie

Librerie della Standard Library (Batteries Included)

Questi moduli sono già installati con Python,
sono essenziali per la gestione dei dati e del sistema.

math: Funzioni matematiche avanzate: trigonometria, logaritmi, costanti come **π** ed **e**.

- *Esempio:* math.sqrt(25), math.ceil(4.2).

datetime: Gestione di date, orari e calcolo degli intervalli temporali (timedelta, datetime).

- *Esempio:* datetime.now(), calcolo della differenza tra due date.

importlib: Permette di importare moduli dinamicamente tramite stringhe.
Utile per plugin o sistemi modulari.

- *Esempio:* importlib.import_module("nome_modulo").

collections: Strutture dati specializzate che potenziano quelle di base.

- *Esempio:* namedtuple (tuple con nomi di campo), Counter (per contare occorrenze), deque.

re (Regular Expressions): Per la ricerca e manipolazione avanzata di stringhe tramite pattern.

Per i Dati e la Scienza

- **numpy**: (Già citata) Calcolo numerico e matriciale ultra-veloce.
- **pandas**: La libreria definitiva per la manipolazione di tabelle e dataset (DataFrame). Fondamentale per la Data Science.
- **matplotlib / seaborn**: Creazione di grafici statici, animati e interattivi.
- **Per il Web e l'Automazione**
- **requests**: La libreria standard per effettuare chiamate HTTP. Molto più intuitiva di urllib.
- **pydantic**: Validazione dei dati e gestione dei tipi (molto usata in FastAPI).
- **python-dotenv**: Per caricare variabili d'ambiente da un file .env (best practice di sicurezza).

2.2.3. Oggetti di sistema e convenzioni delle librerie

Librerie della Standard Library (Batteries Included)

Questi moduli sono già installati con Python,
sono essenziali per la gestione dei dati e del sistema.

math: Funzioni matematiche avanzate: trigonometria, logaritmi, costanti come **π** ed **e**.

- *Esempio:* math.sqrt(25), math.ceil(4.2).

datetime: Gestione di date, orari e calcolo degli intervalli

- *Esempio:* namedtuple (tuple con nomi di campo), Counter (per contare occorrenze), deque.

re (Regular Expressions): Per la ricerca e manipolazione avanzata di stringhe tramite pattern.

Per i Dati e la Scienza

- **numpy**: (Già citata) Calcolo numerico e matriciale ultra-veloce.
- **pandas**: La libreria definitiva per la manipolazione di tabelle e dataset (DataFrame). Fondamentale per la Data Science.
- **matplotlib / seaborn**: Creazione di grafici statici, animati e interattivi.

Libreria	Scopo Principale	Alternativa / Correlata
`json`	Serializzazione dati testuali	`pickle` (binario Python), `bson`
`math`	Operazioni scalari semplici	`numpy` (per operazioni su array/matrici)
`sys`	Stato dell'interprete	`os` (stato del sistema operativo)
`datetime`	Gestione tempo	`time` (per timestamp a basso livello)