Corso Pratico Python

4. Lettura e Visualizzazione di valori
4.1.1. Print
4.1.2. Formatting
4.1.3. Date Time

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

4. Lettura e Visualizzazione di valori

Titolo: Il concetto di Input/Output

Definizione:
L'I/O (Input/Output) è il processo con cui un programma scambia dati con il mondo esterno. Non limitiamoci a pensare all'I/O come stringhe leggibili. I dati possono essere:

1. **Testuali (Stringhe):**
   Sequenze di caratteri (Unicode in Python 3).
2. **Binari (Bytes):**
   Flussi di dati grezzi, rappresentati dal prefisso `b''` (es. `b'\x01\x02'`).
3. **Rappresentazione Esadecimale:**
   Spesso i dati binari vengono visualizzati in **Hex** per facilitarne la lettura durante il debugging di protocolli o file raw.

Sorgenti e Destinazioni

- **Titolo: Flussi e Dispositivi**
- **Standard Streams:**
  - stdin: Input standard (solitamente la tastiera).
  - stdout: Output standard (il terminale).
  - stderr: Per i messaggi di errore.
- **Oltre la Console:** L'I/O in Python può interfacciarsi con:
  - **File System:** Lettura/scrittura di file su disco.
  - **Network:** Socket TCP/UDP per comunicazioni web.
  - **Hardware/Sensori:** Porte seriali (USB), GPIO (Raspberry Pi), o dispositivi esterni.

Visualizzazione Standard (4.1.1)

La funzione print()

Sintassi base:

```
print(*objects, sep=' ',
      end='\n',file=sys.stdout)
```

- **Caratteristiche principali:**
  - **sep**: Definisce cosa inserire tra gli oggetti (default: spazio).
  - **end**: Definisce come terminare la riga (default: a capo).
- **Reindirizzamento:** È possibile usare il parametro file per scrivere direttamente su un file anziché sulla console.

```python
import sys

# 1. Definiamo i dati da stampare
timestamp = "2026-03-16 14:00"
livello = "INFO"
messaggio = "Connessione al database stabilita"

# 2. Scrittura su FILE
# Usiamo 'with' per assicurarci che il file venga chiuso correttamente
with open("output.txt", "w", encoding="utf-8") as f:
    print(
        timestamp,
        livello,
        messaggio,
        sep=" | ",      # Cambia lo spazio di default con un pipe
        end=" ✔\n",     # Aggiunge un'icona e va a capo
        file=f          # Invia l'output al file 'f' invece che a sys.stdout
    )

print("Operazione completata. Controlla il file 'output.txt'!")
```

Formattazione dei Dati (4.1.2)

String Formatting

Esistono tre ere della formattazione in Python:

- **% Operator (Legacy):** Ereditato dal C (es. %d, %s). Ormai sconsigliato.
- **str.format():** Metodo versatile che usa le parentesi graffe {}.
- **F-Strings (Moderno):** Introdotte in Python 3.6, sono le più performanti e leggibili. Permettono di inserire espressioni direttamente nelle stringhe: f"{variabile}".
  - **Mini-language:** Supportano arrotondamenti (:.2f), allineamenti e padding.

```python
# Esempio Binario ed Hex
dati_raw = b'\xff\x00\x41' # Hex per 255, 0 e 'A'
print(f"Dati binari: {dati_raw}")

# Formattazione avanzata con F-String
prezzo = 49.99
sconto = 0.15
print(f"Prezzo scontato: {prezzo * (1 - sconto):.2f}€")

# Uso di sep ed end
print("A", "B", "C", sep=" -> ", end=" [FINE]")
# Output: A -> B -> C [FINE]
```

Gestione del Tempo (4.1.3)

Date e Time

Modulo datetime:
Lo standard per manipolare date e orari.

1. **datetime.now():**
   Cattura l'istante attuale.
2. **strftime (Format):**
   Converte un oggetto data in una stringa leggibile.
3. **strptime (Parse):**
   Converte una stringa in un oggetto data.

Modulo Timedelta:
Oggetto usato per rappresentare la durata o la differenza tra due date (es. "tra 5 giorni").

```python
from datetime import datetime, timedelta

# 1. Ottenere e formattare la data attuale
ora_esatta = datetime.now()
print(f"Oggi è il: {ora_esatta.strftime('%d/%m/%Y %H:%M')}")

# 2. Operazioni temporali
tra_una_settimana = ora_esatta + timedelta(days=7)
print(f"Scadenza: {tra_una_settimana.strftime('%A, %d %B')}")

# 3. Parsing da stringa
data_str = "2026-12-25"
natale = datetime.strptime(data_str, "%Y-%m-%d")
```

Extra stdin/stdout/stderr

```python
import sys

def elabora_dati():
    # 1. Lettura da STDIN (Standard Input)
    # sys.stdin.readline() legge l'intera riga inclusa la sosta \n
    print("Inserisci il tuo codice identificativo (solo numeri):")
    input_utente = sys.stdin.readline().strip()

    # 2. Logica di controllo
    if not input_utente.isdigit():
        # 3. Scrittura su STDERR (Standard Error)
        # Usiamo file=sys.stderr per separare l'errore dai dati corretti
        print(f"ERRORE: '{input_utente}' non è un numero valido!", file=sys.stderr)
        sys.exit(1) # Esce con un codice di errore
    else:
        # 4. Scrittura su STDOUT (Standard Output)
        # Questo è l'output "pulito" del programma
        risultato = int(input_utente) * 2
        print(f"Codice elaborato correttamente: {risultato}")

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

Extra stdin/stdout/stderr

```python
import sys

def elabora_dati():
    # 1. Lettura da STDIN (Standard Input)
    # sys.stdin.readline() legge l'intera riga inclusa la sosta \n
    print("Inserisci il tuo codice identificativo (solo numeri):")
    input_utente = sys.stdin.readline().strip()

    # 2. Logica di controllo
    if not input_utente.isdigit():
        # 3. Scrittura su STDERR (Standard Error)
        # Usiamo file=sys.stderr per separare l'errore dai dati corretti
        print(f"ERRORE: '{input_utente}' non è un numero valido!", file=sys.stderr)
        sys.exit(1) # Esce con un codice di errore
    else:
        # 4. Scrittura su STDOUT (Standard Output)
        # Ques
        risult
        print

if __name__ ==
    elabora_da
```

```python
import sys

print("ERRORE CRITICO: Disco pieno!", file=sys.stderr)
```

Extra logging [a.k.a.: log4python]

Oltre il semplice `print()` a una gestione professionale dei log in Python, il modulo standard da utilizzare è **`logging`**. È il sistema di "log4python" (ispirato a log4j) che permette di differenziare i messaggi per importanza e destinazione.
Il vantaggio principale è che puoi decidere cosa loggare e dove mandarlo (console, file, email, server remoto) senza modificare il codice logica, ma solo la configurazione.
Severity: Python definisce 5 livelli standard. Di default, il sistema mostra solo dal livello `WARNING` in su.

Livello	Valore	Scopo
DEBUG	10	Dettagli diagnostici (es. variabili, flussi).
INFO	20	Conferma che le cose funzionano come previsto.
WARNING	30	Qualcosa di inaspettato (es. spazio disco quasi pieno).
ERROR	40	Un problema serio, il software non può eseguire una funzione.
CRITICAL	50	Errore gravissimo, il programma potrebbe interrompersi.

```python
import logging
import sys

# 1. Creazione del Logger
logger = logging.getLogger("AppMonitor")
logger.setLevel(logging.DEBUG) # Cattura tutto, dal debug in su

# 2. Handler per la Console (scrive su stderr)
c_handler = logging.StreamHandler(sys.stderr)
c_handler.setLevel(logging.WARNING) # In console solo avvisi o errori

# 3. Handler per il File
f_handler = logging.FileHandler("app_log.txt", mode="a", encoding="utf-8")
f_handler.setLevel(logging.DEBUG) # Nel file scriviamo tutto il dettaglio

# 4. Creazione dei Formati (Layout)
# %(asctime)s -> timestamp, %(name)s -> nome logger, %(levelname)s -> live
c_format = logging.Formatter('%(name)s - %(levelname)s - %(message)s')
f_format = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(m

c_handler.setFormatter(c_format)
f_handler.setFormatter(f_format)

# 5. Aggiunta degli handler al logger
logger.addHandler(c_handler)
logger.addHandler(f_handler)

# --- UTILIZZO ---
logger.debug("Questa è una nota tecnica invisibile in console")
```

```python
try:
    risultato = 10 / 0
except ZeroDivisionError:
    logger.error("Tentativo di divisione per zero!", exc_info=True)
```