ChatGPT

Cassetta degli Attrezzi Python per Programmatori

In questa cassetta degli attrezzi troverai una raccolta di funzioni base del linguaggio Python e 100 snippet di codice utili per risolvere problemi comuni nello sviluppo. Il documento è suddiviso in due parti: (1) le funzioni essenziali per i principali tipi di dato Python (con esempi pratici), e (2) una collezione di 100 "attrezzi" ovvero script e utility Python per automatizzare compiti frequenti, facilitare il debugging, analizzare codice staticamente, gestire formati di file, ottimizzare performance, migliorare logging, e molto altro. Tutto il contenuto è presentato in italiano, con codice ben formattato e commentato per facilitarne la comprensione.

Indice

1. Funzioni Base per i Tipi Principali di Python

Funzioni Built-in Generali

Python mette a disposizione funzioni built-in utilizzabili su (quasi) tutti gli oggetti. Ecco alcune delle più comuni con esempi:

- `type(obj)` : restituisce il tipo dell'oggetto `obj` .

```
x = 42
print(type(x))    # <class 'int'>
y = "hello"
print(type(y))    # <class 'str'>
```

- `isinstance(obj, Classe)`: verifica se `obj` è un'istanza della classe (o tipo) specificata.

```
num = 3.14
print(isinstance(num, float))   # True, num è un float
print(isinstance(num, int))     # False, num non è un int
```

- `id(obj)`: restituisce un identificativo unico dell'oggetto (il riferimento in memoria).

```
a = 10
b = a
print(id(a), id(b))  # Stampa due valori uguali, a e b referenziano lo
stesso oggetto int 10
```

- `str(obj)`: converte l'oggetto in rappresentazione testuale (stringa). (Da non confondere con il metodo `.__str__()`.)

```
n = 255
print(str(n))       # "255" (numero convertito in stringa)
print(str(3.5))     # "3.5"
print(str([1, 2]))  # "[1, 2]" (lista convertita in stringa)
```

- `dir(obj)`: senza argomenti, elenca nomi definiti nel contesto corrente; con un oggetto, ne elenca attributi e metodi utili per introspezione.

```
import math
print(dir(math))   # Elenca tutti gli attributi e funzioni del modulo
math
print(dir("abc"))  # Elenca i metodi disponibili per le stringhe
```

Stringhe (`str`)

Le stringhe in Python sono sequenze immutabili di caratteri. Ecco alcune funzioni e metodi base per le stringhe:

- `len(s)`: restituisce la lunghezza (numero di caratteri) della stringa `s`.

```
testo = "Python"
print(len(testo))  # 6
```

- `s.lower()` / `s.upper()`: restituisce una nuova stringa con tutti i caratteri di `s` in minuscolo o maiuscolo.

```
nome = "Mario Rossi"
print(nome.lower())  # "mario rossi"
print(nome.upper())  # "MARIO ROSSI"
```

- `s.strip()`: rimuove gli spazi bianchi iniziali e finali dalla stringa `s` (utile per pulire input).

```
testo = "  ciao mondo  "
print(testo.strip())  # "ciao mondo"
```

- `s.split(delimitatore)`: suddivide la stringa `s` in una lista di sottostringhe usando il delimitatore specificato (di default qualsiasi spazio).

```
righe = "prima riga\nseconda riga\nterza riga"
print(righe.splitlines())
# ["prima riga", "seconda riga", "terza riga"]
parole = "uno, due, tre"
print(parole.split(", "))       # ["uno", "due", "tre"]
```

- `delimiter.join(lista)`: è l'operazione inversa di `split`. Unisce gli elementi di `lista` in una singola stringa, interponendo il delimitatore.

```
elenco = ["pane", "latte", "uova"]
risultato = ", ".join(elenco)
print(risultato)  # "pane, latte, uova"
```

- `s.find(sub)`: restituisce l'indice della prima occorrenza della sottostringa `sub` in `s` (o -1 se non presente). Simile a `s.index(sub)` ma non genera errore se non trovata.

```
frase = "il gatto e il cappello"
print(frase.find("gatto"))    # 3 (posizione in cui inizia "gatto")
print(frase.find("cane"))     # -1 ("cane" non è nella stringa)
```

- Formattazione di stringhe: usando l'operatore `%`, il metodo `format` o, più modernamente, le *f-string*.

```
nome = "Luigi"
eta = 30
# format tradizionale
print("Mi chiamo %s e ho %d anni" % (nome, eta))
# format con metodo format()
print("Mi chiamo {} e ho {} anni".format(nome, eta))
# f-string (Python 3.6+)
print(f"Mi chiamo {nome} e ho {eta} anni")
```

Liste (`list`)

Le liste sono collezioni ordinate e mutabili di elementi. Alcune funzioni e metodi comuni:

- `len(lista)`: restituisce il numero di elementi nella lista.

```
numeri = [10, 20, 30]
print(len(numeri))  # 3
```

- `lista.append(x)`: aggiunge l'elemento `x` alla fine della lista.

```
frutti = ["mela", "banana"]
frutti.append("arancia")
print(frutti)  # ["mela", "banana", "arancia"]
```

- `lista.insert(i, x)`: inserisce l'elemento `x` in posizione `i`, spostando a destra gli altri elementi.

```
frutti = ["mela", "arancia"]
frutti.insert(1, "banana")
print(frutti)  # ["mela", "banana", "arancia"]
```

- `lista.remove(x)`: rimuove la prima occorrenza del valore `x` dalla lista. (Genera errore se `x` non è presente.)

```
numeri = [1, 2, 3, 2]
numeri.remove(2)
print(numeri)  # [1, 3, 2] (rimossa la prima occorrenza di 2)
```

- `lista.pop(i)`: estrae e restituisce l'elemento in posizione `i` (default l'ultimo, `i = -1`).

```
elementi = ['a', 'b', 'c']
ultimo = elementi.pop()
```

```
print(ultimo)      # 'c'
print(elementi)    # ['a', 'b']
```

- `sorted(lista)`: restituisce una nuova lista contenente gli elementi di `lista` in ordine ordinato (senza modificare l'originale). (`lista.sort()` invece ordina in-place la lista stessa e restituisce `None`.)

```
valori = [5, 2, 9, 1]
print(sorted(valori))  # [1, 2, 5, 9]
print(valori)          # [5, 2, 9, 1] (lista originale invariata)
valori.sort()
print(valori)          # [1, 2, 5, 9] (lista ordinata in-place)
```

- **Slicing**: le liste supportano lo slicing (come le stringhe) per ottenere sottosequenze.

```
lettere = ['a', 'b', 'c', 'd', 'e']
print(lettere[1:4])   # ['b', 'c', 'd']  (elementi dagli indici 1 a 3)
print(lettere[:3])    # ['a', 'b', 'c']  (primi 3 elementi)
print(lettere[-2:])   # ['d', 'e']       (ultimi 2 elementi)
```

- **List comprehension**: sintassi concisa per creare liste derivandole da un iterabile.

```
numeri = [1, 2, 3, 4]
quadrati = [x*x for x in numeri]        # [1, 4, 9, 16]
pari = [x for x in numeri if x % 2 == 0]  # [2, 4] (solo elementi pari)
```

Dizionari (`dict`)

I dizionari sono mappe di coppie chiave-valore. Ecco funzioni e metodi base:

- `len(dict)`: numero di coppie chiave-valore nel dizionario.

```
persona = {"nome": "Anna", "età": 25}
print(len(persona))  # 2
```

- `dict.keys()`, `dict.values()`, `dict.items()`: restituiscono rispettivamente un iteratore sulle chiavi, sui valori o sulle coppie (chiave, valore).

```
dati = {"a": 1, "b": 2, "c": 3}
print(list(dati.keys()))    # ['a', 'b', 'c']
print(list(dati.values()))  # [1, 2, 3]
print(list(dati.items()))   # [('a', 1), ('b', 2), ('c', 3)]
```

- `dict.get(chiave, default)`: restituisce il valore associato a `chiave` se esiste, altrimenti ritorna `default` (o `None` se non specificato). Evita KeyError in caso di chiave mancante.

```
contatti = {"Alice": "1234", "Bob": "5678"}
print(contatti.get("Alice"))      # "1234"
print(contatti.get("Charlie"))    # None (chiave non esiste)
print(contatti.get("Charlie", "N/D"))  # "N/D" (valore di default)
```

- Aggiungere o aggiornare elementi: basta assegnare una chiave.

```
persona = {"nome": "Anna", "età": 25}
persona["città"] = "Roma"         # aggiunge nuova coppia
persona["età"] = 26              # aggiorna il valore esistente
print(persona)  # {"nome": "Anna", "età": 26, "città": "Roma"}
```

- Unire due dizionari: in Python 3.9+ si può usare l'operatore unione `|`.

```
d1 = {"x": 1, "y": 2}
d2 = {"y": 3, "z": 4}
d3 = d1 | d2
print(d3)  # {"x": 1, "y": 3, "z": 4}
# In questo caso il valore di 'y' nel dizionario risultato è 3 (di d2,
che sovrascrive d1)
```

- Iterare su un dizionario: si può iterare direttamente sulle chiavi, oppure su items() per chiave e valore.

```
inventario = {"mele": 5, "banane": 3}
for frutto in inventario:
    print(frutto)               # stampa le chiavi ("mele", "banane")
for frutto, quant in inventario.items():
    print(frutto, "->", quant)   # stampa "mele -> 5" etc.
```

Insiemi (`set`)

Gli insiemi sono collezioni non ordinate di elementi unici. Ecco funzioni base:

- `len(set)`: numero di elementi nell'insieme.

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

- `set.add(x)`: aggiunge l'elemento `x` all'insieme (se non presente).

```
elementi = {1, 2}
elementi.add(3)
elementi.add(2)
print(elementi)  # {1, 2, 3} (il duplicato "2" viene ignorato)
```

- `set.remove(x)`: rimuove `x` dall'insieme (KeyError se `x` non esiste). Esiste anche `set.discard(x)` che non genera errore se l'elemento non c'è.

```
elementi = {"a", "b", "c"}
elementi.remove("b")
print(elementi)       # {"a", "c"}
# elementi.remove("d") genererebbe KeyError se decommentato
elementi.discard("d")  # sicuro: non fa nulla se "d" non è presente
```

- Operazioni insiemistiche: `set1.union(set2)` (`|`), `set1.intersection(set2)` (`&`), `set1.difference(set2)` (`-`), restituiscono rispettivamente un nuovo set unione, intersezione o differenza.

```
pari = {2, 4, 6}
dispari = {1, 3, 5}
print(pari | dispari)   # {1,2,3,4,5,6} unione
print(pari & {4,5,6})   # {4,6} intersezione
print(dispari - {1,2})  # {3,5} differenza
```

- Convertire altri tipi in set: `set(lista)` o `set(stringa)`. Utile per eliminare duplicati da una lista.

```
lista_valori = [1, 2, 2, 3, 3, 3]
unici = set(lista_valori)
print(unici)           # {1, 2, 3}
```

Tuple (`tuple`)

Le tuple sono sequenze immutabili di elementi. Operazioni principali (molto simili a quelle delle liste, ma le tuple non possono essere modificate una volta create):

- Creazione di una tupla: usare parentesi tonde o semplicemente separare gli elementi da virgole.

```
t1 = (10, 20, 30)
t2 = "a", "b", "c"   # anche senza parentesi
singleton = (42,)    # tupla di un elemento: nota la virgola
```

- Accesso per indice e slicing funzionano come per le liste.

```
t = ('p', 'y', 't', 'h', 'o', 'n')
print(t[1])    # 'y'
print(t[-1])   # 'n'
print(t[2:5])  # ('t', 'h', 'o')
```

- `len(tuple)`: numero di elementi.

```
t = (1, 2, 3)
print(len(t))  # 3
```

- Metodi count e index: `tuple.count(x)` conta le occorrenze di `x` nella tupla, `tuple.index(x)` dà l'indice della prima occorrenza di `x`.

```
t = (1, 2, 2, 3)
print(t.count(2))   # 2 (il valore 2 appare due volte)
print(t.index(3))   # 3 (il valore 3 è all'indice 3)
```

- *Unpacking* di una tupla: si possono assegnare i valori di una tupla a variabili in un'unica istruzione.

```
coords = (10, 20)
x, y = coords
print(x)  # 10
print(y)  # 20
# Swap di variabili usando tuple unpacking:
a, b = 1, 2
a, b = b, a
print(a, b)  # 2 1
```

Numeri Interi (`int`)

Gli interi in Python rappresentano numeri interi di precisione arbitraria (possono crescere oltre i limiti dei tipici 32/64 bit). Operazioni e funzioni utili:

- Operazioni aritmetiche di base: `+`, `-`, `*`, `/` (divisione float), `//` (divisione intera), `%` (modulo), `**` (esponenziazione).

```
a = 7
b = 3
print(a + b)   # 10
print(a - b)   # 4
print(a * b)   # 21
print(a / b)   # 2.333...
print(a // b)  # 2   (divisione intera)
```

```
print(a % b)   # 1   (resto)
print(a ** b)  # 343 (7 al cubo)
```

- Conversione a intero: `int(x)` prova a convertire `x` in numero intero (arrotondando per difetto se `x` è float, o parsando stringhe numeriche).

```
print(int(3.9))       # 3 (tronca la parte decimale)
print(int("15"))      # 15 (stringa -> int)
print(int(True))      # 1  (True e False convertiti in 1 e 0)
```

- Funzioni matematiche utili: `abs(x)` valore assoluto, `divmod(a, b)` restituisce coppia (quoziente, resto) della divisione intera, `pow(x, y, mod)` calcola (x**y) % mod in modo efficiente.

```
print(abs(-5))          # 5
print(divmod(17, 5))    # (3, 2)  -> 17//5 = 3, 17%5 = 2
print(pow(2, 10, 7))    # 2^10 % 7 = 1024 % 7 = 2
```

- Rappresentazioni numeriche: `bin(x)`, `hex(x)`, `oct(x)` per ottenere stringhe con la rappresentazione binaria, esadecimale, ottale del numero.

```
n = 42
print(bin(n))  # '0b101010'
print(hex(n))  # '0x2a'
print(oct(n))  # '0o52'
```

Numeri Floating Point (`float`)

I numeri float rappresentano numeri in virgola mobile (a precisione doppia ~64 bit). Funzioni e operazioni base:

- Operazioni aritmetiche: stesse di `int` (`+`, `-`, `*`, `/`, `**`, etc.). Attenzione: la divisione `/` tra interi produce un float.
- Conversione: `float(x)` converte `x` in virgola mobile (utile per stringhe o interi).

```
print(float(5))      # 5.0
print(float("3.14")) # 3.14
```

- `round(x, n)`: arrotonda il numero `x` alla precisione di `n` cifre decimali (senza `n` arrotonda all'intero più vicino).

```
val = 3.14159
print(round(val, 2))  # 3.14
print(round(val))     # 3
```

- Funzioni matematiche: nel modulo `math` troviamo molte funzioni (trigonometriche, logaritmi, etc.) e costanti:

```
import math
print(math.sqrt(16))   # 4.0 (radice quadrata)
print(math.floor(2.7)) # 2   (arrotondamento giù)
print(math.ceil(2.1))  # 3   (arrotondamento su)
print(math.pi)         # 3.1415926535...
```

- **Attenzione**: i float sono soggetti a errori di rappresentazione binaria. Esempio classico:

```
print(0.1 + 0.2)  # 0.30000000000000004 (non esattamente 0.3)
```

Per confronti, può essere utile definire una tolleranza oppure usare `math.isclose`.

Booleani (`bool`)

I booleani rappresentano il valore di verità `True` o `False` (sottoclasse di int, dove True=1, False=0). Operazioni base:

- Operatori logici: `and`, `or`, `not`.

```
vero = True
falso = False
print(vero and falso)  # False (vero E falso)
print(vero or falso)   # True  (vero O falso)
print(not vero)        # False (NON vero)
```

- Conversione a bool: `bool(x)` determina il valore booleano di `x` (False per 0, 0.0, stringa vuota, lista vuota, None; True per il resto).

```
print(bool(0))        # False
print(bool(42))       # True
print(bool(""))       # False (stringa vuota)
print(bool("ciao"))   # True  (stringa non vuota)
print(bool([1,2,3]))  # True  (lista non vuota)
```

- `any(iterabile)`: restituisce True se **almeno uno** degli elementi dell'iterabile è True (in senso booleano). `all(iterabile)` restituisce True se **tutti** gli elementi sono True.

```
valori = [0, "", 5, None]
print(any(valori))  # True  (perché c'è un elemento 5 che è True)
print(all(valori))  # False (perché alcuni elementi valutano False)
```

- Comparazioni: `==`, `!=`, `<`, `<=`, `>`, `>=` funzionano per confrontare numeri e altri tipi comparabili. Inoltre Python consente confronti concatenati:

```
x = 5
print(0 < x < 10)   # True (x è compreso tra 0 e 10)
print(x == 5 or x == 6)  # True (x è 5 oppure 6)
```

2. 100 Attrezzi Python per Problemi Quotidiani

Di seguito sono presentati 100 snippet di codice Python – veri e propri "attrezzi" utili – pensati per risolvere compiti comuni nella programmazione quotidiana. Gli esempi includono automazioni frequenti, utilità per il debugging, analisi statica del codice, gestione di file CSV/JSON/XML/YAML, script per la produttività, generatori di template, verifiche di performance, miglioramenti al logging e sistemi di notifica. Ogni snippet è numerato, dotato di un breve commento descrittivo e di codice esemplificativo.

Automazioni Comuni

1. Rinominare file in blocco in una cartella – Aggiunge un prefisso a tutti i file di una directory (utile per rinominare gruppi di file in modo uniforme).

```
import os

prefisso = "ARCHIVIO_"
cartella = "/path/alla/cartella"
for nome_file in os.listdir(cartella):
    vecchio = os.path.join(cartella, nome_file)
    if os.path.isfile(vecchio):
        nuovo_nome = prefisso + nome_file
        nuovo = os.path.join(cartella, nuovo_nome)
        os.rename(vecchio, nuovo)  # Rinomina il file
        print(f"Rinominato {nome_file} -> {nuovo_nome}")
```

2. Inviare un'email tramite SMTP – Invia un'email semplice utilizzando il server SMTP (richiede credenziali e server SMTP validi).

```
import smtplib
from email.message import EmailMessage

# Configurazione email
msg = EmailMessage()
```

```
msg["Subject"] = "Oggetto di Test"
msg["From"] = "mittente@example.com"
msg["To"] = "destinatario@example.com"
msg.set_content("Corpo del messaggio di prova.")

# Invio tramite SMTP (esempio con TLS su porta 587)
server_smtp = "smtp.example.com"
username = "utente"
password = "password"
with smtplib.SMTP(server_smtp, 587) as server:
    server.starttls()               # Avvia connessione sicura TLS
    server.login(username, password)
    server.send_message(msg)
    print("Email inviata con successo.")
```

3. Validare un file JSON – Verifica se una stringa (o file) contiene JSON valido, tentando il parsing.

```
import json

def is_json_valido(testo: str) -> bool:
    try:
        json.loads(testo)
        return True
    except json.JSONDecodeError:
        return False

dati = '{"nome": "Anna", "eta": 25}'
print(is_json_valido(dati))         # True, stringa JSON valida
print(is_json_valido("testo"))      # False, non è JSON valido
# Su file:
# with open("file.json") as f: is_json_valido(f.read())
```

4. Validare un indirizzo email con regex – Utilizza un'espressione regolare per controllare se una stringa ha il formato di un indirizzo email.

```
import re

pattern_email = re.compile(r'^[\w\.-]+@[\w\.-]+\.\w+$')
def email_valida(email: str) -> bool:
    return re.match(pattern_email, email) is not None

print(email_valida("user@example.com"))   # True
print(email_valida("nome.cognome@dominio"))  # False (manca il TLD)
```

5. Scaricare un file da una URL – Effettua il download di un file remoto tramite HTTP. (Richiede la libreria `requests`: `pip install requests`).

```
import requests

url = "https://www.example.com/file.txt"
dest = "downloaded_file.txt"
try:
    response = requests.get(url)
    response.raise_for_status()            # Verifica esito OK
    with open(dest, "wb") as f:
        f.write(response.content)
    print(f"File scaricato e salvato come {dest}")
except Exception as e:
    print(f"Errore nel download: {e}")
```

6. Effettuare il web scraping di una pagina web – Scarica il contenuto HTML di una pagina e ne estrae dati con BeautifulSoup (richiede `pip install beautifulsoup4 requests`).

```
import requests
from bs4 import BeautifulSoup

url = "https://www.example.com"
response = requests.get(url)
soup = BeautifulSoup(response.text, 'html.parser')
titolo = soup.find('title').text   # Estrae il testo del tag <title>
print(f"Titolo della pagina: {titolo}")
# (Ulteriori estrazioni di elementi possono essere effettuate con selettori
CSS o metodi find/find_all)
```

7. Leggere un file di configurazione INI – Utilizza `configparser` per leggere valori da un file `.ini` con sezioni e chiavi.

```
import configparser

config = configparser.ConfigParser()
config.read("config.ini")
# Esempio di file config.ini:
# [Database]
# host = localhost
# port = 5432
host_db = config.get("Database", "host")
port_db = config.getint("Database", "port")
print(f"DB Host: {host_db}, Porta: {port_db}")
```

8. Ricerca e sostituzione di testo in più file – Cerca una stringa in tutti i file di una cartella e la sostituisce con un'altra (ad esempio per modificare un nome di variabile in più file).

```python
import glob

cartella = "./progetto"
testo_da_cercare = "VERSIONE=1.0"
testo_sostitutivo = "VERSIONE=1.1"

for filepath in glob.glob(f"{cartella}/**/*.py", recursive=True):
    with open(filepath, "r+") as f:
        contenuto = f.read()
        if testo_da_cercare in contenuto:
            nuovo_contenuto = contenuto.replace(testo_da_cercare,
testo_sostitutivo)
            f.seek(0)
            f.write(nuovo_contenuto)
            f.truncate()
            print(f"Modificato {filepath}")
```

9. Monitorare modifiche in un file (polling) – Controlla periodicamente se un file è stato modificato (es. per ricaricare configurazioni).

```python
import time, os

file_path = "dati.txt"
intervallo = 5  # secondi tra i controlli
ultimo_modifica = os.path.getmtime(file_path)

print(f"Monitoraggio modifiche su {file_path}...")
while True:
    time.sleep(intervallo)
    try:
        mod_attuale = os.path.getmtime(file_path)
    except FileNotFoundError:
        print("File rimosso o non trovato!")
        break
    if mod_attuale != ultimo_modifica:
        print("Il file è stato modificato!")
        ultimo_modifica = mod_attuale
```

10. Creare un semplice menu interattivo CLI – Mostra un menu testuale e gestisce la scelta dell'utente in input.

```python
def mostra_menu():
    print("Scegli un'opzione:")
    print("1) Saluta")
    print("2) Esci")

while True:
```

```
    mostra_menu()
    scelta = input(">> ")
    if scelta == "1":
        nome = input("Come ti chiami? ")
        print(f"Ciao, {nome}!")
    elif scelta == "2":
        print("Uscita...")
        break
    else:
        print("Scelta non valida, riprova.")
```

11. Ridimensionare immagini in una cartella – Usa la libreria Pillow per ridimensionare tutte le immagini JPEG in una directory (richiede `pip install Pillow`).

```
from PIL import Image
import os

cartella = "./immagini"
larghezza = 800

for nome_file in os.listdir(cartella):
    if nome_file.lower().endswith(".jpg"):
        percorso = os.path.join(cartella, nome_file)
        img = Image.open(percorso)
        rapporto = larghezza / float(img.width)
        altezza = int(img.height * rapporto)
        img_resized = img.resize((larghezza, altezza))
        img_resized.save(os.path.join(cartella, f"resized_{nome_file}"))
        print(f"{nome_file} ridimensionata a {larghezza}px di larghezza")
```

12. Comprimere una cartella in un file ZIP (backup) – Crea un archivio zip contenente il contenuto di una directory, utilizzando `shutil`.

```
import shutil

cartella_da_comprimere = "./cartella_dati"
shutil.make_archive("backup_dati", 'zip', cartella_da_comprimere)
print("Backup ZIP creato come backup_dati.zip")
```

13. Verificare la connessione Internet – Tenta di effettuare una richiesta HTTP per capire se c'è connessione (ad esempio verso Google).

```
import urllib.request

def internet_disponibile(url="http://www.google.com"):
    try:
```

```
        urllib.request.urlopen(url, timeout=5)
        return True
    except Exception:
        return False

if internet_disponibile():
    print("Connessione Internet attiva.")
else:
    print("Connessione Internet NON disponibile.")
```

14. Schedulare un'operazione periodica (cron-like) – Usa la libreria `schedule` per eseguire una funzione ad intervalli regolari (richiede `pip install schedule`).

```
import schedule
import time

def lavoro():
    print("Esecuzione del task programmato...")

# Pianifica il task ogni 10 secondi
schedule.every(10).seconds.do(lavoro)

print("Avvio scheduler. Premi CTRL+C per interrompere.")
while True:
    schedule.run_pending()
    time.sleep(1)
```

15. Eseguire un comando di sistema e catturare l'output – Utilizza il modulo `subprocess` per eseguire un comando del sistema operativo e leggerne l'output (stdout e stderr).

```
import subprocess

comando = ["echo", "Hello World"]  # Comando da eseguire (qui un semplice
echo)
try:
    risultato = subprocess.run(comando, capture_output=True, text=True,
check=True)
    print("Output:", risultato.stdout.strip())
except subprocess.CalledProcessError as e:
    print("Errore nell'esecuzione:", e)
```

16. Aprire un URL nel browser predefinito – Usa il modulo `webbrowser` per lanciare il browser di default con un certo URL.

```
import webbrowser
```

```
url = "https://www.python.org"
webbrowser.open(url)
```

17. Eliminare file temporanei in una directory – Rimuove tutti i file con estensione specifica (es: `.tmp`) in una cartella e sottocartelle.

```
import os

cartella = "./progetto"
estensione_da_eliminare = ".tmp"
for root, dirs, files in os.walk(cartella):
    for nome_file in files:
        if nome_file.endswith(extensione_da_eliminare):
            percorso = os.path.join(root, nome_file)
            os.remove(percorso)
            print(f"Eliminato {percorso}")
```

18. Effettuare richieste HTTP (GET/POST) ad un API con gestione JSON – Usa `requests` per inviare una richiesta GET o POST ad un'API REST e gestisce la risposta JSON (richiede `pip install requests`).

```
import requests

url = "https://api.example.com/data"
payload = {"param": 42}
try:
    # Esempio di richiesta GET
    resp = requests.get(url, params=payload)
    resp.raise_for_status()
    dati = resp.json()  # Parse JSON della risposta
    print("Dati ricevuti:", dati)
    # Esempio di richiesta POST
    resp = requests.post(url, json=payload)
    resp.raise_for_status()
    print("Risposta POST:", resp.json())
except requests.RequestException as e:
    print("Errore nella richiesta:", e)
```

19. Caricare variabili d'ambiente da un file .env – Legge un file `.env` contenente coppie CHIAVE=VALORE e le imposta come variabili d'ambiente (richiede la libreria `python-dotenv`: `pip install python-dotenv`).

```
from dotenv import load_dotenv
import os

load_dotenv(".env")  # Carica le variabili dal file .env nella directory
```

```
corrente
# Ora è possibile accedere alle variabili tramite os.getenv o os.environ
db_password = os.getenv("DB_PASSWORD")
print("Password DB caricata:", db_password)
```

Utility di Debugging

20. Decoratore per loggare chiamate di funzione e argomenti – Definisce un decoratore che, applicato a una funzione, stampa automaticamente il nome della funzione chiamata e i parametri passati, utile per tracciare l'esecuzione.

```
def log_calls(func):
    def wrapper(*args, **kwargs):
        print(f"Chiamata funzione {func.__name__} con args={args},
kwargs={kwargs}")
        result = func(*args, **kwargs)
        print(f"{func.__name__} ha restituito {result}")
        return result
    return wrapper

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

somma(5, 7)
# Output:
# Chiamata funzione somma con args=(5, 7), kwargs={}
# somma ha restituito 12
```

21. Context manager per misurare il tempo di esecuzione di un blocco di codice – Implementa un context manager personalizzato che, usato con `with`, misura il tempo impiegato nel blocco di codice al suo interno.

```
import time

class Timer:
    def __enter__(self):
        self.inizio = time.time()
        return self
    def __exit__(self, exc_type, exc_val, exc_tb):
        fine = time.time()
        print(f"Tempo trascorso: {fine - self.inizio:.4f} secondi")

# Utilizzo:
with Timer():
    # Blocco di codice di cui misurare la durata
```

```
    time.sleep(1)           # simulazione di operazione costosa
    lista = [x**2 for x in range(1000000)]
```

22. Uso di pdb per debug interattivo (inserire breakpoint) – Mostra come inserire manualmente un breakpoint per avviare il debugger interattivo `pdb` durante l'esecuzione di uno script.

```
import pdb

for i in range(5):
    print(f"Iterazione {i}")
    if i == 2:
        pdb.set_trace()  # Pausa qui ed entra in modalità interattiva
(debugger)
    # All'arrivo a i == 2, si aprirà il prompt pdb dove poter ispezionare
variabili passo-passo
```

23. Stampare lo stack trace completo in caso di eccezione – In una clausola `except`, utilizza il modulo `traceback` per loggare la traccia completa dell'errore (utile per debug approfondito).

```
import traceback

try:
    risultato = 10 / 0
except Exception as e:
    print("Errore catturato:", e)
    traceback.print_exc()  # Stampa dettagliata dello stack trace
dell'eccezione
```

24. Uso avanzato del logging per debug (settaggio livello DEBUG) – Configura il modulo `logging` in modo da stampare messaggi di debug e info con timestamp, facilitando il debug senza usare print.

```
import logging

logging.basicConfig(level=logging.DEBUG,
                    format="%(asctime)s [%(levelname)s] %(message)s")
logging.debug("Messaggio di debug, variabile x = ...")
logging.info("Messaggio informativo")
logging.error("Messaggio di errore")
```

25. Contare il numero di chiamate a una funzione (decoratore) – Crea un decoratore che conta quante volte una funzione viene invocata, utile per monitorare utilizzi.

```
def conta_chiamate(func):
    func.counter = 0
```

```
    def wrapper(*args, **kwargs):
        func.counter += 1
        print(f"{func.__name__} chiamata #{func.counter}")
        return func(*args, **kwargs)
    return wrapper

@conta_chiamate
def saluta(nome):
    print(f"Ciao {nome}!")

saluta("Mondo")
saluta("Python")
# Output:
# saluta chiamata #1
# Ciao Mondo!
# saluta chiamata #2
# Ciao Python!
```

26. Tracciare l'utilizzo di memoria con tracemalloc – Usa il modulo `tracemalloc` per avviare il tracking della memoria allocata e visualizzare quanta memoria è stata utilizzata in un blocco di codice.

```
import tracemalloc

tracemalloc.start()
# ... codice di cui tracciare l'allocazione ...
lista = [bytes(1000) for _ in range(10000)]  # alloca 10 million bytes (~10
MB)
snapshot = tracemalloc.take_snapshot()
top_stats = snapshot.statistics('filename')
total = sum(stat.size for stat in top_stats)
print(f"Memoria totale allocata: {total / 1024:.1f} KB")
tracemalloc.stop()
```

27. Pretty print di strutture dati complesse (pprint) – Utilizza `pprint` per stampare in modo leggibile strutture dati annidate (dizionari, liste di liste, etc.), utile durante il debugging.

```
import pprint

dati = {"utenti": [{"nome": "Alice", "età": 30, "hobby": ["lettura",
"tennis"]},
                   {"nome": "Bob", "età": 25, "hobby": ["musica",
"viaggi"]}]}
pprint.pprint(dati, width=40)
# Stampa ben formattata del dizionario, andando a capo per elementi annidati
```

28. Verificare la profondità di ricorsione con sys.getrecursionlimit – Mostra il limite di ricorsione corrente e come modificarlo (operazione da fare con cautela).

```python
import sys

print("Limite di ricorsione:", sys.getrecursionlimit())
sys.setrecursionlimit(2000)
print("Nuovo limite di ricorsione:", sys.getrecursionlimit())

# Funzione ricorsiva di esempio
def fattoriale(n):
    return 1 if n == 0 else n * fattoriale(n-1)

print(fattoriale(5))  # 120
# (Attenzione: chiamare fattoriale con n molto grande può causare
RecursionError se supera il nuovo limite)
```

29. Ispezionare oggetti dinamicamente (dir() e inspect) – Usa `dir()` e funzioni del modulo `inspect` per scoprire attributi, sorgente o documentazione di un oggetto a runtime.

```python
import inspect

class Prova:
    def metodo(self):
        """Docstring di esempio."""
        pass

oggetto = Prova()
print(dir(oggetto))             # Elenca attributi/metodi dell'oggetto
print(inspect.getdoc(Prova.metodo))  # Ottiene la docstring del metodo
print(inspect.getsource(Prova.metodo))  # Mostra il codice sorgente del
metodo
```

30. Tracciare l'esecuzione del codice riga per riga (modulo trace) – Utilizza il modulo `trace` per eseguire uno script Python tracciando ogni riga eseguita (utile per capire il flusso).

```python
import trace

tracer = trace.Trace(trace=True)   # trace=True per tracciare esecuzione
def saluta(nome):
    msg = f"Ciao, {nome}"
    print(msg)
tracer.run('saluta("Python")')
# Questo stamperà ogni linea eseguita e le chiamate di funzione durante
l'esecuzione di saluta("Python").
```

Analisi Statica del Codice

31. Contare il numero di linee di codice Python in un progetto – Visita ricorsivamente una directory contando le righe di tutti i file `.py`.

```
import os

def conta_linee_cartella(percorso):
    total = 0
    for root, dirs, files in os.walk(percorso):
        for fname in files:
            if fname.endswith(".py"):
                path = os.path.join(root, fname)
                with open(path, "r", encoding="utf8", errors="ignore") as f:
                    for _ in f:
                        total += 1
    return total

cartella_progetto = "./mia_cartella"
print("Linee di codice totali:",
      conta_linee_cartella(cartella_progetto))
```

32. Trovare tutti i TODO/FIXME nel codice – Cerca all'interno dei file di codice delle parole chiave "TODO" o "FIXME" per individuare promemoria o parti di codice da rivedere.

```
import re, glob

for filepath in glob.glob("**/*.py", recursive=True):
    with open(filepath, "r", encoding="utf8", errors="ignore") as f:
        for num_linea, riga in enumerate(f, start=1):
            if re.search(r"\b(TODO|FIXME)\b", riga):
                print(f"{filepath}:{num_linea}: {riga.strip()}")
```

33. Verificare la presenza di import inutilizzati – Semplice analisi che segnala se un modulo è importato ma non viene effettivamente utilizzato nel file (richiede miglioramenti per casi complessi).

```
import ast

def import_non_usati(filepath):
    with open(filepath, "r") as f:
        tree = ast.parse(f.read(), filename=filepath)
    imports = [node.names[0].name for node in ast.walk(tree) if
isinstance(node, ast.Import)]
    imports += [node.module for node in ast.walk(tree) if isinstance(node,
ast.ImportFrom)]
    # Trova nomi usati nel codice
    names_used = {node.id for node in ast.walk(tree) if isinstance(node,
```

```
ast.Name)}
    # Moduli importati ma il cui nome base non compare mai nel codice:
    non_usati = [imp for imp in imports if imp.split('.')[0] not in
names_used]
    return non_usati

file_py = "script.py"
print("Import non usati:", import_non_usati(file_py))
```

34. Identificare funzioni non chiamate (analisi AST) – Trova definizioni di funzioni in un file Python e verifica se i loro nomi non compaiono mai come chiamate.

```
import ast

def funzioni_non_chiamate(filepath):
    with open(filepath, "r") as f:
        tree = ast.parse(f.read(), filename=filepath)
    funzioni_definite = [node.name for node in ast.walk(tree) if
isinstance(node, ast.FunctionDef)]
    chiamate = [node.func.id for node in ast.walk(tree)
                if isinstance(node, ast.Call) and isinstance(node.func,
ast.Name)]
    return [fn for fn in funzioni_definite if fn not in chiamate]

print("Funzioni non chiamate:",
      funzioni_non_chiamate("mio_script.py"))
```

35. Elencare tutte le funzioni e classi in un file Python (AST) – Utilizza l'AST per estrarre i nomi di tutte le funzioni e classi definite in un modulo.

```
import ast

with open("mio_modulo.py", "r") as f:
    tree = ast.parse(f.read())
funzioni = [node.name for node in ast.walk(tree) if isinstance(node,
ast.FunctionDef)]
classi = [node.name for node in ast.walk(tree) if isinstance(node,
ast.ClassDef)]
print("Funzioni trovate:", funzioni)
print("Classi trovate:", classi)
```

36. Eseguire il linter PEP8 (flake8) da script – Lancia `flake8` (tool esterno per controlli PEP8) tramite `subprocess` e stampa l'output dei problemi di stile riscontrati. (Richiede `pip install flake8`).

```
import subprocess
```

```
try:
    risultato = subprocess.run(["flake8", "--max-line-length=88", "."],
capture_output=True, text=True, check=True)
    output = risultato.stdout.strip()
    if output:
        print("Problemi di stile trovati:\n", output)
    else:
        print("Nessun problema di stile rilevato.")
except FileNotFoundError:
    print("flake8 non installato. Installalo con 'pip install flake8'.")
```

37. Calcolare la complessità ciclomatica con radon – Usa la libreria `radon` per calcolare la complessità del codice (metrica di manutentibilità). (Richiede `pip install radon`).

```
import subprocess

file = "mio_script.py"
try:
    result = subprocess.run(["radon", "cc", "-s", "-a", file],
capture_output=True, text=True)
    print(result.stdout)  # Stampa la complessità ciclomatica di ogni
funzione e un totale (A, B, C... etc.)
except FileNotFoundError:
    print("Radon non installato. Installalo con 'pip install radon'.")
```

38. Controllare la presenza di tabulazioni vs spazi nelle indentazioni – Scansiona i file Python per verificare se l'indentazione è consistente (solo spazi o solo tab) e segnala eventuali mix che possono causare errori di indentazione.

```
for filepath in glob.glob("**/*.py", recursive=True):
    with open(filepath, "r", encoding="utf8", errors="ignore") as f:
        lines = f.readlines()
        for num, line in enumerate(lines, 1):
            if "\t" in line and line.lstrip(" \t").startswith(" "):
                print(f"Indentazione mista (tab/spazi) in {filepath} linea
{num}")
```

39. Analizzare le dipendenze delle importazioni in un progetto – Costruisce l'elenco di moduli importati in ciascun file di un progetto, per avere un quadro delle dipendenze interne ed esterne.

```
import ast

dipendenze = {}
for filepath in glob.glob("**/*.py", recursive=True):
    with open(filepath, "r", encoding="utf8", errors="ignore") as f:
        tree = ast.parse(f.read(), filename=filepath)
```

```
    imports = []
    for node in ast.walk(tree):
        if isinstance(node, ast.Import):
            imports += [alias.name for alias in node.names]
        elif isinstance(node, ast.ImportFrom):
            mod = node.module
            if mod: imports.append(mod.split('.')[0])
    dipendenze[filepath] = sorted(set(imports))

for file, mods in dipendenze.items():
    print(f"{file}: {mods}")
```

40. Trovare variabili globali definite in uno script – Segnala tutte le variabili assegnate a livello globale in un file Python (che potrebbero essere da evitare in favore di costanti o passaggio parametri).

```
import ast

with open("script.py", "r") as f:
    tree = ast.parse(f.read())
assegnazioni_globali = []
for node in ast.walk(tree):
    if isinstance(node, ast.Assign):
        for target in node.targets:
            if isinstance(target, ast.Name):
                # Esclude import e definizioni di funzioni/classi
                if not any(isinstance(parent, (ast.FunctionDef,
ast.ClassDef)) for parent in ast.walk(node)):
                    assegnazioni_globali.append(target.id)
assegnazioni_globali = sorted(set(assegnazioni_globali))
print("Variabili globali assegnate:", assegnazioni_globali)
```

41. Verificare se ci sono linee di codice troppo lunghe – Controlla nei file se esistono linee che eccedono una certa lunghezza (es. 79 caratteri, standard PEP8).

```
max_len = 79
for filepath in glob.glob("**/*.py", recursive=True):
    with open(filepath, "r", encoding="utf8", errors="ignore") as f:
        for num_linea, linea in enumerate(f, start=1):
            if len(linea.rstrip("\n")) > max_len:
                print(f"{filepath}:{num_linea} - linea troppo lunga
({len(linea)} caratteri)")
```

42. Verificare che ogni file termini con una newline finale – Assicura che i file di codice abbiano un fine linea a fine file (buona pratica per strumenti di controllo versione e POSIX).

```
for filepath in glob.glob("**/*.py", recursive=True):
    with open(filepath, "rb") as f:
        if f.read().endswith(b'\n'):
            pass  # OK
        else:
            print(f"{filepath} non termina con newline finale")
```

Gestione Formati di File Comuni

43. Leggere un file JSON e accedere ai dati – Carica un file JSON in un oggetto Python (di solito dizionari annidati e liste) utilizzando il modulo `json`.

```
import json

with open("dati.json", "r", encoding="utf8") as f:
    dati = json.load(f)
# Supponendo che dati.json contenga: {"utenti": [{"nome": "Alice"}, {"nome":
"Bob"}]}
utenti = dati.get("utenti", [])
for utente in utenti:
    print("Nome utente:", utente.get("nome"))
```

44. Scrivere dati Python su un file JSON (serializzazione) – Converte un dizionario/oggetto Python in formato JSON e lo salva su file.

```
import json

risultati = {"somma": 45, "media": 15.0, "valori": [10, 20, 15]}
with open("risultati.json", "w", encoding="utf8") as f:
    json.dump(risultati, f, ensure_ascii=False, indent=4)
print("Dati salvati in risultati.json")
```

45. Leggere un file CSV e trasformarlo in elenco di dizionari – Usa il modulo `csv` per leggere un file CSV dove la prima riga contiene i nomi delle colonne, producendo un dizionario per ogni riga.

```
import csv

with open("dati.csv", newline="", encoding="utf8") as csvfile:
    reader = csv.DictReader(csvfile)
    righe = [riga for riga in reader]
    print(righe)
    # Ogni elemento di righe è un dizionario {colonna: valore}
    # Esempio output: [{'Nome': 'Anna', 'Età': '30'}, {'Nome': 'Bob', 'Età':
'25'}]
```

46. Scrivere una lista di dizionari su un file CSV – Salva un elenco di dizionari (tutti con le stesse chiavi) in un file CSV con intestazione.

```
import csv

records = [
    {"Nome": "Anna", "Età": 30},
    {"Nome": "Bob", "Età": 25}
]
with open("output.csv", "w", newline="", encoding="utf8") as csvfile:
    fieldnames = records[0].keys()
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
    writer.writeheader()
    writer.writerows(records)
print("File CSV creato: output.csv")
```

47. Leggere un file YAML – Utilizza la libreria PyYAML per caricare un file YAML in una struttura Python (richiede `pip install pyyaml`).

```
import yaml

with open("config.yaml", "r") as f:
    dati = yaml.safe_load(f)
# Esempio: se config.yaml contiene:
# server:
#   host: localhost
#   port: 8080
print(dati["server"]["host"])  # "localhost"
print(dati["server"]["port"])  # 8080
```

48. Scrivere dati su un file YAML – Serializza un dizionario Python in formato YAML e lo salva su file (utilizzando PyYAML).

```
import yaml

config = {
    "server": {"host": "localhost", "port": 8080},
    "debug": True
}
with open("config_out.yaml", "w") as f:
    yaml.safe_dump(config, f)
print("File config_out.yaml generato.")
```

49. Analizzare un file XML ed estrarre informazioni specifiche – Usa `xml.etree.ElementTree` (in Python standard) per effettuare il parsing di XML e trovare elementi.

```python
import xml.etree.ElementTree as ET

tree = ET.parse('documento.xml')
root = tree.getroot()
# Esempio: estraiamo tutti i titoli di elementi <item><title> in un feed RSS
titoli = [elem.text for elem in root.findall(".//item/title")]
for titolo in titoli:
    print("Titolo:", titolo)
```

50. Convertire un CSV in JSON – Legge un file CSV e lo converte in una struttura JSON (lista di dizionari), salvandolo su file.

```python
import csv, json

csv_file = "dati.csv"
json_file = "dati.json"
with open(csv_file, newline="", encoding="utf8") as f:
    reader = csv.DictReader(f)
    righe = list(reader)
with open(json_file, "w", encoding="utf8") as f:
    json.dump(righe, f, indent=4, ensure_ascii=False)
print(f"Convertito {csv_file} in {json_file}")
```

51. Unire più file CSV in un unico file – Legge tutti i file CSV in una cartella e li combina (stesse colonne) in un unico CSV di output.

```python
import csv, glob

file_csv = sorted(glob.glob("dataset/*.csv"))
with open("unione.csv", "w", newline="", encoding="utf8") as output:
    writer = None
    for idx, file in enumerate(file_csv):
        with open(file, newline="", encoding="utf8") as f:
            reader = csv.reader(f)
            intestazione = next(reader)
            if idx == 0:
                writer = csv.writer(output)
                writer.writerow(intestazione)  # scrive header solo una volta
            for riga in reader:
                writer.writerow(riga)
print("File CSV unificato generato: unione.csv")
```

52. Leggere e scrivere file Excel (XLSX) – Usa `openpyxl` per caricare un file Excel e salvarne uno nuovo (richiede `pip install openpyxl`).

```python
from openpyxl import load_workbook, Workbook

# Lettura di un file Excel esistente
wb = load_workbook("input.xlsx")
foglio = wb.active
valore = foglio["A1"].value
print("Valore cella A1:", valore)

# Creazione di un nuovo file Excel con dati
nuovo_wb = Workbook()
foglio2 = nuovo_wb.active
foglio2["A1"] = "Nome"
foglio2["B1"] = "Età"
foglio2.append(["Alice", 30])
foglio2.append(["Bob", 25])
nuovo_wb.save("output.xlsx")
print("File Excel output.xlsx salvato.")
```

Gestione del Tempo e Produttività

53. Implementare un timer countdown (conto alla rovescia) – Effettua un semplice conto alla rovescia da un numero di secondi, mostrando il tempo rimanente.

```python
import time

secondi = 5
for rimanenti in range(secondi, 0, -1):
    print(f"Tempo rimanente: {rimanenti} secondi")
    time.sleep(1)
print("Tempo scaduto!")
```

54. Realizzare un semplice timer Pomodoro – Un timer che alterna 25 minuti di lavoro e 5 minuti di pausa (Pomodoro Technique). Qui ridotto per esempio (5 secondi lavoro, 2 secondi pausa).

```python
import time

def pomodoro(cicli=1):
    for ciclo in range(1, cicli+1):
        print(f"Pomodoro {ciclo} iniziato (fase lavoro)...")
        time.sleep(5)  # 25*60 secondi in un caso reale
        print("Tempo di fare una pausa breve!")
        time.sleep(2)  # 5*60 secondi pausa in un caso reale
        print(f"Pomodoro {ciclo} completato.\n")

pomodoro(cicli=2)
```

55. Schedulare attività a intervalli regolari (usando schedule) – Simile allo snippet 14, pianifica l'esecuzione periodica di funzioni con il modulo `schedule` (ad esempio ogni giorno alle 9:00).

```
import schedule, time

def rapporto_giornaliero():
    print("Generazione rapporto giornaliero...")

# Pianifica l'attività ogni giorno alle 09:00
schedule.every().day.at("09:00").do(rapporto_giornaliero)

while True:
    schedule.run_pending()
    time.sleep(60)  # controllo ogni minuto
```

56. Misurare il tempo di esecuzione di una funzione (decoratore con time) – Un decoratore che calcola il tempo impiegato da una funzione ad ogni chiamata.

```
import time

def misura_tempo(func):
    def wrapper(*args, **kwargs):
        start = time.perf_counter()
        result = func(*args, **kwargs)
        end = time.perf_counter()
        print(f"{func.__name__} eseguita in {end - start:.6f} secondi")
        return result
    return wrapper

@misura_tempo
def calcola_somma(n):
    return sum(range(n))

calcola_somma(1000000)
```

57. Mostrare una barra di avanzamento per un loop (usando tqdm) – Integra la libreria `tqdm` per visualizzare una progress bar su terminale durante l'iterazione di un loop (richiede `pip install tqdm`).

```
from tqdm import tqdm
import time

for i in tqdm(range(100), desc="Elaborazione"):
    time.sleep(0.01)  # simulazione lavoro
```

58. Convertire una stringa di data/ora in un oggetto datetime – Usa `datetime.strptime` per fare il parsing di una stringa formattata in un oggetto data/ora Python.

```
from datetime import datetime

data_str = "2025-07-08 14:00:00"
formato = "%Y-%m-%d %H:%M:%S"
data_obj = datetime.strptime(data_str, formato)
print(data_obj, type(data_obj))
# Output: 2025-07-08 14:00:00 <class 'datetime.datetime'>
```

59. Calcolare la differenza tra due date/ore – Sottrae due oggetti `datetime` per ottenere un oggetto `timedelta` che rappresenta la differenza (giorni, secondi, etc.).

```
from datetime import datetime

inizio = datetime(2025, 7, 1, 9, 30)
fine = datetime(2025, 7, 8, 17, 45)
differenza = fine - inizio
print("Differenza:", differenza)
print("Giorni:", differenza.days, "Secondi residui:", differenza.seconds)
# (Nota: differenza.seconds dà i secondi oltre i giorni pieni)
```

60. Ritardare l'esecuzione di un codice (sleep o Timer) – Utilizza `time.sleep` per attendere qualche secondo prima di procedere, oppure `threading.Timer` per eseguire una funzione dopo un ritardo senza bloccare il thread principale.

```
import time, threading

# Esempio 1: semplice pausa con sleep
print("Attesa di 3 secondi...")
time.sleep(3)
print("3 secondi trascorsi.")

# Esempio 2: usare Timer per eseguire asincronamente
def saluta():
    print("Ciao dopo 5 secondi (esecuzione differita)!")
timer = threading.Timer(5.0, saluta)
timer.start()
print("Timer avviato, attenderò 5 secondi senza bloccare il programma.")
```

61. Eseguire un'azione in un momento specifico (es. alle 10:00) – Calcola il tempo da aspettare fino a un certo orario e poi esegue un'azione.

```
import time, datetime

def esegui_all_ora(target_ora, target_min):
    now = datetime.datetime.now()
    target = now.replace(hour=target_ora, minute=target_min, second=0,
microsecond=0)
    if target < now:
        target += datetime.timedelta(days=1)
# programma per domani se l'ora è già passata
    attesa = (target - now).total_seconds()
    print(f"Attesa di {attesa:.0f} secondi fino alle {target_ora:02d}:
{target_min:02d}")
    time.sleep(attesa)
    print("Ora programmata raggiunta, eseguo l'azione...")

esegui_all_ora(10, 0)  # Esegue l'azione alle 10:00 di oggi (o domani se già
passato)
```

62. Gestire fusi orari e conversioni (zoneinfo/pytz) – Converte un orario naive in timezone specifica usando zoneinfo (in Python 3.9+) o pytz.

```
from datetime import datetime
from zoneinfo import ZoneInfo  # Python 3.9+

ora_utc = datetime.utcnow().replace(tzinfo=ZoneInfo("UTC"))
ora_italia = ora_utc.astimezone(ZoneInfo("Europe/Rome"))
print("Ora UTC:", ora_utc.strftime("%Y-%m-%d %H:%M:%S %Z"))
print("Ora Italia:", ora_italia.strftime("%Y-%m-%d %H:%M:%S %Z"))
```

Generatori di Template

63. Generare un template di docstring per una funzione – Data una funzione, costruisce automaticamente una stringa che rappresenta una docstring base con nomi dei parametri e spiega brevemente.

```
def crea_docstring(func):
    nome = func.__name__
    params = func.__code__.co_varnames[:func.__code__.co_argcount]
    doc = f'"""{nome}:\n'
    for p in params:
        doc += f":param {p}: descrizione di {p}\n"
    doc += ':return: descrizione del valore di ritorno\n"""'
    return doc

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

```
print(crea_docstring(funzione_esempio))
# Output:
# """funzione_esempio:
# :param x: descrizione di x
# :param y: descrizione di y
# :return: descrizione del valore di ritorno
# """
```

64. Generare la struttura base di una classe Python – Crea dinamicamente un file con il template di una classe, includendo il costruttore `__init__` con attributi dati.

```
def genera_template_classe(nome_classe, attributi):
    linee = [f"class {nome_classe}:"]
    linee.append("    def __init__(self, " + ", ".join(attributi) + "):")
    for attr in attributi:
        linee.append(f"        self.{attr} = {attr}")
    linee.append("")  # linea vuota a fine file
    with open(f"{nome_classe.lower()}.py", "w") as f:
        f.write("\n".join(linee))
    print(f"Creato file {nome_classe.lower()}.py con la classe
{nome_classe}.")

genera_template_classe("Persona", ["nome", "eta", "citta"])
# Questo creerà un file persona.py con la definizione base della classe
Persona.
```

65. Creare un file README template per un nuovo progetto – Genera un file README.md di base con titolo, descrizione e sezione per requisiti e utilizzo.

```
def crea_readme(nome_progetto, descrizione):
    contenuto = f"# {nome_progetto}\n\n"
    contenuto += f"{descrizione}\n\n"
    contenuto += "## Installazione\n\n"
    contenuto += "Descrivi come installare le dipendenze.\n\n"
    contenuto += "## Utilizzo\n\n"
    contenuto += "Esempi di come eseguire il programma.\n"
    with open("README.md", "w") as f:
        f.write(contenuto)
    print("Creato README.md di base.")

crea_readme("ProgettoDemo", "Questo progetto è un esempio di utilizzo del
toolbox.")
```

66. Generare stub di test unitari per funzioni esistenti – Crea automaticamente un file di test unitari vuoti per ciascuna funzione in un modulo dato, pronto per essere compilato.

```
import inspect, importlib

def genera_test_stub(nome_modulo):
    mod = importlib.import_module(nome_modulo)
    funzioni = [fn for fn, obj in inspect.getmembers(mod,
inspect.isfunction)]
    nome_file = f"test_{nome_modulo}.py"
    with open(nome_file, "w") as f:
        f.write("import pytest\n")
        f.write(f"import {nome_modulo}\n\n")
        for fn in funzioni:
            f.write(f"def test_{fn}():\n")
            f.write(f"    # TODO: implement test for {fn}\n")
            f.write(f"    assert False  # rimpiazza con test vero\n\n")
    print(f"Creato file di test: {nome_file}")

# Esempio d'uso: genera_test_stub("mio_modulo")
```

67. Creare automaticamente file `__init__.py` in una nuova cartella di pacchetto – Quando si crea un nuovo package (cartella) Python, questo snippet aiuta a creare un file `__init__.py` vuoto al suo interno.

```
import os

def crea_package(path_cartella):
    os.makedirs(path_cartella, exist_ok=True)
    init_path = os.path.join(path_cartella, "__init__.py")
    open(init_path, "a").close()  # crea il file __init__.py vuoto
    print(f"Creato package '{path_cartella}' con __init__.py")

crea_package("nuovo_pacchetto/utilità")
```

68. Usare Jinja2 per generare codice o documentazione da template – Utilizza il motore di templating Jinja2 per popolare un modello con variabili (richiede `pip install jinja2`).

```
from jinja2 import Template

template_str = """\
Ciao {{ nome }},
Grazie per aver utilizzato {{ prodotto }}!
"""
dati = {"nome": "Maria", "prodotto": "IlSoftware"}
template = Template(template_str)
output = template.render(dati)
print(output)
# Output:
```

```
# Ciao Maria,
# Grazie per aver utilizzato IlSoftware!
```

69. Generare uno script Python con logging preconfigurato – Crea un file Python che contiene già configurazione di base del logging e uno scheletro di main.

```
def genera_script_logging(nome_script):
    contenuto = """\
import logging

logging.basicConfig(level=logging.INFO, format="%(asctime)s [%(levelname)s] %
(message)s")

def main():
    logging.info("Script avviato.")
    # TODO: logica principale

if __name__ == "__main__":
    main()
"""
    with open(nome_script, "w") as f:
        f.write(contenuto)
    print(f"Creato script {nome_script} con logging configurato.")

genera_script_logging("script_base.py")
```

Verificatori di Performance

70. Misurare il tempo di esecuzione con timeit – Usa il modulo `timeit` per eseguire spezzoni di codice molte volte e misurarne il tempo medio di esecuzione con precisione.

```
import timeit

setup = "import math"
codice = "math.sqrt(12345)"
tempo = timeit.timeit(codice, setup=setup, number=1000000)
print(f"Esecuzione di sqrt(12345) 1,000,000 volte: {tempo:.4f} secondi")
```

71. Profilare uno script con cProfile – Esegue il profiler standard `cProfile` su uno script o una funzione per ottenere statistiche sui tempi di esecuzione delle varie parti.

```
import cProfile, pstats, io

def lavoro():
    somma = 0
    for i in range(1000000):
```

```
        somma += i
    return somma

pr = cProfile.Profile()
pr.enable()
lavoro()
pr.disable()
s = io.StringIO()
ps = pstats.Stats(pr, stream=s).sort_stats("cumtime")
ps.print_stats(10)  # stampa le 10 funzioni più lente
print(s.getvalue())
```

72. Utilizzare tracemalloc per individuare memory leak – Usa ancora `tracemalloc` (vedi snippet 26) ma concentrandosi sulle statistiche per scoprire dove viene allocata più memoria, per individuare possibili memory leak.

```
import tracemalloc

tracemalloc.start()
# Codice sospetto di consumare memoria:
lista = []
for i in range(100000):
    lista.append(str(i) * 1000)  # creando molte stringhe (esempio)
snapshot = tracemalloc.take_snapshot()
top_stats = snapshot.statistics('traceback')[:3]  # top 3 allocazioni
for stat in top_stats:
    print(stat)
tracemalloc.stop()
```

73. Confrontare le prestazioni di due approcci (benchmark semplice) – Misura il tempo di due diverse funzioni che risolvono lo stesso problema per stabilire quale sia più efficiente.

```
import time

def metodo1(n):
    return [x*2 for x in range(n)]

def metodo2(n):
    res = []
    for x in range(n):
        res.append(x*2)
    return res

n = 1000000
t0 = time.time()
metodo1(n)
t1 = time.time()
metodo2(n)
```

```
t2 = time.time()

print(f"metodo1: {t1 - t0:.4f} s, metodo2: {t2 - t1:.4f} s")
```

74. Misurare l'utilizzo di CPU e memoria di un processo (psutil) – Usa la libreria `psutil` per ottenere informazioni sull'uso di CPU e memoria corrente (richiede `pip install psutil`).

```
import psutil, time

# Informazioni sull'uso CPU e memoria del processo corrente
processo = psutil.Process()
cpu_perc = processo.cpu_percent(interval=1.0)  # percentuale CPU in un
intervallo di 1s
mem_info = processo.memory_info()
mem_mb = mem_info.rss / (1024 * 1024)
print(f"Utilizzo CPU: {cpu_perc}%")
print(f"Memoria usata: {mem_mb:.2f} MB")
```

75. Usare lru_cache per migliorare performance di funzioni ripetitive – Applica il decoratore di caching `functools.lru_cache` a una funzione pesante per evitare ricalcoli su input già visti.

```
import functools
import time

@functools.lru_cache(maxsize=None)
def f_complesso(n):
    time.sleep(1)  # simulazione di calcolo pesante
    return n * n

# Prima chiamata con 10 (lenta, calcola davvero)
print(f_complesso(10))   # impiega ~1s
# Seconda chiamata con 10 (istantanea, usa cache)
print(f_complesso(10))   # impiega ~0s grazie alla cache
```

Logging e Notifiche

76. Configurare logging di base con timestamp e livello – Inizializza il logging per stampare messaggi con data/ora e livello di severità, utile per sostituire i print nel codice di produzione.

```
import logging

logging.basicConfig(level=logging.INFO,
                    format="%(asctime)s [%(levelname)s] %(message)s",
                    datefmt="%Y-%m-%d %H:%M:%S")
logging.info("Applicazione avviata")
```

```
logging.warning("Questo è un avviso")
logging.error("Si è verificato un errore")
```

77. Salvare i log sia su file che su console – Configura due handler di logging: uno scrive su file e l'altro sulla console, così da mantenere traccia persistente dei log.

```
import logging

logger = logging.getLogger("app")
logger.setLevel(logging.DEBUG)
# Handler per file
fh = logging.FileHandler("app.log")
fh.setLevel(logging.DEBUG)
# Handler per console
ch = logging.StreamHandler()
ch.setLevel(logging.INFO)
# Formato comune
formatter = logging.Formatter("%(asctime)s [%(levelname)s] %(message)s")
fh.setFormatter(formatter)
ch.setFormatter(formatter)
logger.addHandler(fh)
logger.addHandler(ch)

logger.debug("Debug (solo su file)")
logger.info("Info (su file e console)")
logger.error("Errore (su file e console)")
```

78. Usare la libreria loguru per un logging semplificato – Loguru offre un'interfaccia semplificata per logging avanzato, con output colorato e rotazione file (richiede `pip install loguru`).

```
from loguru import logger

logger.add("debug.log", level="DEBUG", rotation="1 MB")
logger.debug("Messaggio di debug (comparirà su debug.log)")
logger.info("Messaggio informativo")
logger.error("Messaggio di errore")
```

79. Inviare una notifica desktop al termine di uno script – Utilizza la libreria `plyer` per mostrare una notifica di sistema (cross-platform) quando uno script finisce (richiede `pip install plyer`).

```
from plyer import notification

# ... codice lungo ...
notification.notify(
    title='Script completato',
    message='L\'elaborazione dei dati è terminata con successo.',
```

```
    app_name='MioScript'
)
```

80. Inviare una email di avviso in caso di errore critico – Se un'eccezione non gestita si verifica, manda automaticamente un'email di allerta agli sviluppatori (utilizzando smtplib).

```
import smtplib, traceback
from email.message import EmailMessage

def invia_mail_alert(oggetto, corpo):
    msg = EmailMessage()
    msg["Subject"] = oggetto
    msg["From"] = "monitor@azienda.com"
    msg["To"] = "devteam@azienda.com"
    msg.set_content(corpo)
    # Invia tramite server SMTP interno (esempio)
    with smtplib.SMTP("smtp.azienda.com") as s:
        s.send_message(msg)

try:
    # codice critico...
    raise RuntimeError("Errore critico di esempio")
except Exception as e:
    errore = traceback.format_exc()
    invia_mail_alert("Allarme: Errore Critico nello Script", errore)
```

81. Inviare un messaggio Slack/Telegram tramite webhook – Usa una richiesta HTTP POST verso un webhook Slack (o Telegram) per inviare notifiche da script (richiede `pip install requests`).

```
import requests
import json

webhook_url = "https://hooks.slack.com/services/T00000000/B00000000/
XXXXXXXXXXXXXXXXXXXXXXXX"
messaggio = {"text": "Script completato con successo :tada:"}
try:
    resp = requests.post(webhook_url, data=json.dumps(messaggio),
                         headers={'Content-Type': 'application/json'})
    if resp.status_code == 200:
        print("Notifica Slack inviata.")
    else:
        print("Errore nell'invio Slack:", resp.status_code)
except Exception as e:
    print("Eccezione durante invio notifica:", e)
```

82. Riprodurre un suono di avviso (beep) a fine elaborazione – Emette un suono alla fine di un processo lungo per avvisare l'utente (diverso per OS; su Windows usa winsound, su Linux/Mac si può

usare un beep ASCII).

```
import platform
import sys

def beep():
    if platform.system() == "Windows":
        import winsound
        winsound.MessageBeep(winsound.MB_ICONEXCLAMATION)
    else:
        # Su Linux/Mac potrebbe essere necessario avere il campanello attivo
nel terminale
        sys.stdout.write('\a')
        sys.stdout.flush()

# ... operazione lunga ...
beep()
print("Operazione completata, segnale acustico emesso.")
```

Concorrenza e Parallelismo

83. Avviare più thread per eseguire funzioni in parallelo – Crea e avvia manualmente thread per eseguire funzioni contemporaneamente (ad esempio, calcolare in parallelo).

```
import threading

def lavora(nome):
    print(f"[{nome}] Inizio lavoro")
    # ... operazioni lunghe ...
    print(f"[{nome}] Fine lavoro")

# Avvia 3 thread paralleli
threads = []
for i in range(3):
    t = threading.Thread(target=lavora, args=(f"Thread-{i}",))
    t.start()
    threads.append(t)
# Attendi che tutti i thread finiscano
for t in threads:
    t.join()
print("Tutti i thread completati.")
```

84. Utilizzare ThreadPoolExecutor per operazioni I/O-bound in parallelo – Usa `concurrent.futures.ThreadPoolExecutor` per gestire thread pool facilmente, ad esempio per scaricare più pagine web in parallelo (I/O-bound).

```python
from concurrent.futures import ThreadPoolExecutor
import requests

urls = ["http://example.com", "http://python.org", "http://github.com"]

def fetch(url):
    resp = requests.get(url)
    return url, resp.status_code

with ThreadPoolExecutor(max_workers=5) as executor:
    results = list(executor.map(fetch, urls))
for url, status in results:
    print(f"{url} -> {status}")
```

85. Eseguire calcoli in parallelo con ProcessPoolExecutor – Usa process pool (multiprocessing) per eseguire funzioni CPU-bound in parallelo su più core.

```python
from concurrent.futures import ProcessPoolExecutor
import math

def conta_primi(n):
    # Conta numeri primi fino a n (algoritmo semplice)
    def is_prime(x):
        return x > 1 and all(x % i for i in range(2, int(math.sqrt(x)) + 1))
    return sum(1 for i in range(2, n) if is_prime(i))

numeri = [100000, 200000, 300000]
with ProcessPoolExecutor() as executor:
    risultati = list(executor.map(conta_primi, numeri))
print("Numeri primi trovati:", risultati)
```

86. Comunicare tra thread tramite una coda – Utilizza `queue.Queue` per scambiare dati tra thread in modo thread-safe (ad esempio un thread produttore e un thread consumatore).

```python
import threading, queue, time

q = queue.Queue()

def produttore():
    for i in range(5):
        item = f"elemento-{i}"
        q.put(item)
        print("Prodotto", item)
        time.sleep(1)
    q.put(None)  # segnale di terminazione

def consumatore():
```

```
    while True:
        item = q.get()
        if item is None:  # condizione di uscita
            break
        print("Consumata", item)
        q.task_done()

t1 = threading.Thread(target=produttore)
t2 = threading.Thread(target=consumatore)
t1.start(); t2.start()
t1.join(); t2.join()
```

87. Esempio di uso di asyncio per operazioni non bloccanti – Definisce funzioni asincrone e le esegue con asyncio per gestire operazioni concorrenti (es. attese) nel singolo thread.

```
import asyncio

async def lavoro(index):
    print(f"Task {index} inizio")
    await asyncio.sleep(1)  # simula operazione asincrona (es. I/O)
    print(f"Task {index} fine")
    return index * 2

async def main():
    tasks = [asyncio.create_task(lavoro(i)) for i in range(3)]
    risultati = await asyncio.gather(*tasks)
    print("Risultati finali:", risultati)

asyncio.run(main())
```

88. Scaricare più pagine web in parallelo con asyncio e aiohttp – Combina asyncio con la libreria `aiohttp` per effettuare richieste HTTP concorrenti (richiede `pip install aiohttp`).

```
import asyncio
import aiohttp

urls = ["http://httpbin.org/delay/2", "http://httpbin.org/delay/3", "http://
httpbin.org/delay/1"]

async def fetch(session, url):
    async with session.get(url) as resp:
        data = await resp.text()
        print(f"Scaricato {url} (status {resp.status})")
        return data

async def main():
    async with aiohttp.ClientSession() as session:
        tasks = [asyncio.create_task(fetch(session, url)) for url in urls]
```

```
        await asyncio.gather(*tasks)

asyncio.run(main())
```

89. Usare multiprocessing per elaborare file multipli in parallelo – Crea più processi (multiprocessing) per elaborare diversi file contemporaneamente, aumentando throughput su CPU multi-core.

```
import multiprocessing, time

def elabora_file(file):
    print(f"[Process {multiprocessing.current_process().name}] Elaboro
{file}")
    time.sleep(1)  # simulazione elaborazione
    return file

files = [f"file_{i}.dat" for i in range(4)]
with multiprocessing.Pool(processes=4) as pool:
    risultati = pool.map(elabora_file, files)
print("Elaborazione parallela completata. Risultati:", risultati)
```

90. Limitare il numero di thread attivi con un semaforo – Utilizza `threading.Semaphore` per non superare un certo numero di thread concorrenti (utile se lanci thread in un loop).

```
import threading, time

max_thread = 3
semaforo = threading.Semaphore(max_thread)

def task(nome):
    with semaforo:  # permette solo max_thread thread alla volta
        print(f"{nome} avviato")
        time.sleep(2)
        print(f"{nome} terminato")

threads = []
for i in range(10):
    t = threading.Thread(target=task, args=(f"Task-{i}",))
    t.start()
    threads.append(t)
for t in threads:
    t.join()
print("Tutti i task completati.")
```

Altri Strumenti Utili

91. Generare una password casuale sicura con secrets – Usa il modulo `secrets` per creare una password alfanumerica casuale crittograficamente sicura.

```
import secrets
import string

lunghezza = 12
alfabeto = string.ascii_letters + string.digits + string.punctuation
password = "".join(secrets.choice(alfabeto) for _ in range(lunghezza))
print("Password generata:", password)
```

92. Avviare un semplice server HTTP locale – Usa il modulo built-in `http.server` per avviare un server web locale che serve la directory corrente (utile per test veloci).

```
import http.server
import socketserver

PORT = 8000
Handler = http.server.SimpleHTTPRequestHandler
httpd = socketserver.TCPServer(("", PORT), Handler)
print(f"Server HTTP in esecuzione su porta {PORT}")
httpd.serve_forever()
```

93. Copiare e incollare testo dalla clipboard (pyperclip) – Utilizza la libreria `pyperclip` per accedere agli appunti di sistema e copiare/incollare testo (richiede `pip install pyperclip`).

```
import pyperclip

# Copia testo nella clipboard
pyperclip.copy("Testo copiato negli appunti")
# Incolla testo dalla clipboard
incollato = pyperclip.paste()
print("Dagli appunti ho ottenuto:", incollato)
```

94. Parsare argomenti da linea di comando con argparse – Utilizza il modulo `argparse` per definire e leggere parametri passati allo script da linea di comando.

```
import argparse

parser =
argparse.ArgumentParser(description="Script di esempio che somma due
numeri.")
parser.add_argument("x", type=int, help="primo numero")
```

```
parser.add_argument("y", type=int, help="secondo numero")
parser.add_argument("-v", "--verbose", action="store_true", help="mostra
dettagli")
args = parser.parse_args(["3", "5", "-v"])  # simulazione di parametri;
rimuovere lista per usare sys.argv
risultato = args.x + args.y
if args.verbose:
    print(f"La somma di {args.x} e {args.y} è {risultato}")
else:
    print(risultato)
```

95. Stampare testo colorato in console (colorama) – Usa la libreria `colorama` per aggiungere colori al testo nel terminale (richiede `pip install colorama`).

```
from colorama import Fore, Back, Style, init
init(autoreset=True)  # Inizializza colorama (solo su Windows è necessario)

print(Fore.RED + "Testo in rosso")
print(Back.YELLOW + "Sfondo giallo" + Style.RESET_ALL)
print(Fore.GREEN + "Testo verde" + Style.DIM + " in grassetto attenuato")
```

96. Usare dataclasses per strutturare dati in modo semplice – Crea una classe di dati usando il decoratore `@dataclass` per evitare di scrivere boilerplate (Python 3.7+).

```
from dataclasses import dataclass

@dataclass
class Punto:
    x: int
    y: int

p1 = Punto(3, 5)
p2 = Punto(3, 5)
print(p1)         # Output: Punto(x=3, y=5) (rappresentazione intuitiva)
print(p1 == p2)   # True (metodo __eq__ generato automaticamente)
```

97. Serializzare e deserializzare oggetti Python con pickle – Usa il modulo `pickle` per salvare un oggetto Python su file binario e ricaricarlo (attenzione: il pickle non è sicuro per dati non fidati).

```
import pickle

# Dato un oggetto qualsiasi, ad esempio un dizionario:
data = {"nome": "Ada", "età": 32, "lingue": ["Python", "C++"]}

# Serializzazione (dump)
with open("data.pkl", "wb") as f:
```

```
    pickle.dump(data, f)
# Deserializzazione (load)
with open("data.pkl", "rb") as f:
    data_caricato = pickle.load(f)
print("Oggetto caricato:", data_caricato)
```

98. Creare ed utilizzare un file temporaneo (tempfile) – Usa il modulo `tempfile` per creare file temporanei su disco (che verranno automaticamente rimossi alla chiusura) utili per scrivere dati provvisori.

```
import tempfile

with tempfile.NamedTemporaryFile(mode="w+", delete=True) as tmp:
    tmp.write("Dati temporanei\n")
    tmp.seek(0)
    print("Contenuto temporaneo:", tmp.read().strip())
    print("Percorso file temporaneo:", tmp.name)
# Al termine del blocco with, il file temporaneo viene eliminato
automaticamente
```

99. Valutare un'espressione Python da stringa (eval/ast.literal_eval) – Usa `ast.literal_eval` per valutare in modo sicuro un'espressione Python rappresentata come stringa (ad es. convertire la stringa "[1, 2, 3]" nell'oggetto lista [1,2,3]).

```
import ast

expr = "[1, 2, 3]"
lista = ast.literal_eval(expr)
print(lista, type(lista))  # [1, 2, 3] <class 'list'>

# Attenzione: eval() normale esegue codice arbitrario ed è pericoloso su
input non fidati.
safe_expr = "{'a': 1, 'b': 2}"
diz = ast.literal_eval(safe_expr)
print(diz, type(diz))  # {'a': 1, 'b': 2} <class 'dict'>
```

100. Confrontare due file di testo e mostrare le differenze (difflib) – Usa `difflib` per effettuare un diff riga per riga di due file di testo e stampare le differenze in stile unificato.

```
import difflib

file1 = "file_vecchio.txt"
file2 = "file_nuovo.txt"
with open(file1, "r") as f1, open(file2, "r") as f2:
    testo1 = f1.readlines()
    testo2 = f2.readlines()
```

```
diff = difflib.unified_diff(testo1, testo2, fromfile=file1, tofile=file2)
for linea in diff:
    print(linea, end="")
```
