Appunti di Programmazione in Python

Guido Pacciani

12 giugno 2025

Indice

Capitolo 1

Variabili e operazioni di base

Questo capitolo introduce alcuni dei concetti fondamentali della programmazione Python: le variabili e le operazioni matematiche elementari. Si impara come gestire l'input dell'utente, convertire i tipi di dati e utilizzare gli operatori aritmetici.

1.1 Esercizio 1 – Le quattro operazioni matematiche fondamentali

Scrivi un codice che stampi il risultato delle quattro operazioni matematiche elementari tra due numeri interi forniti dall'utente.

Analisi del problema

Per risolvere questo esercizio è necessario:

1. Acquisire due numeri dall'utente tramite la funzione `input()`
2. Convertire le stringhe in numeri interi usando `int()`
3. Eseguire le quattro operazioni: somma, sottrazione, moltiplicazione e divisione
4. Stampare i risultati formattati

Implementazione

```
# Il programma richiede all'utente l'inserimento di due numeri.
# La funzione input restituisce una stringa.
a_str = input("Inserisci il primo numero: ")
b_str = input("Inserisci il secondo numero: ")

# Conversione delle stringhe in numeri interi mediante "casting"
a_int = int(a_str)
b_int = int(b_str)

# Calcolo e stampa dei risultati delle quattro operazioni aritmetiche di base
print("La somma dei numeri inseriti dà come risultato:", a_int + b_int)
print("La differenza dei numeri inseriti dà come risultato:", a_int - b_int)
print("Il prodotto dei numeri inseriti dà come risultato:", a_int * b_int)
print("Il quoziente dei numeri inseriti dà come risultato:", a_int / b_int)
```

Output del programma

```
Inserisci il primo numero: 10
Inserisci il secondo numero: 20
La somma dei numeri inseriti dà come risultato: 30
La differenza dei numeri inseriti dà come risultato: -10
Il prodotto dei numeri inseriti dà come risultato: 200
Il quoziente dei numeri inseriti dà come risultato: 0.5
```

Note utili

- La funzione `input()` restituisce sempre una stringa
- Per poter effettuare un'operazione matematica su un dato fornito dall'utente, è necessario convertire la stringa in un numero utilizzando `int()` per i numeri interi o `float()` per i numeri razionali.
- La divisione `/` restituisce sempre un numero in virgola mobile (float)
- È possibile concatenare stringhe e numeri usando la virgola nella `print()`

1.2 Esercizio 2 – Il quoziente e i tipi di divisione

Partendo da due numeri interi forniti dall'utente, esplora le diverse modalità con cui Python gestisce la divisione. In particolare, sperimenta l'utilizzo degli operatori `/` e `//` per osservare la differenza tra divisione in virgola mobile e divisione intera. Utilizza inoltre la funzione `round()` per confrontare l'effetto dell'arrotondamento rispetto al semplice troncamento. Stampa i risultati a schermo e commenta brevemente cosa osservi.

Implementazione

```
# Utilizziamo i numeri dell'esericizio precedente

# 1) Divisione     classica     → sempre float
res_float = a_int / b_int
print(f"Divisione: {a_int} / {b_int} = {res_float:.2f}")

# 2) Divisione intera (floor division)
res_floor = a_int // b_int
print(f"Divisione intera (//): {a_int} // {b_int} = {res_floor}")

# 3) Risultato arrotondato con round()
#    Applico round sul risultato float, con 0 decimali
res_round = round(res_float, 0)
print(f"Risultato arrotondato (round): round({res_float:.2f}, 0) = {
    res_round:.0f}")

# 4) Resto della divisione (modulo)
res_mod = a_int % b_int
print(f"Resto della divisione: {a_int} % {b_int} = {res_mod}")
```

Output del programma

Dati in input i numeri 7 e 3:

```
Divisione: 7 / 3 = 2.33
Divisione intera (//): 7 // 3 = 2
Risultato arrotondato (round): round(2.33, 0) = 2
Resto della divisione: 7 % 3 = 1
```

Note utili

- La divisione normale `/` produce sempre un float, anche quando il risultato è un numero intero
- La divisione intera `//` esegue un "floor" (arrotondamento verso il basso)
- La funzione `round()` arrotonda al numero intero più vicino
- L'operatore modulo `%` restituisce il resto della divisione intera

1.3 Esercizio 3 – Modulo e potenza tra due numeri

Scrivi un programma che richieda all'utente due numeri: uno in virgola mobile e uno intero. Utilizza questi valori per calcolare e visualizzare:

- la potenza del primo numero elevato al secondo
- il resto della divisione tra i due numeri (operatore modulo `%`)

Formatta l'output utilizzando le `f-string` per rendere la stampa chiara e leggibile.

Analisi del problema

Per risolvere questo esercizio dobbiamo:

1. Acquisire un numero in virgola mobile e un numero intero dall'utente
2. Convertire correttamente i dati utilizzando `float()` e `int()`
3. Calcolare la potenza e il modulo
4. Stampare i risultati usando la formattazione `f-string`

Implementazione

```
# Richiesta di un numero in virgola mobile e un numero intero
c_float = float(input("Inserisci un numero razionale: "))
d_int = int(input("Inserisci un numero intero: "))

```

```
# Calcolo della potenza e del resto della divisione all'interno della formattazione f-string
print(f"{c_float} ** {d_int} = {c_float ** d_int}")  # Potenza
print(f"{c_float} % {d_int} = {c_float % d_int}")    # Modulo (resto)
```

Output del programma

```
Inserisci il primo numero: 2
Inserisci il secondo numero: 3
2.0 ** 3 = 8.0
2.0 % 3 = 2.0
```

Note utili

- L'operatore `**` in Python calcola la potenza
- L'operatore `%` restituisce il resto della divisione tra due numeri
- Le `f-string` permettono di formattare facilmente l'output unendo stringhe e risultati numerici

1.4 Esercizio 4 – Scambio di variabili ("Swapping")

Scrivi un programma che acquisisca due valori da tastiera e ne mostri lo scambio:

- prima utilizzando una variabile temporanea,
- poi sfruttando l'assegnazione multipla di Python.

Verifica il corretto scambio stampando i valori prima e dopo ogni operazione.

Analisi del problema

Per risolvere l'esercizio dobbiamo:

1. Acquisire due valori dall'utente con `input()`

2. Scambiarli una prima volta usando una variabile temporanea
3. Scambiarli di nuovo usando l'assegnazione multipla
4. Stampare i valori prima e dopo ogni scambio

Implementazione

```
# Acquisizione di due valori da tastiera
A = input("Inserisci il primo valore: ")
B = input("Inserisci il secondo valore: ")

# Stampa dei valori originali
print(f"A = {A}, B = {B}")

# Scambio dei valori utilizzando una variabile temporanea C
C = B
B = A
A = C
print(f"A = {A}, B = {B}")

# Scambio dei valori con assegnazione multipla (senza variabile
    temporanea)
A, B = B, A
print(f"A = {A}, B = {B}")
```

Output del programma

```
Inserisci il primo valore: 10
Inserisci il secondo valore: 5
A = 10, B = 5
A = 5, B = 10
A = 10, B = 5
```

Note utili

- Lo scambio con variabile temporanea è universale, usabile anche in altri linguaggi
- L'assegnazione multipla è una sintassi specifica e molto compatta di Python

1.5 Esercizio 5 – Casting e formattazione dell'output

Scrivi un programma che richieda all'utente alcuni dati anagrafici e fisici (nome, anno e luogo di nascita, altezza e peso), calcoli l'età e stampi un riepilogo completo in formato leggibile.
Utilizza le `f-string` per formattare l'output e arrotonda altezza e peso a due cifre decimali.

Implementazione

```
# Acquisizione dei dati personali da tastiera
Nome = input("Nome: ")
Anno_nascita = int(input("Anno di nascita: "))
Eta = 2025 - Anno_nascita  # Calcolo dell'età
Luogo_nascita = input("Luogo di nascita: ")
Altezza = float(input("Altezza (m): "))
Peso = float(input("Peso (kg): "))

# Stampa del riepilogo formattato con arrotondamento a due decimali
print(f"{Nome}, {Eta} anni, nato a {Luogo_nascita} nel " +
      f"{Anno_nascita}, altezza: {Altezza:.2f} m, peso: {Peso:.2f} "
      + "kg")
```

Output del programma

```
Nome: Guido
Anno di nascita: 1997
Luogo di nascita: Parma
Altezza (m): 1.85
Peso (kg): 80
Guido, 28 anni, nato a Parma nel 1997, altezza: 1.85 m, peso: 80.00
   kg
```

Note utili

- L'età è calcolata come differenza tra l'anno corrente e l'anno di nascita

- Il formato `:.2f` in una `f-string` arrotonda il numero a due cifre decimali.
- Il formato `:.0f` in una `f-string` arrotonda il numero al valore intero più vicino, eliminando completamente la parte decimale.
- La funzione `input()` restituisce sempre una stringa: è necessario convertire altezza e peso

Riepilogo comandi principali

Questa tabella riassume i comandi e costrutti fondamentali introdotti nel capitolo.

Comando	Descrizione	Note utili
`input("")`	Legge un dato da tastiera come stringa	Serve conversione con `int()`, `float()` per usarlo come numero
`int()`	Converte una stringa (o float) in numero intero	Lancia errore se la stringa non è convertibile
`float()`	Converte una stringa (o int) in numero decimale	Utile per rappresentare misure, pesi, ecc.
`print("")`	Stampa un messaggio sullo schermo	Supporta stringhe, numeri, variabili, f-string
`+ - * / // % **`	Operatori matematici	Somma, differenza, prodotto, divisione, divisione intera, modulo, potenza
`round(x, n)`	Arrotonda il numero `x` a `n` cifre decimali	`round(3.14159, 2)` → `3.14`
`f""`	Formattazione stringhe	Inserisce variabili in una stringa: `f" altezza: {Altezza:.2f}"`
`a, b = b, a`	Scambia i valori di due variabili	Alternativa compatta all'uso di variabile temporanea

Capitolo 2

Collezione di dati

In questo capitolo affronteremo le principali strutture dati offerte da Python per la raccolta e la gestione di insiemi di informazioni. In particolare, utilizzeremo le `liste`, le `tuple`, i `set` e i `dizionari`, strumenti fondamentali per organizzare dati strutturati in modo efficiente.

Principali strutture dati in Python

Python mette a disposizione diverse strutture dati per organizzare insiemi di valori.

- **Liste** (`list`): collezioni ordinate e modificabili di elementi. Gli elementi sono racchiusi tra parentesi quadre `[]` e possono essere eterogenei (es. numeri, stringhe, altre liste...).
- **Tuple** (`tuple`): collezioni ordinate ma **immutabili**. Gli elementi sono racchiusi tra parentesi tonde `()`. Una volta definita, una tupla non può essere modificata.
- **Set** (`set`): collezioni **non ordinate** di elementi **univoci**. Si definiscono con parentesi graffe o con il costruttore `set()`. Sono ottimi per operazioni di appartenenza, unione, intersezione e differenza tra insiemi.
- **Dizionari** (`dict`): strutture che associano a ogni *chiave* un *valore*. Le coppie chiave-valore sono racchiuse tra parentesi graffe `{}`, con la sintassi `chiave: valore`. Le chiavi devono essere univoche.

Le strutture possono anche essere **annidate**, cioè contenere al loro interno altre strutture. Per esempio: una lista di tuple, un set di dizionari, o un dizionario con liste come valori.

Accesso ai dati

- Per accedere a un elemento di una lista o tupla si usa l'indice numerico: `lista[0]`.
- Per accedere a un elemento di un set non si può usare l'indice (sono non ordinati); si verifica l'appartenenza con `valore in mio_set`.
- Per accedere a un valore in un dizionario si usa la chiave: `diz["Mario"]`.
- In strutture annidate si possono combinare i diversi tipi di accesso: `diz["Mario"][2]` pesca il terzo elemento dalla lista a chiave `"Mario"`, oppure `mia_lista[0] in mio_set` per testare se il primo elemento di una lista è in un set.

2.1 Esercizio – Pagelle scolastiche

Scrivi un programma che rappresenti i record scolastici di più studenti. La struttura dati dovrà essere un dizionario in cui:

- La chiave è il nome dello studente (stringa)
- Il valore è una lista di tuple, una per ciascuna materia
- Ogni tupla contiene: nome della materia (stringa), voto (numero), ore di assenza (numero)

2.1.1 A. Creazione della struttura dati iniziale

Popola il dizionario iniziale utilizzando i dati scolastici di tre studenti.

Breve terminologia: un *dizionario* è una collezione di coppie *chiave:valore* (es. `{"studente": voti}`); la *chiave* è qui una stringa, il *valore* è una *lista* di *tuple* (ogni tupla, es. `("Materia", voto, ore_assenza)`, raccoglie tre elementi).

Implementazione

```
pagelle = {
    "Luca Moretti": [
        ("Matematica", 9, 0),
        ("Italiano", 7, 3),
        ("Inglese", 7.5, 4),
        ("Storia", 7.5, 4),
        ("Geografia", 5, 7)
    ],
    "Chiara Bianchi": [
        ("Matematica", 8, 1),
        ("Italiano", 6, 1),
        ("Inglese", 9.5, 0),
        ("Storia", 8, 2),
        ("Geografia", 8, 1)
    ],
    "Marco De Luca": [
        ("Matematica", 7.5, 2),
        ("Italiano", 6, 2),
        ("Inglese", 4, 3),
        ("Storia", 8.5, 2),
        ("Geografia", 8, 2)
    ]
}
```

2.1.2 B. Aggiunta di un nuovo studente

Aggiungi al dizionario un nuovo studente, chiamato `Elena Ricci`, con voto 10 in tutte le materie e 0 ore di assenza.

Sintassi di inserimento: per aggiungere una nuova coppia chiave–valore si usa `dizionario[chiave] = valore`.

Implementazione

```
pagelle["Elena Ricci"] = [
    ("Matematica", 10, 0),
    ("Italiano", 10, 0),
    ("Inglese", 10, 0),
    ("Storia", 10, 0),
    ("Geografia", 10, 0)
]
```

2.1.3 Aggiunta della materia Fisica

Aggiungi la materia `Fisica` alla pagella di tutti gli studenti con i seguenti voti e assenze:

- Luca Moretti: voto 9.5, assenze 0
- Chiara Bianchi: voto 8, assenze 1
- Marco De Luca: voto 8, assenze 3
- Elena Ricci: voto 10, assenze 0

Implementazione

```
pagelle["Luca Moretti"].append(("Fisica", 9.5, 0))
pagelle["Chiara Bianchi"].append(("Fisica", 8, 1))
pagelle["Marco De Luca"].append(("Fisica", 8, 3))
pagelle["Elena Ricci"].append(("Fisica", 10, 0))
```

2.1.4 Accesso alla materia Matematica per uno studente

Stampa la tupla corrispondente alla materia `Matematica` per lo studente `Luca Moretti`.

Implementazione

```
print(pagelle["Luca Moretti"][0])
```

Output del programma

```
('Matematica', 9, 0)
```

2.1.5 Accesso alla materia Inglese

Stampa la tupla corrispondente alla materia `Inglese` per lo studente `Marco De Luca`.

Implementazione

```
print(pagelle["Marco De Luca"][2])
```

Output del programma

```
('Inglese', 4, 3)
```

2.1.6 Accesso a un voto specifico

Stampa solo il voto di `Chiara Bianchi` nella materia `Geografia`.

Implementazione

```
print(pagelle["Chiara Bianchi"][-2][1])
```

Output del programma

```
8
```

Riepilogo strutture dati

Questa tabella sintetizza i principali costrutti utilizzati nel capitolo precedente.

Concetto	**Descrizione**	**Esempio**
`lista`	Collezione ordinata e modificabile di elementi, racchiusa tra parentesi quadre	`[1, 2, 3]`
`tupla`	Collezione ordinata ma immutabile, racchiusa tra parentesi tonde	`("Matematica", 9, 0)`
`dizionario`	Collezione non ordinata di coppie chiave-valore, racchiusa tra parentesi graffe	`{"Luca": [...]}`
`chiave`	Identificatore univoco all'interno di un dizionario per accedere a un valore	`pagelle["Luca"]`
`append()`	Metodo che aggiunge un elemento in fondo a una lista	`lista.append(10)`
`indice numerico`	Posizione di un elemento all'interno di una lista o tupla, a partire da zero	`lista[0]`
`accesso annidato`	Accesso combinato in strutture composte (es. dizionario di liste di tuple)	`pagelle["Chiara"][2][1]`

Capitolo 3

Istruzioni condizionali

In questo capitolo vengono illustrate le tecniche per realizzare programmi che reagiscono a condizioni diverse, eseguendo percorsi di codice differenti in base ai valori assunti da specifiche variabili. L'istruzione `if` rappresenta lo strumento fondamentale che consente a Python di prendere decisioni.

Le istruzioni condizionali permettono di controllare il flusso di esecuzione di un programma. In Python, la sintassi base è:

```
if condizione:
    # blocco di istruzioni eseguito se la condizione è vera
elif altra_condizione:
    # blocco eseguito se la precedente era falsa e questa vera
else:
    # blocco eseguito se tutte le precedenti condizioni sono false
```

- Il blocco di codice condizionato va rientrato (indentato) rispetto all'istruzione `if`.
- Le condizioni sono espressioni booleane: restituiscono `True` o `False`.
- Gli operatori logici `and`, `or`, `not` permettono di costruire condizioni complesse.

3.1 Esercizio – Weird or Not Weird

Scrivi un programma che legga da input un numero intero `n` e stampi un messaggio secondo le seguenti regole:

- Se `n` è dispari, stampa: `"Weird"`
- Se `n` è pari e compreso tra 2 e 5 (inclusi), stampa: `"Not Weird"`
- Se `n` è pari e compreso tra 6 e 20 (inclusi), stampa: `"Weird"`
- Se `n` è pari e maggiore di 20, stampa: `"Not Weird"`

Vincolo: $1 \leq n \leq 100$

Implementazione: versione estesa

```python
# Chiedo all'utente di inserire un numero
# Siccome l'input viene letto come stringa, lo converto in intero
n = int(input("Scrivi un numero intero (tra 1 e 100): "))

# Verifica che il numero sia nel range richiesto
if 1 <= n <= 100:
    # Se n è dispari
    if n % 2 != 0: # divisione per due con resto diverso da zero
        print("Weird")
    # Se n è pari e tra 2 e 5
    elif 2 <= n <= 5:
        print("Not Weird")
    # Se n è pari e tra 6 e 20
    elif 6 <= n <= 20:
        print("Weird")
    # Se n è pari e maggiore di 20
    else:
        print("Not Weird")
else:
    print("Numero fuori dall'intervallo previsto.")
```

Output del programma

```
Scrivi un numero intero: 8
Weird
```

Note utili

- Il controllo `n % 2 != 0` verifica se il numero è dispari.
- Le istruzioni `elif` permettono di specificare condizioni aggiuntive in alternativa all'`if`.
- Il blocco `else` viene eseguito solo se nessuna condizione precedente è vera.

Implementazione: versione compatta con condizione multipla

Modifica il programma in modo che:

- Il numero venga richiesto finché non è valido
- La verifica finale venga effettuata con un'unica condizione logica

```python
n = int(input("Scrivi un numero intero compreso tra 1 e 100: "))

# Blocco 1: Finché il numero è fuori dal range, continuo a chiedere
while not (1 <= n <= 100):
    print("Numero fuori dall'intervallo previsto. Riprova")
    n = int(input("Scrivi un numero intero compreso tra 1 e 100: "))

# Blocco 2: Il numero è valido! Applico la verifica "Weird / Not Weird"
if n % 2 != 0 or 6 <= n <= 20:
    print("Weird")
else:
    print("Not Weird")
```

Output del programma

```
Scrivi un numero intero compreso tra 1 e 100: 200
Numero fuori dall'intervallo previsto. Riprova
Scrivi un numero intero compreso tra 1 e 100: 130
Numero fuori dall'intervallo previsto. Riprova
Scrivi un numero intero compreso tra 1 e 100: 12
Weird
```

Note utili

- L'uso di `while not (...)` consente di validare l'input in modo elegante.
- L'espressione `n % 2 != 0 or 6 <= n <= 20` consente di unificare le due condizioni che richiedono la stampa di `"Weird"`.
- La struttura compatta rende il codice più sintetico, ma richiede maggiore attenzione nella lettura della logica.

Riepilogo istruzioni condizionali

Questa tabella riassume le principali istruzioni condizionali e di controllo del flusso in Python. Ogni blocco deve essere indentato correttamente per evitare errori di sintassi.

Costrutto	**Descrizione**	**Esempio**
`if condizione:`	Esegue il blocco solo se la condizione è vera.	`if n > 10:` `print("Maggiore di 10")`
`if not condizione:`	Esegue il blocco solo se la condizione è falsa.	`if not (n > 10):` `print("Non è maggiore di 10")`

`elif condizione:`	Verifica una nuova condizione se la precedente era falsa.	`if n < 0:`
		`print("Negativo")`
		`elif n == 0:`
		`print("Zero")`

`else:`	Esegue il blocco se tutte le condizioni precedenti sono false.	`if n < 0:`
		`print("Negativo")`
		`else:`
		`print("Positivo o zero")`

`if ... or ...`	Vero se almeno una delle condizioni è vera.	`if n < 0 or n > 100:` `print("Fuori intervallo")`
`if ... and ...`	Vero solo se entrambe le condizioni sono vere.	`if n >= 1 and n <= 100:` `print("OK")`

`while condizione:`	Ripete il blocco finché la condizione è vera.	`while n != 0:`
		`print(n)`
		`n = n - 1`

`while not condizione:`	Continua finché la condizione è falsa (utile per input validi).	`while not (1 <= n <= 100):`
		`print("Non valido")`
		`n = int(input())`

Indentazione	I blocchi interni devono essere rientrati di 4 spazi. Python non usa parentesi graffe.	`if condizione:`
		`  print("Ok")`
		`print("Fine")`

Capitolo 4

Cicli condizionali

In questo capitolo vengono proposti esercizi pratici volti a consolidare l'applicazione combinata di istruzioni condizionali e strutture iterative in Python. Le esercitazioni prevedono acquisizione di input, utilizzo di cicli, controllo del flusso e manipolazione di strutture dati, con particolare attenzione a cicli `while`, condizioni multiple, accesso a dizionari e analisi di stringhe.

Python offre due strutture principali per l'iterazione:

- `while`: esegue un blocco finché la condizione rimane vera. Utile quando non si conosce a priori il numero di iterazioni.
- `for`: esegue un blocco per ogni elemento di una sequenza (lista, stringa, intervallo numerico ecc.). Spesso usato con `range()`.

Esempio di ciclo `while`:

```
i = 0
while i < 5:
    print(i)
    i += 1
```

Esempio di ciclo `for`:

```
for i in range(5):
    print(i)
```

Nota: l'operatore `i += 1` è una forma abbreviata di `i = i + 1`. In pratica prende il valore corrente di `i`, ci aggiunge 1 e poi lo riassegna a `i`. Per tipi mutabili (es. liste) modifica l'oggetto in-place, mentre per tipi immutabili (es. interi, float) crea un nuovo oggetto e lo assegna a `i`.

4.1 Esercizio 1 – Potenze quadrate

Leggi un numero intero `N` compreso tra 1 e 20 (inclusi). Stampa il quadrato di tutti i numeri interi da 0 a `N-1`.

Implementazione

```
# Chiedo all'utente di inserire un numero compreso tra 1 e 20
N = int(input("Inserisci un numero intero compreso tra 1 e 20: "))

# Inizializzo la variabile i a 0 (sarà il contatore del ciclo)
i = 0

# Verifico se il numero inserito è valido (compreso tra 1 e 20)
if N < 1 or N > 20:
    print("Numero non valido")
else:
    while i < N:
        print(i**2)
        i += 1
```

Output del programma

```
Inserisci un numero intero compreso tra 1 e 20: 5
0
1
4
9
16
```

Note utili

- Il ciclo `while` viene usato quando si vuole controllare manualmente l'indice `i`.
- `i**2` è il quadrato del numero `i`.

4.2 Esercizio 2 – Trova il secondo classificato

Scrivi un programma che riceve in input una serie di punteggi separati da spazi e restituisce il secondo punteggio più alto. I punteggi possono contenere duplicati. Il secondo classificato è colui che ha ottenuto il secondo valore più alto distinto.

Per determinare il secondo massimo in un insieme di punteggi, è utile:

- eliminare i duplicati con la funzione `set()`
- ordinare i valori (se necessario)
- oppure confrontare ciascun elemento con gli altri per determinare quanti sono superiori

Metodo 1: confronto diretto

```
# 1. Chiedo all'utente di inserire 5 punteggi separati da spazi
scores = input("Inserisci i cinque punteggi separati da spazi: ")

# 2. Converto la stringa in una lista di numeri interi
# Uso split() per separare la stringa e map(int, ...) per convertire ogni elemento
scores = list(map(int, scores.split()))

# 3. Elimino eventuali duplicati usando un set
# Questo mi serve per confrontare solo i punteggi distinti
scores = set(scores)

# 4. Cerco il secondo punteggio più alto:
# l'idea è: per ogni punteggio i, conto quanti altri punteggi sono maggiori se esiste un solo punteggio maggiore → i è il secondo classificato
for i in scores:
    larger = 0 # contatore inizializzato a 0
    for j in scores:
        if i < j:
            larger += 1
    if larger == 1:
        break  # i è il secondo punteggio più alto

# 5. Stampo il risultato
print(f"Il punteggio del secondo classificato è {i}")
```

Listing 4.1: Metodo 1 – Confronto tra punteggi

Output del programma

```
Inserisci i cinque punteggi separati da spazi: 3 5 1 2 4
Il punteggio del secondo classificato è 4
```

Osservazioni

- L'algoritmo conta quante volte ciascun punteggio è inferiore ad altri.
- L'interruzione con `break` si verifica quando trova il punteggio con un solo valore più alto.
- L'uso del `set()` è essenziale per evitare ambiguità dovute ai duplicati.

Metodo 2: ordinamento

```python
# 1. Chiedo all'utente di inserire 5 numeri separati da spazi in un'unica riga
scores = input("Inserisci i cinque punteggi separati da spazi: ")

# 2. Uso map(int, ...) per convertire direttamente ogni elemento della stringa in intero
# Uso set(...) per rimuovere eventuali punteggi duplicati, così il confronto sarà più corretto
scores = set(map(int, scores.split()))

# 3. Riconverto il set in lista perché i set non sono ordinabili
# Uso sort() per ordinare i punteggi in ordine crescente (default)
scores = list(scores)
scores.sort()

# 4. Stampo i punteggi ordinati per controllo
print(scores)

# 5. Se ci sono almeno due punteggi distinti, stampo il secondo più alto
# Uso l'indice -2 perché la lista è ordinata in modo crescente
if len(scores) >= 2:
    print(f"Il secondo punteggio più alto è: {scores[-2]}")
```

```
20 else:
21     print("Non ci sono abbastanza punteggi distinti per determinare
   un secondo massimo.")
```

Output del programma

```
1 Inserisci i cinque punteggi separati da spazi: 8 8 7 7 3
2 [3, 7, 8]
3 Il secondo punteggio più alto è: 7
```

Osservazioni

- L'approccio è più semplice e sfrutta l'ordinamento automatico delle liste.
- L'indice `-2` accede al penultimo elemento, cioè al secondo massimo.
- Se i valori sono troppo pochi o uguali, il programma segnala il problema.

4.3 Esercizio 3 – Registro studenti

Scrivi un programma che riceve in input un numero `N` e successivamente `N` righe, ciascuna con il nome di uno studente seguito da tre voti (Matematica, Fisica, Chimica). Salva questi dati in un dizionario con nome come chiave e lista di voti come valore. Infine, leggi un nome e stampa la media dei voti dello studente (con due cifre decimali).

Analisi del problema

Il problema richiede l'uso di un dizionario per associare a ogni studente i suoi voti. È importante:

- saper costruire un ciclo `for` che si ripete `N` volte
- saper usare `split()`, `map()` e `float()` per convertire i voti
- accedere ai dati con `dict[nome]` per calcolare la media

Metodo 1: accesso con variabili separate

```
# 1. Inizializzo un dizionario vuoto per salvare i voti degli studenti
# Uso un dizionario perché mi serve associare a ciascun nome una lista di voti
students = {}

# 2. Chiedo all'utente quanti studenti inserirà
# Salvo il valore in N per poter ripetere l'inserimento N volte
N = int(input("Inserisci il numero di studenti: "))

# 3. Inserisco i dati per ciascuno studente
# Per ogni studente, leggo nome e tre voti separati da spazi
# Converto i voti in float per calcolare la media in seguito
# Salvo il nome come chiave e i voti come lista nel dizionario
for _ in range(N):
    name, math, physics, chemistry = input("Inserisci nome e tre voti: ").split()
    # Aggiungo una nuova voce al dizionario students:
    # uso il nome come chiave e associo ad esso una lista con i tre voti convertiti in float.
    # Questo mi permette di accedere ai voti di ogni studente usando direttamente il suo nome, e di poter calcolare facilmente la media successivamente.
    students[name] = [float(math), float(physics), float(chemistry)]

# 4. Chiedo il nome dello studente di cui voglio calcolare la media
query_name = input("Inserisci il nome dello studente da analizzare: ")

# 5. Verifico se il nome è presente nel dizionario
# Se sì, calcolo e stampo la media con due cifre decimali
# Altrimenti, stampo un messaggio di errore
if query_name in students:
    scores = students[query_name]  # Estraggo i voti dello studente per calcolare la media
    average_score = sum(scores) / len(scores)
    print(f"{average_score:.2f}")
else:
    print(f"Studente {query_name} non trovato.")
```

Output del programma

```
Inserisci il numero di studenti: 3
Paolo 23 25 27
Michele 30 29 28
Gesù 18 19 30
Inserisci il nome dello studente da analizzare: Gesù
22.33
```

Osservazioni

- La struttura è chiara e leggibile, utile per esercizi didattici.
- Il controllo con `if nome in dizionario` previene errori.

Metodo 2: gestione più compatta dei dati

```python
# Inizializzo un dizionario per associare a ciascuno studente la
   lista dei suoi voti
students_marks = {}

# Leggo il numero totale di studenti da inserire
n = int(input())

# Per ciascuno studente:
#   - leggo il nome e i voti da input (tutti in una riga)
#   - prendo il primo elemento come nome
#   - converto i successivi in float per poter calcolare la media
#   - salvo tutto nel dizionario usando il nome come chiave
for _ in range(n):
    record = input().split()
    name = record[0]
    marks = list(map(float, record[1:]))
    students_marks[name] = marks

# Leggo il nome dello studente di cui voglio calcolare la media
query_name = input()

# Recupero la lista dei voti associata a quel nome
marks = students_marks[query_name]

# Calcolo la media dei voti usando sum e len
average = sum(marks) / len(marks)
```

```
26
27 # Stampo la media formattata con due cifre decimali
28 print(f"{average:.2f}")
```

Output del programma

```
1 3
2 Mario 18 19 30
3 Paolo 21 20 29
4 Andrea 18 23 26
5 Andrea
6 22.33
```

Note utili

- Il codice è più compatto e scalabile.
- L'uso di `map()` consente di convertire direttamente più valori in `float`.
- Il metodo presuppone che i nomi degli studenti siano univoci.

4.4 Esercizio 4 – Palindromi

Scrivi un programma che legge ripetutamente parole da input e verifica se ciascuna di esse è un palindromo. Il ciclo termina quando l'utente inserisce la parola `"stop"`. Un palindromo è una parola che si legge allo stesso modo in entrambi i sensi (es. `otto`, `radar`, `anna`).

Analisi del problema

Per verificare se una parola è un palindromo:

- si confronta la parola con la sua versione invertita (`parola[::-1]`)
- si può utilizzare il metodo `lower()` per rendere il confronto case-insensitive

Nota sullo slicing: in Python lo slicing utilizza la sintassi

`sequenza[start:stop:step]`

dove:

- `start`: indice iniziale (inclusivo); se omesso equivale all'inizio;
- `stop`: indice finale (esclusivo); se omesso equivale alla fine;
- `step`: passo tra elementi; se omesso vale `1`; se negativo (es. `-1`) scorre la sequenza all'indietro.

Ad esempio, `parola[::-1]` restituisce la stringa invertita.

Il programma utilizza un ciclo `while` per continuare a ricevere input finché l'utente non digita `"stop"`.

Metodo 1: confronto diretto con stringa invertita

```
# 1. Chiedo all'utente di inserire una parola
parola = input("Inserisci una parola: ")

# 2. Continuo finche la parola non e "stop"
while parola.lower() != "stop":
    # 3. Controllo se la parola e un palindromo (ignorando maiuscole)
    if parola.lower() == parola.lower()[::-1]:
        print(f"{parola} e un palindromo")
    else:
        print(f"{parola} non e un palindromo")

    # 4. Chiedo una nuova parola
    parola = input("Inserisci una parola: ")
```

Output del programma

```
Inserisci una parola: Otto
Otto è un palindromo
Inserisci una parola: Radar
Radar è un palindromo
Inserisci una parola: Stop
```

Note utili

- La funzione `[::-1]` permette di invertire velocemente una stringa.

Metodo 2: confronto lettera per lettera

```

# 1. Chiedo all'utente di inserire una parola
word = input("Inserisci una parola: ")

# 2. Continuo finché la parola non è "stop"
while word.lower() != "stop":

    # 3. Inizio assumendo che la parola sia un palindromo
    # Se trovo un carattere che non corrisponde, cambierò idea
    is_palindrome = True

    # 4. Controllo carattere per carattere confrontando estremi opposti
    for i in range(len(word)):
        j = len(word) - 1 - i
        if word.lower()[i] != word.lower()[j]:
            is_palindrome = False
            break

    # 5. Stampo il risultato del controllo
    if is_palindrome:
        print(f"{word} è un palindromo")
    else:
        print(f"{word} non è un palindromo")

    # 6. Chiedo una nuova parola per ripetere il ciclo
    word = input("Inserisci una parola: ")
```

Output del programma

```
Inserisci una parola: Anna
Anna è un palindromo
Inserisci una parola: Stop
```

Note utili

- Il metodo mostra passo passo la logica dietro al controllo, utile in contesti didattici.
- Il ciclo `for` scorre la parola per confrontare simmetricamente i caratteri.
- La variabile booleana `is_palindrome` permette di salvare lo stato del controllo.

Tabella riassuntiva – Costrutti condizionali e cicli

Costrutto	Descrizione	Esempio
`for variabile in range():`	Ciclo su un intervallo di numeri. Esegue il blocco una volta per ogni valore generato da `range()`	`for i in range(3):` `  print(i)`

`break`	Interrompe immediatamente l'esecuzione del ciclo più interno e prosegue con il codice successivo al ciclo	`for i in range(10):`
		`  if i == 5:`
		`  break`

`continue`	Salta il resto del corpo del ciclo corrente e inizia subito l'iterazione successiva. Utile per ignorare casi specifici senza uscire dal ciclo.	`for i in range(5):`
		`  if i % 2 == 0:`
		`  continue`
		`  print(i)`

`pass`	Placeholder per un blocco vuoto: non esegue nulla ma mantiene la sintassi valida	`if x > 0:` `  pass`

Capitolo 5

Le funzioni

In questo capitolo si applica il concetto di funzione alla risoluzione di problemi pratici, organizzando il codice in blocchi riutilizzabili, leggibili e facilmente testabili.

5.1 Esercizio 1 - Calcolo dell'area della circonferenza

Definisci una funzione che, prendendo in input il raggio di una circonferenza, restituisca la relativa area.

Formula di riferimento: $A = \pi \cdot r^2$, dove r è il raggio fornito in input e π è approssimato con 3.14.

Implementazione

```
# Definisco il valore di pi greco che userò per il calcolo dell'area.
pi = 3.14

# Definisco una funzione per calcolare l'area di una circonferenza a partire dal valore del suo raggio.
# Il risultato verrà stampato con due cifre decimali.
def area_circonferenza(raggio):
    # Converto il raggio in numero decimale (float) per garantire precisione nel calcolo anche se viene passato come stringa o intero.
    raggio = float(raggio)
```

```
9
10      # Applico la formula dell'area del cerchio:
11      # A = π * r^2
12      # Dove pi è il valore definito sopra e r è il raggio della
        circonferenza.
13      area = pi * (raggio ** 2)
14
15      # Stampo il risultato formattato con due cifre decimali.
16      # Uso f-string per rendere l'output leggibile e chiaro.
17      print(f"L'area della circonferenza di raggio {raggio} è {area:.2
        f}")
18
19  # Esempio di chiamata della funzione.
20  area_circonferenza(5.234)
```

Output del programma

```
1  L'area della circonferenza di raggio 5.234 è 86.02
```

5.2 Esercizio 2 - Verifica di anno bisestile

Scrivi una funzione `is_leap(year)` che:

- prende come parametro un anno (numero intero);
- restituisce `True` se l'anno è bisestile, `False` altrimenti.

Regole: un anno è bisestile se è divisibile per 4, ma non per 100, a meno che non sia anche divisibile per 400.

Implementazione

```
1  # Funzione che stabilisce se un anno è bisestile usando una
     variabile di stato (leap_year) inizializzata come False, che può
     cambiare solo se le condizioni lo giustificano.
2  def is_leap(year):
3      # Inizialmente considero che l'anno NON sia bisestile
4      leap_year = False
```

```
5
6       # Se è divisibile per 4, allora può esserlo
7       if year % 4 == 0:
8
9           # Se NON è divisibile per 100 → è bisestile
10          if year % 100 != 0:
11              leap_year = True
12
13          # Se invece è divisibile per 100, potrebbe NON esserlo.
14          # In questo caso aggiungo un'ulteriore condizione: se è
    divisibile anche per 400, allora è comunque bisestile.
15          elif year % 400 == 0:
16              leap_year = True
17
18      # Alla fine restituisco il risultato sotto forma di messaggio
19      if leap_year:
20          return f"{year} è un anno bisestile"
21      else:
22          return f"{year} non è un anno bisestile"
23
24  # Esempio di chiamata della funzione
25  is_leap(2020)
```

Output del programma

```
1  2020 è un anno bisestile
```

5.3 Esercizio 3 - Verifica di anagrammi

Definisci una funzione che, prendendo in input due parole, verifichi se sono anagrammi. Due parole sono anagrammi se contengono gli stessi caratteri, con le stesse quantità, anche se in ordine diverso.

Implementazione

```
1  # Funzione che verifica se due parole sono anagrammi.
```

```python
# Due parole sono anagrammi se contengono esattamente le stesse lettere, con la stessa quantità, anche se in ordine diverso.
def is_anagram(parola1, parola2):
    # Converto entrambe le parole in minuscolo con .lower() in modo da ignorar eventuali differenze tra lettere maiuscole e minuscole.
    # ATTENZIONE: lower è una funzione e va sempre scritta con le parentesi ()
    parola1 = sorted(parola1.lower())
    parola2 = sorted(parola2.lower())

    # Uso sorted() per trasformare le parole in liste di lettere ordinate.
    # Esempio: "strani" → ['a', 'i', 'n', 'r', 's', 't']
    # In questo modo posso confrontare direttamente le lettere, in ordine.

    # Se le due liste di lettere ordinate sono uguali, allora le parole sono anagrammi.
    if parola1 == parola2:
        return "Le due parole sono anagrammi"
    else:
        return "Le due parole non sono anagrammi"
```

5.4 Esercizio 4 - Validazione di indirizzi email

Scrivi una funzione che, prendendo in input una lista di indirizzi email, restituisca una nuova lista contenente solo gli indirizzi validi, ordinati alfabeticamente.
Un indirizzo email è considerato valido se:

- ha il formato: `nomeutente@dominio.estensione`;
- il nome utente contiene solo lettere, numeri, trattini (−) o underscore (_);
- il dominio contiene solo lettere e numeri;
- l'estensione è lunga al massimo 3 caratteri.

Implementazione

```
# Utilizzo *args per permettere alla funzione di ricevere un numero qualsiasi di email,
# che Python raccoglie automaticamente in una tupla (immutabile).
# Subito dopo converto la tupla in lista per poterla ordinare.
def verifica_email(*args):
    # 1. Ordino alfabeticamente gli argomenti (email)
    email_list = sorted(list(args))

    # 2. Preparo una lista vuota in cui salverò solo le email valide
    email_valide = []

    # 3. Ciclo su ogni email e applico parsing e validazioni
    #    Blocco try-except: se qualcosa fallisce (parsing o validazione),
    #    l'eccezione viene catturata e salto subito all'email successiva.
    for email in email_list:
        try:
            # 3a) Parsing: separo utente, dominio ed estensione
            user_part, domain_part = email.split("@")
            domain, extension = domain_part.split(".")

            # ------------------------------ VALIDAZIONI ----------------------------------
            # Controllo carattere per carattere il nome utente:
            # solo lettere, numeri, '-' e '_' sono consentiti
            for char in user_part:
                if not (char.isalnum() or char in "-_"):
                    raise ValueError("Carattere non valido nel nome utente")

            # Verifica che il dominio sia alfanumerico
            if not domain.isalnum():
                raise ValueError("Dominio non valido")

            # Verifica che l'estensione non superi i 3 caratteri
            if len(extension) > 3:
                raise ValueError("Estensione troppo lunga")
            #
            ------------------------------------------------------------------------------



            # Se tutte le validazioni passano, salvo l'email
            email_valide.append(f"{user_part}@{domain}.{extension}")
```

```
38
39          except ValueError as e:
40              # Qui ignoro l'email malformata e continuo con la
    prossima
41              continue
42
43      # 4. Stampo il risultato formattato
44      print("Email valide:")
45      for e in email_valide:
46          print(f"{e}")
```

Spiegazione del blocco `try-except`: Nel codice sopra, il blocco `try` racchiude le operazioni che possono generare errori (parsing e validazioni). Se in `try` si solleva una `ValueError`, l'esecuzione salta immediatamente al blocco `except`, dove con `continue` si scarta l'email corrente senza interrompere l'intero ciclo.

Output del programma

```
1 Email valide:
2 correct-mail@site.eu
3 guidopacciani@virgilio.it
4 rinabertocchi@gmail.com
```

Note utili

- L'utilizzo del costrutto `try-except` consente di gestire errori senza interrompere il programma.
- La funzione `isalnum()` restituisce `True` se e solo se la stringa su cui viene chiamata non è vuota e tutti i suoi caratteri sono alfabetici (A–Z, a–z) oppure numerici (0–9); in tutti gli altri casi (stringa vuota o presenza di spazi o simboli) restituisce `False`.
- Il filtro finale restituisce solo email con estensione massima di 3 caratteri.

Schema di base per la definizione di una funzione in Python

1. **Intestazione (signature)**

```
def nome_funzione(arg1: Tipo, arg2: Tipo = Default) -> Tipo:
```

2. **Docstring**

```
"""Breve descrizione della funzione.
Parametri:
arg1 (Tipo): descrizione.
arg2 (Tipo): descrizione (default = Default).
Ritorna: Tipo: descrizione del valore restituito.
"""
```

3. **Validazione degli input**

```
 # Se necessario, verifiche preliminari
if not condizione(arg1, arg2):
raise ValueError("messaggio di errore")
```

4. **Elaborazione principale**

```
 # Logica core della funzione
result = ...
```

5. **Operazioni secondarie (opzionali)**

```
 # Eventuale logging o trasformazioni aggiuntive
logger.info(f"Result = {result}")
```

6. **Ritorno del risultato**

```
 return result
```

Capitolo 6

Programmazione ad Oggetti

Il paradigma della programmazione ad oggetti consente di strutturare il codice mediante la definizione di classi e l'istanza di oggetti. In questo capitolo vengono illustrate le modalità di creazione e utilizzo delle classi, il meccanismo di istanziazione degli oggetti e i concetti di metodi, attributi e costruttori.

Introduzione teorica

Perché nascono le classi? Un problema concreto

Immagina di dover gestire centinaia di circonferenze in un programma di grafica. Per ciascuna servono raggio, diametro, perimetro e area. Un primo tentativo potrebbe essere usare liste parallele:

```
raggi      = [5, 7, 12]
diametri   = [r * 2 for r in raggi]
perimetri  = [r * 2 * math.pi for r in raggi]
aree       = [math.pi * r**2 for r in raggi]
```

Basta eliminare o inserire un elemento nella lista `raggi` alla posizione sbagliata e l'intero allineamento va perso. **Serve un contenitore che mantenga insieme i dati correlati**. Qui entra in gioco l'*oggetto*.

Classe → stampo per oggetti

Una **classe** è come uno *stampo per biscotti*: definisce una forma unica con cui cuocere tanti biscotti quasi identici. In codice:

```
class Circle:        # lo stampo (template)
```

```
    pass

c1 = Circle()          # primo biscotto
c2 = Circle()          # secondo biscotto
```

Gli oggetti `c1` e `c2` sono vuoti: non contengono ancora alcun raggio. Come si inizializzano?

Nascita dell'oggetto: il costruttore `__init__`

Quando invii `Circle()`, Python compie due passi:

1. Riserva memoria per un nuovo oggetto *vuoto*;
2. Esegue, se esiste, `Circle.__init__(oggetto, ...)`.

La funzione speciale `__init__` è quindi il *costruttore*: serve a fornire i dati iniziali all'oggetto mentre nasce.

```
class Circle:
    def __init__(self, raggio):
        print("-> Creo un nuovo Circle...")
        self.r = raggio            # salvo il raggio nell'oggetto
        print(f"   self.r = {self.r}")

c = Circle(5)
print("Oggetto creato:", c)
```

Output (semplificato):

```
-> Creo un nuovo Circle...
   self.r = 5
Oggetto creato: <__main__.Circle object at 0x...>
```

Così ogni istanza *porta con sé* il proprio raggio.

Chi è davvero `self`?

`self` è semplicemente il nome convenzionale del primo parametro che riferisce all'oggetto stesso. Quando Python chiama il costruttore:

```
c1 = Circle(7)             # sintassi "alta" di Python
# dietro le quinte diventa...
Circle.__init__(c1, 7)     # self  <-  c1
```

Quindi: «usa il nome `self` per accedere agli attributi di *questo* cerchio». Non c'è alcuna magia.

Metodi di istanza

Qualsiasi funzione definita dentro la classe che abbia `self` come primo parametro è un *metodo di istanza*: opera sui dati di un singolo oggetto.

```
class Circle:
    def __init__(self, r):
        self.r = float(r)

    def diameter(self):
        return self.r * 2

    def area(self):
        from math import pi
        return pi * self.r ** 2
```

Esempio d'uso con tracciamento esplicito:

```
c = Circle(3)
print("diametro:", c.diameter())
print("area:", c.area())
```

Metodi speciali (*dunder* methods)

I metodi con doppio underscore `__metodo__` permettono di personalizzare il comportamento predefinito di un oggetto.

- `__str__` – rappresentazione leggibile con `print`.
- `__eq__` – confronto con `==`.
- `__add__` – somma con `+` (se ha senso per quell'oggetto).

Esempio:

```
class Circle:
    ...
    def __str__(self):
        return f"Circle(r={self.r})"
```

Ora `print(c)` stampa `Circle(r=3)` anziché una stringa criptica.

Underscore singolo: una convenzione, non una regola

Un attributo che inizia con `_` (singolo underscore) segnala "uso interno, non toccare". Python non lo rende davvero privato; è solo un cartello per altri programmatori.

In sintesi

1. Una **classe** è lo stampo; l'**oggetto** è il prodotto.
2. Il costruttore `__init__` si esegue alla nascita e inizializza gli attributi.
3. `self` è l'oggetto stesso: permette ai metodi di accedere ai dati *di quella* istanza.
4. I *dunder methods* personalizzano operatori e funzioni built-in.
5. Il singolo underscore è solo una cortesia: indica un attributo interno.

Nelle sezioni successive applicheremo questi concetti agli esercizi **Circle** e **Vector** per fissare la teoria con il codice.

6.1 Esercizio 1 - Classe Circonferenza

Definisci una classe `Circle` che, prendendo in input il raggio, permetta di calcolare diametro, perimetro e area. Documenta i metodi con docstring.

Implementazione

```
import math
from typing import Any

class Circle:
    """Rappresenta una circonferenza nel piano.

    Parameters
    ----------
    raggio : float
        Raggio strettamente positivo.

```

```
    Raises
    ------
    ValueError
        Se `raggio` non è positivo.
    """

    def __init__(self, raggio: float) -> None:
        if raggio <= 0:
            raise ValueError("Il raggio deve essere un numero positivo.")
        self._raggio: float = float(raggio)

    # ---------- Proprietà di sola lettura ----------
    @property
    def radius(self) -> float:
        """float: Raggio della circonferenza (read-only)."""
        return self._raggio

    # -------------- Metodi di calcolo --------------
    def diameter(self) -> float:
        """Restituisce il diametro (2 × r)."""
        return 2 * self._raggio

    def circumference(self) -> float:
        """Restituisce la lunghezza della circonferenza (2 π r)."""
        return 2 * math.pi * self._raggio

    def area(self) -> float:
        """Restituisce l'area (pi * r^2)."""
        return math.pi * (self._raggio ** 2)

    # -------------- Rappresentazioni ---------------
    def __str__(self) -> str:
        return f"Circonferenza di raggio {self._raggio:.2f}"

    def __repr__(self) -> str:
        return f"Circle(r={self._raggio})"
```

Esempio di utilizzo

```
c = Circle(5)

print(f"Raggio       : {c.radius:.2f}")
```

```
print(f"Diametro      : {c.diameter():.2f}")
print(f"Circonferenza: {c.circumference():.2f}")
print(f"Area          : {c.area():.2f}")
```

Output del programma

```
Raggio        : 5.00
Diametro      : 10.00
Circonferenza: 31.42
Area          : 78.54
```

Note utili

- Il costruttore `__init__` valida l'input e inizializza l'attributo `_raggio`.
- Ogni metodo ha una docstring che ne descrive il comportamento.
- Il metodo `__str__` rende leggibile la stampa dell'oggetto.

6.2 Esercizio 2 - Vettori numerici

Definisci una classe `Vector` che rappresenti un vettore numerico. Implementa somma, differenza, prodotto scalare, confronto di uguaglianza, metodi per la somma totale, norma e indicizzazione.

Implementazione

```
from math import sqrt
from typing import List, Union, Optional

class Vector:
    """
    Classe che rappresenta un vettore numerico unidimensionale.
    """

    def __init__(self, v: List[Union[int, float]]):
        self._v: List[float] = [float(val) for val in v]
```

```
11
12      def __str__(self) -> str:
13          return f"Vector({self._v})"
14
15      def __getitem__(self, i: int) -> float:
16          return self._v[i]
17
18      def __len__(self) -> int:
19          return len(self._v)
20
21      def __add__(self, other: "Vector") -> Optional["Vector"]:
22          if len(self) != len(other):
23              return None
24          return Vector([a + b for a, b in zip(self._v, other._v)])
25
26      def __sub__(self, other: "Vector") -> Optional["Vector"]:
27          if len(self) != len(other):
28              return None
29          return Vector([a - b for a, b in zip(self._v, other._v)])
30
31      def __mul__(self, other: "Vector") -> Optional[float]:
32          if len(self) != len(other):
33              return None
34          return sum(a * b for a, b in zip(self._v, other._v))
35
36      def __eq__(self, other: object) -> bool:
37          if not isinstance(other, Vector):
38              return False
39          return self._v == other._v
40
41      def sum(self) -> float:
42          return sum(self._v)
43
44      def norm(self) -> float:
45          return sqrt(sum(val ** 2 for val in self._v))
46
47  # Esempio di utilizzo
48  v = Vector([2, 14, 20, 7])
49  w = Vector([12, 6, 25, 2])
50
51  print(f"Il vettore v è composto da {v[0]}, {v[1]}, {v[2]} e {v[3]}")
52  print(f"mentre il vettore w è composto da {w[0]}, {w[1]}, {w[2]} e {
        w[3]}")
53  print(f"La loro somma dà luogo al vettore somma = {v + w}")
```

```
54 print(f"La loro differenza dà luogo al vettore differenza = {v - w}"
       )
55 print(f"Il prodotto scalare dà luogo allo scalare = {v * w}")
56 print(f"La norma del vettore v è {v.norm():.2f} e la norma del
       vettore w è {w.norm():.2f}")
```

Output del programma

```
1 Il vettore v è composto da 2.0, 14.0, 20.0 e 7.0 mentre il vettore w
      è composto da 12.0, 6.0, 25.0 e 2.0
2 La loro somma dà luogo al vettore somma = Vector([14.0, 20.0, 45.0,
      9.0])
3 La loro differenza dà luogo al vettore differenza = Vector([-10.0,
      8.0, -5.0, 5.0])
4 Il prodotto scalare dà luogo allo scalare = 622.0
5 La norma del vettore v è 25.48 e la norma del vettore w è 28.44
```

Note utili

- La classe implementa i principali operatori aritmetici come metodi speciali (`__add__`, `__sub__`, `__mul__`).
- I controlli su lunghezze diverse sono gestiti restituendo `None`.
- L'accesso agli elementi tramite indice è possibile grazie a `__getitem__`.

Guida rapida alla costruzione di una classe

Blocco	Metodo	Scopo	Quando?
Istanziazione	`__init__`	Inizializza attributi; valida input.	Quasi sempre.
	`__new__`	Crea l'oggetto prima di `__init__` (classi immutabili).	Raro.
Rappresentazione	`__repr__`	Stringa non ambigua per debug/log.	Buona pratica.
	`__str__`	Stringa leggibile da `print`.	Se vuoi un output "umano".
Attributi	`@property`	Espone attributo di sola lettura.	Incapsulamento leggero.
	`@x.setter`	Valida/trasforma in scrittura.	Se vuoi controlli su assegnazione.
Confronto	`__eq__` + `__lt__`...	Uguaglianza e ordinamento.	Se confronti o ordini istanze.
Sequenza	`__len__`, `__getitem__`	Rendono l'oggetto indicizzabile/iterabile.	Collezioni custom.
Operatori	`__add__`, `__mul__`...	Aritmetica naturale (`+`, `*`, ecc.).	Classi numeriche / vettori.
Gestione risorse	`__enter__` / `__exit__`	Abilita `with obj: ...`.	File, lock, connessioni.

Capitolo 7

Gestione degli errori: assert e try/except

Questo capitolo illustra gli strumenti fondamentali per il controllo degli errori e il rafforzamento della robustezza di un programma—le istruzioni `assert`, `try` e `except`. Viene spiegato come proteggere il codice da situazioni inattese, quali input non validi, operazioni illegali (ad esempio la divisione per zero) o incoerenze nei dati (come vettori di lunghezza diversa).

Teoria: assert, try, except

`assert`

L'istruzione `assert` permette di verificare che una condizione sia vera. Se la condizione è falsa, viene sollevata un'eccezione (`AssertionError`) e il programma si interrompe. È molto utile per:

- verificare pre-condizioni (es. dimensione di una lista);
- garantire l'integrità dei dati;
- scrivere codice più chiaro e robusto.

Sintassi:

```
assert condizione, "Messaggio di errore personalizzato"
```

`try / except`

Questo costrutto serve a gestire gli errori (eccezioni) che potrebbero verificarsi durante l'esecuzione del programma, evitando che esso vada in crash.

Funziona così:

```
try:
    # codice che potrebbe generare un errore
except Errore as e:
    # cosa fare se l'errore avviene
```

Utilità:

- Il programma non si interrompe in caso di errore.
- Possiamo stampare un messaggio più leggibile.
- Possiamo salvare l'errore nella variabile `e` ed usarla se ci serve.

7.1 Esercizio 1 – Vettori "re-loaded"

Nell'esercitazione precedente è stata implementata una classe per rappresentare vettori. In vari metodi veniva controllato che le lunghezze di due vettori coincidessero. Sostituire tali verifiche basate su `if` con l'istruzione `assert`.

Implementazione

```
from math import sqrt
from typing import List, Union


class Vector:
    """
    Classe che rappresenta un vettore numerico unidimensionale.
    """

    def __init__(self, v: List[Union[int, float]]):
        """
        Inizializza un oggetto Vector con una lista di numeri.
        I valori vengono convertiti in float per uniformità.
```

```
14
15          Parametri:
16              v (list): Lista di numeri (interi o decimali).
17          """
18          self._v = [float(val) for val in v]
19
20
21
22      def __getitem__(self, i: int) -> float:
23          """
24          Permette l'accesso agli elementi tramite indice.
25
26          Parametri:
27              i (int): Indice dell'elemento da restituire.
28
29          Ritorna:
30              float: Valore all'indice specificato.
31          """
32          return self._v[i]
33
34
35
36      def __len__(self) -> int:
37          """
38          Restituisce il numero di elementi del vettore.
39
40          Ritorna:
41              int: Lunghezza del vettore.
42          """
43          return len(self._v)
44
45
46
47      def __str__(self) -> str:
48          """
49          Rende il vettore stampabile in forma leggibile.
50
51          Ritorna:
52              str: Rappresentazione testuale del vettore.
53          """
54          return f"Vector({self._v})"
55
56
57
58      def __add__(self, other: "Vector") -> "Vector":
```

```
59          """
60          Somma elemento per elemento tra due vettori.
61          Solleva un'eccezione se le lunghezze sono diverse.
62
63          Ritorna:
64              Vector: Nuovo vettore somma.
65          """
66          assert len(self) == len(other), "I vettori hanno lunghezza
    differente"
67          return Vector([a + b for a, b in zip(self._v, other._v)])
68
69
70
71      def __sub__(self, other: "Vector") -> "Vector":
72          """
73          Differenza elemento per elemento tra due vettori.
74          Solleva un'eccezione se le lunghezze sono diverse.
75
76          Ritorna:
77              Vector: Nuovo vettore differenza.
78          """
79          assert len(self) == len(other), "I vettori hanno lunghezza
    differente"
80          return Vector([a - b for a, b in zip(self._v, other._v)])
81
82
83
84      def __eq__(self, other: object) -> bool:
85          """
86          Verifica se due vettori sono uguali elemento per elemento.
87          Solleva un'eccezione se le lunghezze sono diverse.
88
89          Ritorna:
90              bool: True se tutti gli elementi coincidono nell'ordine.
91          """
92          if not isinstance(other, Vector):
93              return False
94
95          assert len(self) == len(other), "I vettori hanno lunghezza
    differente"
96          return self._v == other._v
97
98
99
100     def __mul__(self, other: "Vector") -> float:
```

```
        """
        Calcola il prodotto scalare tra due vettori.
        Solleva un'eccezione se le lunghezze sono diverse.

        Ritorna:
            float: Valore del prodotto scalare.
        """
        assert len(self) == len(other), "I vettori hanno lunghezza differente"
        return sum(a * b for a, b in zip(self._v, other._v))



    def sum(self) -> float:
        """
        Calcola la somma degli elementi del vettore.

        Ritorna:
            float: Somma dei valori.
        """
        return sum(self._v)



    def norm(self) -> float:
        """
        Calcola la norma euclidea del vettore.

        Ritorna:
            float: Radice quadrata della somma dei quadrati degli elementi.
        """
        return sqrt(sum(x ** 2 for x in self._v))
```

Output del programma

```
# Esempio d'uso con gestione delle eccezioni
if __name__ == "__main__":
    v = Vector([2, 14, 20, 7])
    w = Vector([12, 6, 25])  # vettore più corto

    try:
        print(v + w)
```

```
        print(v - w)
        print(v * w)
    except AssertionError as e:
        print(e)  # "I vettori hanno lunghezza differente"
```

```
I vettori hanno lunghezza differente
```

7.2 Esercizio 2 – Calcolatrice con gestione errori

Definisci una funzione per eseguire dei semplici calcoli. La funzione prende in ingresso due valori numerici ed una stringa contenente un'operazione aritmetica (+, -, *, /) ed esegue tale operazione tra i due numeri.
Requisiti:

- Utilizza il `try/except` per gestire l'eccezione che avviene nel caso in cui venga inserito un valore non numerico.
- Utilizza `assert` per assicurarti che l'operatore sia tra quelli validi.
- Gestisci il caso di divisione per 0.

Implementazione

```
def calculator(a: float, b: float, op: str) -> float:
    """
    Esegue l'operazione aritmetica specificata tra due numeri.

    Parametri:
        a (float): Primo numero.
        b (float): Secondo numero.
        op (str): Operatore aritmetico ('+', '-', '*', '/').

    Ritorna:
        float: Risultato dell'operazione, oppure None se divisione
   per zero.
    """
    assert op in ["+", "-", "*", "/"], "Operatore non valido"

    if op == "+":
        return a + b
```

```
17
18      elif op == "-":
19          return a - b
20
21      elif op == "*":
22          return a * b
23
24      elif op == "/":
25          try:
26              return a / b
27          except ZeroDivisionError:
28              print("Non puoi dividere per 0")
29              return None
30
31
32
33  def chiedi_numero(testo_input: str) -> float:
34      """
35      Chiede all'utente un numero valido, ripetendo finché necessario.
36
37      Parametri:
38          testo_input (str): Messaggio da mostrare all'utente.
39
40      Ritorna:
41          float: Numero inserito dall'utente.
42      """
43      while True:
44          try:
45              return float(input(testo_input))
46          except ValueError:
47              print("Valore non valido, riprova.")
48
49
50
51  def chiedi_operatore(testo_input: str) -> str:
52      """
53      Chiede all'utente un operatore tra quelli validi (+, -, *, /).
54
55      Parametri:
56          testo_input (str): Messaggio da mostrare all'utente.
57
58      Ritorna:
59          str: Operatore aritmetico scelto.
60      """
61      while True:
```

```
62          op = input(testo_input)
63          if op in ["+", "-", "*", "/"]:
64              return op
65          else:
66              print("Operatore non valido, riprova.")
```

Output del programma

```
1  if __name__ == "__main__":
2      a = chiedi_numero("Inserisci il primo numero: ")
3      b = chiedi_numero("Inserisci il secondo numero: ")
4      op = chiedi_operatore("Inserisci l'operatore (+, -, *, /): ")
5
6      risultato = calculator(a, b, op)
7      if risultato is not None:
8          print("Risultato:", risultato)
```

```
1  Inserisci il primo numero: 5
2  Inserisci il secondo numero: 10
3  Inserisci l'operatore (+, -, *, /): /
4  Risultato: 0.5
```

Guida rapida: come gestire gli errori in Python

Strumento	Quando si usa	Esempio minimo
`assert`	Verificare che una condizione sia vera; se falsa solleva `AssertionError` con messaggio.	`assert x > 0, "x deve essere positivo"`
`raise`	Sollevare esplicitamente un'eccezione personalizzata o di sistema.	`raise ValueError("Input non valido")`
`try / except`	Gestire in modo sicuro codice che può sollevare errori.	`try: a=10/0; except ZeroDivisionError: print ("div 0")`
`else`	Eseguire codice solo se nel blocco `try` non è stata sollevata alcuna eccezione.	`try: r=1/2; except ZeroDivisionError: ...; else: print(r)`
`finally`	Codice che si esegue sempre, con o senza eccezione (rilascio risorse).	`try: f=open("f"); finally : f.close()`
`with ...`	Usare un context manager che apre/chiude automaticamente risorse.	`with open("file.txt") as f: data=f.read()`

Le eccezioni più frequenti

Eccezione	Quando si verifica	Esempio minimale
`AssertionError`	Fallisce un'asserzione `assert`.	`assert 2+2==5`
`ValueError`	Argomento di tipo corretto ma valore inadatto.	`int("abc")`
`TypeError`	Operazione applicata a tipo non compatibile.	`"a" + 3`
`IndexError`	Indice fuori range in sequenze.	`[1,2][5]`
`KeyError`	Chiave inesistente in dizionario.	`["k"]`
`ZeroDivisionError`	Divisione per zero.	`10/0`
`FileNotFoundError`	File o percorso inesistente in I/O.	`open("nope.txt")`
`SyntaxError`	Errore di sintassi (parsing).	`if True print("hi")`
`ImportError`	Import fallito: modulo o attributo mancante.	`import fantamod`
`AttributeError`	Attributo/method assente nell'oggetto.	`[].push(1)`
`NameError`	Variabile non definita nello scope.	`print(xyz)`
`OverflowError`	Numero troppo grande per il tipo.	`math.exp(1000)`
`IndentationError`	Indentazione errata nel codice.	Codice non allineato
`MemoryError`	Memoria esaurita (allocazioni enormi).	`a=[0]*10**10`
`RuntimeError`	Errore generico a runtime (es. ricorsione infinita).	`sys.setrecursionlimit(10); f=lambda x:f(x); f(0)`

Capitolo 8

Operare sui file

Operare sui file è una competenza essenziale nella programmazione, utile per gestire dati esterni, scambiare informazioni tra programmi e conservare risultati nel tempo. In questo capitolo si apprenderà come leggere, analizzare e scrivere file di testo e CSV, applicando tecniche semplici e dirette per svolgere le operazioni più comuni sui dati.

Per lavorare sui file in Python è utile conoscere la differenza tra due concetti fondamentali: • Un *metodo* è una funzione legata a un oggetto o modulo, e si richiama con parentesi tonde. Esempio: `file.readlines()` restituisce una lista di righe dal file. • Un *attributo* è una proprietà o caratteristica di un oggetto accessibile direttamente senza parentesi tonde. Esempio: `file.closed` restituisce `True` se il file è stato chiuso, altrimenti `False`.

Questa distinzione è utile per orientarsi correttamente quando si consultano metodi e attributi delle classi in Python durante la manipolazione di file e dati.

8.1 Esercizio 1 - Filtra i proverbi

Leggi il contenuto del file `proverbi.txt` e scrivi su un nuovo file chiamato `proverbi_filtrati.txt` solo i proverbi che iniziano per vocale oppure che sono più brevi di 25 caratteri.

Implementazione

```
# Questo comando chiede di autorizzare Colab ad accedere ai file su Drive.
from google.colab import drive
```

```
drive.mount('/content/drive')

# Definisco il percorso completo al file da leggere.
# Utilizzo una stringa "raw" (cioè r"...") per evitare che gli spazi e i simboli nel percorso (come ":" o "\") vengano interpretati da Python.
file_path = (
    r"/content/drive/MyDrive/Colab Notebooks/"
    r"Modulo 1: Programmazione con Python/Esercitazioni/"
    r"Esercitazione_8_Operare_sui_File/proverbi.txt"
)

# Apro il file in modalità lettura ("r") e salvo ogni riga in una lista.
# Ogni elemento della lista è una riga (proverbio) con il carattere di a capo \n alla fine.
with open(file_path, "r") as proverbs_file:
    proverbs_filtered_file = proverbs_file.readlines()

# Creo la lista di vocali e inizializzo la lista vuota di proverbi filtrati.
vocali = "AEIOU"
proverbi_filtrati = []

# Creo un ciclo per analizzare ciascun proverbio e, se soddisfa almeno una delle due condizioni richieste (inizia per vocale oppure contiene meno di 25 caratteri), lo aggiungo alla lista proverbi_filtrati.
for row in proverbs_filtered_file:
    # .strip() rimuove eventuali spazi iniziali/finali e il \n a fine riga, così posso controllare correttamente il primo carattere e la lunghezza
    row = row.strip()
    # Controllo se la riga non è vuota ( if row ) e se soddisfa almeno una delle due condizioni richieste.
    if row and (row[0] in vocali or len(row) < 25):
        # Aggiungo nuovamente il carattere di a capo per mantenere il formato corretto
        proverbi_filtrati.append(row + "\n")

# Creo un nuovo file di testo nella stessa directory e scrivo in esso tutti i proverbi filtrati, riga per riga
output_path = (
    r"/content/drive/MyDrive/Colab Notebooks/"
    r"Modulo 1: Programmazione con Python/Esercitazioni/"
```

```
    r"Esercitazione_8_Operare_sui_File/proverbi_filtrati.txt"
)

with open(output_path, "w") as new_file:
    new_file.writelines(proverbi_filtrati)
```

8.2 Esercizio 2 - Analisi di magazzino

Apri il file `shirts.csv` e calcola le seguenti statistiche:

- Numero totale di prodotti presenti
- Valore totale del magazzino
- Prezzo medio dei prodotti
- Numero di prodotti per ogni colore
- Numero di prodotti per ogni taglia

Metodo 1 - Procedurale dettagliato

```
import csv

input_file = (
    r"/content/drive/MyDrive/Colab Notebooks/"
    r"Modulo 1: Programmazione con Python/Esercitazioni/"
    r"Esercitazione_8_Operare_sui_File/shirts.csv"
)

output_file = (
    r"/content/drive/MyDrive/Colab Notebooks/"
    r"Modulo 1: Programmazione con Python/Esercitazioni/"
    r"Esercitazione_8_Operare_sui_File/shirts_stats.json"
)

#-------------Leggo e rendo il file analizzabile----------

with open(input_file, "r") as input_file:
```

```
18    # Creo un oggetto csv_reader, che è un iteratore capace di leggere
        riga per riga il file. NOTA: al momento non ho ancora iniziato a
        leggere
19    csv_reader = csv.reader(input_file)
20    # Inizio a leggere: leggo solo la prima riga e la memorizzo in "
       header"
21    header = next(csv_reader)
22    # Ora il puntatore si trova all'inizio della seconda riga! Posso
       quindi memorizzare tutte le restanti righe di csv_reader in una
       nuova lista "rows"
23    rows = list(csv_reader)
24
25    # Ora posso usare rows quante volte voglio per analisi, conteggi,
       ecc.
26
27  #--------------Analisi statistiche---------------
28
29    # 1. Conteggio dei prodotti
30    counter = len(rows)
31
32    # 2. Somma dei prezzi: per ogni riga "r", prendo il 4 elemento (il
        prezzo) e lo sommo al prezzo totale
33    total_price = sum(float(r[3]) for r in rows)
34    # Arrotondo il prezzo totale a 2 cifre decimali dopo la vigola
35    total_price = round(total_price, 2)
36
37    # 3. Calcolo del prezzo medio dei prodotti
38    average_price = total_price / counter if counter > 0 else 0
39    # Arrotondo il prezzo medio a 2 cifre decimali dopo la vigola
40    average_price = round(average_price, 2)
41
42    # 4. Creo un set dove memorizzo una sola volta tutti i colori
       disponibili. Successivamente creo un dizionario chiave:valore per
        ciascun colore trovato, mettendo come chiave il colore, e come
       valore 0 (temporaneamente)
43    colors_set = {r[2] for r in rows}
44    colors_counter = {c: 0 for c in colors_set}
45
46    # 5. Insieme delle taglie: utilizzo lo stesso procedimento del
       punto precedente
47    sizes_set = {r[1] for r in rows}
48    sizes_counter = {s: 0 for s in sizes_set}
49
50    # 6. Conto il numero di prodotti per colore
51    for r in rows:
```

```
52      colore = r[2]
53      # Estraggo il valore associato alla chiave "colore" e lo aumento
        di 1
54      colors_counter[colore] += 1
55
56    # 7. Conto il numero di prodotti per taglia
57    for r in rows:
58      taglia = r[1]
59      sizes_counter[taglia] += 1
60
61  #-----------Stampa a schermo dei risultati------------
62
63    print(f"Il numero totale di prodotti equivale a {counter}")
64    print(f"Il valore totale del magazzino corrisponde a {total_price}
        €")
65    print(f"Il prezzo medio dei prodotti è {average_price} €")
66    print(f"Il numero di prodotti per ogni colore è {colors_counter}")
67    print(f"Il numero di prodotti per ogni taglia è {sizes_counter}")
68
69
70  #---------Salvataggio dei risultati in json-----------
71
72  import json
73
74  # Creo il dizionario "stats", che contiene tutte le statistiche
       calcolate.
75  stats = {
76      "numero_prodotti": counter,
77      "valore_totale_magazzino": total_price,
78      "prezzo_medio": average_price,
79      "conteggio_per_colore": colors_counter,
80      "conteggio_per_taglia": sizes_counter
81  }
82
83  # Apro un file (se non esiste, viene creato) in modalità scrittura e
       utilizzo il metodo .dump del modulo json per salvare il
       dizionario all'interno del file in formato leggibile (indentato).
84  with open(output_file, "w") as output_file:
85      json.dump(stats, output_file, indent=2)
```

Output del programma

```
1  Il numero totale di prodotti equivale a 100
```

```
Il valore totale del magazzino corrisponde a 1247.5 €
Il prezzo medio dei prodotti è 12.47 €
Il numero di prodotti per ogni colore è {'rosso': 20, 'bianco': 47, 'verde': 33}
Il numero di prodotti per ogni taglia è {'L': 26, 'M': 25, 'S': 25, 'XL': 24}
```

Metodo 2 - Diretto e compatto con DictReader

```
#-------------Lettura e inizializzazione-------------

import csv
import json

# Inizializzo un dizionario vuoto che conterrà tutte le statistiche
stats = {
    "prodotti": 0,           # Conta totale dei prodotti
    "valore_totale": 0,      # Somma dei prezzi di tutti i prodotti
    "taglie": {},            # Dizionario per contare i prodotti per taglia
    "colore": {}             # Dizionario per contare i prodotti per colore
}

#-------------Analisi diretta del file CSV--------------

# Apro il file shirts.csv e lo leggo riga per riga usando DictReader che trasforma ogni riga in un dizionario con chiavi basate sull'intestazione
with open(
    r"/content/drive/MyDrive/Colab Notebooks/"
    r"Modulo 1: Programmazione con Python/Esercitazioni/"
    r"Esercitazione_8_Operare_sui_File/shirts.csv") as csv_file:
    csv_reader = csv.DictReader(csv_file)

    for row in csv_reader:
        # 1. Incremento il numero totale di prodotti
        stats["prodotti"] += 1

        # 2. Sommo il prezzo del prodotto corrente al valore totale
        stats["valore_totale"] += float(row["prezzo"])

```

```
30          # 3. Registro la taglia: se già presente, incremento;
    altrimenti
31          # inizializzo a 1
32          if row["taglia"] in stats["taglie"]:
33              stats["taglie"][row["taglia"]] += 1
34          else:
35              stats["taglie"][row["taglia"]] = 1
36
37          # 4. Registro il colore con la stessa logica
38          if row["colore"] in stats["colore"]:
39              stats["colore"][row["colore"]] += 1
40          else:
41              stats["colore"][row["colore"]] = 1
42
43  #-----------Elaborazione finale dei dati------------
44
45  # Arrotondo il valore totale a 2 cifre decimali
46  stats["valore_totale"] = round(stats["valore_totale"], 2)
47
48  # Calcolo il prezzo medio per prodotto e lo aggiungo al dizionario
49  stats["prezzo_medio"] = round(stats["valore_totale"] / stats["
    prodotti"], 2)
50
51  #-----------Output e salvataggio in JSON-----------
52
53  # Stampo il dizionario completo con tutte le statistiche
54  print(stats)
55
56  # Salvo le stesse statistiche in un file JSON, con indentazione per
    leggibilità
57  with open(
58      r"/content/drive/MyDrive/Colab Notebooks/"
59      r"Modulo 1: Programmazione con Python/Esercitazioni/"
60      r"Esercitazione_8_Operare_sui_File/shirts_stats.json", "w") as
    json_file:
61      json.dump(stats, json_file, indent=3)
62
63  #----------Note: Differenze rispetto al mio metodo-------
64
65  # - Questo approccio non legge prima tutte le righe: analizza il
    file riga per riga
66  # - Usa direttamente csv.DictReader per lavorare con nomi colonna
    invece che indici
67  # - I dizionari per colori e taglie vengono creati "man mano", senza
     set
```

```
68 # iniziali
69 # - C'è un solo ciclo for, che fa tutto: conteggio, somma e
       classificazione
70
71 # Metafora:
72 # È come contare magliette in un negozio.
73 # Nel primo metodo esposto:
74 #   - Faccio un primo giro per segnare su un foglio tutte le taglie
       esistenti
75 #   - Poi faccio un secondo giro per contarle una a una.
76 # Nel secondo metodo esposto:
77 #   - Prendo una maglietta alla volta.
78 #   - Se la taglia non è ancora sul foglio, la aggiungo e segno "1".
79 #   - Se la taglia c'è già, aumento il numero.
80
81 # Entrambi i metodi funzionano: il primo è più analitico e separa i
       passaggi, il secondo è più snello e diretto, ma richiede più
       attenzione sul flusso.
```

Tabella dei principali metodi e moduli utilizzati

Modulo/Metodo	Tipo	Descrizione
`Built-in`	Modulo integrato	Metodi integrati direttamente in Python
`open()`	Metodo	Apre un file in modalità lettura, scrittura o altro tipo specificato.
`round()`	Metodo	Arrotonda un numero al numero specificato di cifre decimali.
`File object`	Oggetto file	Metodi applicabili agli oggetti file aperti con `open()`
`readlines()`	Metodo	Legge tutte le righe di un file e le restituisce come lista di stringhe.
`write()`	Metodo	Scrive una stringa in un file aperto.
`writelines()`	Metodo	Scrive una lista di stringhe in un file aperto.
`csv`	Modulo	Permette di leggere e scrivere file CSV
`csv.reader()`	Metodo	Legge file CSV riga per riga, restituendo liste.
`csv.DictReader()`	Metodo	Legge file CSV riga per riga, restituendo dizionari basati sulle intestazioni.
`json`	Modulo	Gestisce la serializzazione e deserializzazione dei dati JSON.
`json.dump()`	Metodo	Salva un oggetto Python (es. dizionario) in formato JSON su file.
`dict`	Struttura dati	Metodi delle strutture dati dizionario.
`.get()`	Metodo	Restituisce il valore associato a una chiave, o un valore di default se la chiave è assente.
`str`	Tipo stringa	Metodi integrati per la manipolazione delle stringhe.
`.strip()`	Metodo	Rimuove spazi e caratteri speciali all'inizio e alla fine di una stringa.

Capitolo 9

Standard Library: moduli e applicazioni

Introduzione alla Standard Library

La **Standard Library** di Python è una vasta raccolta di moduli pronti all'uso che coprono esigenze frequenti—gestione di date, file, numeri, interazione con il sistema operativo e molto altro—evitando di dover implementare soluzioni da zero. Conoscerne e padroneggiarne i componenti consente di scrivere codice più potente, leggibile e conciso.

Nel presente capitolo verranno presentati alcuni fra i moduli più utili della Standard Library, applicandoli a scenari pratici.

9.1 Esercizio 1 - Poesie su un file

Leggi tutte le poesie dalla directory `poesie` e salvale all'interno di un unico file chiamato `raccolta_poesie.txt`, che dovrà essere creato all'interno di una nuova directory chiamata `raccolta`.
Prima di ogni poesia, inserisci un'intestazione con un contatore numerico (es: `POESIA 1`, `POESIA 2`, ecc.).

Implementazione

```
1 # Collego il mio Google Drive a Colab forzando il remount se
    necessario
```

```
from google.colab import drive
drive.mount('/content/drive', force_remount=True)
```

```
# Importo il modulo della Standard Library per operazioni sul file system
import os

# Memorizzo nella variabile "input_dir" il percorso della cartella che contiene le poesie.
# Uso una stringa raw (r"...") per evitare che i caratteri speciali nel percorso come spazi e backslash) causino errori.
input_dir = (
    r"/content/drive/MyDrive/Colab Notebooks/"
    r"Modulo 1: Programmazione con Python/Esercitazioni/"
    r"Esercitazione_9_Standard_Library/Poesie"
)

# Costruisco dinamicamente il percorso della cartella "Raccolta" che conterrà il file finale.
# Il metodo os.path.join unisce i due segmenti del percorso in modo sicuro.
output_dir = os.path.join(input_dir, "Raccolta")

# Verifico se la cartella di output esiste già.
# Il metodo os.path.exists restituisce True se il percorso esiste.
# Se non esiste, la creo con il metodo os.makedirs
if not os.path.exists(output_dir):
    os.makedirs(output_dir)

# Creo il percorso completo del file di output "raccolta_poesie.txt" all'interno della cartella "Raccolta"
output_file = os.path.join(output_dir, "raccolta_poesie.txt")

# Apro il file finale in modalità scrittura ("w").
# Se esiste già, verrà sovrascritto. Se non esiste, viene creato.
with open(output_file, "w") as f:

    # Scorro tutti i file e cartelle presenti nella directory di input.
    # os.listdir restituisce una lista dei nomi (stringhe) di tutto ciò che c'è nella cartella.
    # La funzione built-in "enumerate" mi fornisce anche un indice numerico (i), utile per numerare le poesie.
    for i, filename in enumerate(os.listdir(input_dir)):
```

```
33
34          # Creo il percorso assoluto di ciascun elemento con os.path.
    join (unione di cartella + nome file)
35          full_path = os.path.join(input_dir, filename)
36
37          # Salto l'elemento se è una cartella (come "Raccolta")
    invece che un file contenente una poesia (.txt)
38          # Il metodo os.path.isdir restituisce True se il percorso è
    una directory.
39          if os.path.isdir(full_path):
40              continue
41
42          # Se l'if restituisce False, apro il file della poesia in
    sola lettura ("r").
43          # Leggo tutto il contenuto del file e lo scrivo nel file di
    output con un'intestazione numerata.
44          with open(full_path, "r") as f2:
45              f.write(f"POESIA {i+1}\n")  # intestazione prima di ogni
     poesia
46              f.write(f2.read())          # contenuto della poesia
47              f.write("\n\n")  # Due a capo per creare una riga bianca
     visibile
```

9.2 Esercizio 2 - Quanto manca al tuo compleanno?

Inserisci in input la data del tuo compleanno nel formato `giorno/mese` (es. `11/06`). Scrivi un programma che calcoli quanti giorni mancano al tuo prossimo compleanno rispetto alla data attuale, e stampa il risultato in output.

Implementazione

```
1  # Importo la classe datetime dal modulo datetime
2  from datetime import datetime
3
4  # Ottengo la data e ora attuali usando il metodo .now()
5  # Restituisce un oggetto datetime completo (con anno, mese, giorno,
       ora, minuti, secondi...)
6  now = datetime.now()
7
```

```
# Richiedo all'utente di inserire la data del suo compleanno nel formato giorno/mese. Il risultato è una stringa, es: "30/06"
birthdate_str = input("Inserisci la data del tuo compleanno "+
    "in formato 'giorno/mese': ")

# Converto la stringa inserita in un oggetto datetime.
# Il metodo .strptime(stringa, formato) → converte una stringa in una data secondo il formato specificato.
# %d = giorno (es. 11), %m = mese (es. 06)
birthdate = datetime.strptime(birthdate_str, "%d/%m")

# A questo punto l'oggetto birthdate sarà, ad esempio, 30/06

# Sostituisco l'anno della data inserita con l'anno corrente
# Il metodo .replace() → restituisce una nuova data con il campo modificato
birthdate = birthdate.replace(year=now.year)

# A questo punto l'oggetto birthdate sarà, ad esempio, 30/06/2025

# Mostro all'utente la data di oggi, formattata in modo leggibile (solo giorno/mese).
# Il metodo .strftime(formato) → converte una data in una stringa formattata
print("Data di oggi: " + now.strftime("%d/%m"))

# Se il compleanno deve ancora arrivare quest'anno:
if birthdate > now:
    print("Il %s deve ancora venire" % birthdate.strftime("%d/%m"))

    # Calcolo i giorni mancanti al compleanno
    # La differenza tra due datetime restituisce un oggetto timedelta
    # L'attributo .days restituisce il numero intero di giorni
    days_left = birthdate - now       #days_left è un oggetto timedelta
    print("Mancano ancora %d giorni" % days_left.days)

# Altrimenti, il compleanno è già passato:
else:
    print("Il %s è già passato" % birthdate.strftime("%d/%m"))

    # Calcolo i giorni mancanti al compleanno dell'anno successivo
    next_birthday = birthdate.replace(year=now.year + 1)
    days_left = next_birthday - now
```

```
46      print("Considerando l'anno prossimo, mancano ancora "+
47            "%d giorni" % days_left.days)
```

9.3 Esercizio 3 - Equazioni per il Machine Learning

Implementa le seguenti funzioni di attivazione: Sigmoide, ReLU, Tangente iperbolica. Implementa anche la funzione di costo `log loss` usando il modulo `math` dove possibile.

Implementazione

```
1  # Importo il modulo math per usare funzioni matematiche
2
3  import math
4
5  # Funzione Sigmoide
6  def sigmoide(z):
7      """
8      Calcola la funzione sigmoide:
9          σ(z) = 1 / (1 + e^(-z))
10     Valori tipici tra 0 e 1.
11     """
12     return 1 / (1 + math.exp(-z))
13
14 # Funzione Sigmoide
15 def relu(z):
16     """
17     Calcola la funzione ReLU (Rectified Linear Unit):
18         ReLU(z) = 0 se z < 0
19                   z se z >= 0
20     """
21     return 0 if z < 0 else z
22
23 # Funzione di attivazione "unità lineare rettificata"
24 def tanh(z):
25     """
26     Calcola la funzione tangente iperbolica secondo la definizione:
27         tanh(z) = (1 - e^(-2z)) / (1 + e^(-2z))
28     """
29     numeratore = 1 - math.exp(-2 * z)
```

```
    denominatore = 1 + math.exp(-2 * z)
    return numeratore / denominatore

# Funzione di costo
def log_loss(y, a):
    """
    Calcola la funzione di costo log loss:
        J(y, a) = -1/N * Σ [y_i * log(a_i) + (1 - y_i) * log(1 - a_i)]

    - y: lista di etichette vere (0 o 1)
    - a: lista di predizioni del modello (valori tra 0 e 1)

    Usa una protezione numerica con epsilon per evitare log(0).
    """
    assert len(y) == len(a), "Le liste y e a devono avere la stessa lunghezza"

    epsilon = 1e-15  # Piccolo valore per evitare log(0)
    loss = 0

    for y_i, a_i in zip(y, a):
        # Protezione numerica: limito a_i tra [ε, 1-ε]
        a_i = max(min(a_i, 1 - epsilon), epsilon)

        # Calcolo il contributo del singolo termine
        term = y_i * math.log(a_i) + (1 - y_i) * math.log(1 - a_i)
        loss += term

    # Media e negazione del risultato
    return -loss / len(y)
```

```
# Quando si testa una funzione e si assegna il suo risultato a una variabile, è importante **non usare lo stesso nome della funzione ** per la variabile.
# Altrimenti si sovrascrive la funzione e non sarà più richiamabile.
# Ad esempio:
#     sigmoide = sigmoide(1)  sovrascrive la funzione sigmoide
#     sig_result = sigmoide(1) corretto

# Test delle funzioni di attivazione e della log loss

sig_result = sigmoide(1)
print(f"La sigmoide di 1 è: {sig_result:.2f}")
```

```
11
12 relu_result = relu(-0.12)
13 print(f"La ReLU di -0.12 è: {relu_result:.2f}")
14
15 tanh_result = tanh(1)
16 print(f"La tangente iperbolica di 1 è: {tanh_result:.2f}")
17
18 # Dati di esempio: etichette reali (y) e predizioni (a)
19 y = [1, 0, 1, 1, 0]
20 a = [0.75, 0.4, 0.6, 0.8, 0.1]
21
22 loss_result = log_loss(y, a)
23 print(f"La log loss sui dati di esempio è: {loss_result:.2f}")
```

Output del programma

```
1 La sigmoide di 1 è: 0.73
2 La ReLU di -0.12 è: 0.00
3 La tangente iperbolica di 1 è: 0.76
4 La log loss sui dati di esempio è: 0.33
```

Moduli usati in questo capitolo

Modulo	Funzionalità principale + Metodi/Attributi utilizzati	Tipo
`os`	`os.path.join()` – unisce percorsi di file (metodo) `os.path.exists()` – verifica se un percorso esiste (metodo) `os.makedirs()` – crea directory anche annidate (metodo) `os.listdir()` – restituisce i contenuti di una directory (metodo) `os.path.isdir()` – verifica se un percorso è una directory (metodo)	Modulo + Metodi
`datetime`	`datetime.now()` – restituisce la data/ora attuale (metodo) `datetime.strptime()` – converte stringa in data (metodo) `datetime.strftime()` – converte data in stringa (metodo) `datetime.replace()` – crea una nuova data con un campo modificato (metodo) `timedelta.days` – restituisce giorni da un oggetto timedelta (attributo)	Modulo + Metodi/Attributi
`math`	`math.exp()` – calcola l'esponenziale (metodo) `math.log()` – calcola il logaritmo naturale (metodo)	Modulo + Metodi

Capitolo 10

Pacchetti esterni: PyPI e `pip`

Python mette a disposizione, oltre alla Standard Library, migliaia di *pacchetti esterni* pubblicati dalla comunità sul **Python Package Index (PyPI)**. Questi pacchetti ampliano le funzionalità di base – data-science, imaging, automazione, machine-learning, sviluppo web – riducendo la necessità di scrivere soluzioni da zero.

I pacchetti esterni *non* fanno parte della Standard Library: devono perciò essere installati separatamente, per esempio con `pip install pillow`.

In questo capitolo verrà usato il pacchetto `Pillow` per la manipolazione delle immagini e si mostrerà come combinare moduli standard ed esterni per costruire un semplice editor grafico.

Verifiche preliminari sull'ambiente Python

- **Controllare se Python è installato e la versione**

```
python --version          # oppure python3 --version
```

- **Percorso dell'eseguibile**

```
which python              # Linux / macOS
where python              # Windows (cmd)
```

- **Versione di `pip`**

```
pip --version
```

- **Installare un pacchetto da PyPI**

```
pip install pillow
```

- **Elencare i pacchetti installati**

```
pip list
```

- **Disinstallare un pacchetto**

```
pip uninstall pillow
```

- **Creare un ambiente virtuale**

```
# Creazione
python -m venv myenv

# Attivazione
#   Linux/macOS  : source myenv/bin/activate
#   Windows CMD  : myenv\Scripts\activate
#   Windows PowerShell: myenv\Scripts\Activate.ps1

# Disattivazione
deactivate
```

- **Installare o aggiornare Python**
  - Windows – installer ufficiale da `python.org`
  - macOS – `brew install python`
  - Linux – gestore pacchetti della distribuzione (`apt`, `dnf`, `pacman`...)
- **Rimuovere Python (solo se strettamente necessario)**
  - Windows – "Aggiungi/Rimuovi programmi"
  - Linux – `sudo apt remove python3`... (*attenzione*: molti strumenti di sistema lo richiedono)

Con questi comandi è possibile verificare lo stato dell'ambiente, installare in sicurezza i pacchetti necessari e isolarli in virtual env per evitare conflitti di versione.

10.1 Esercizio 1 - Editor di immagini

Crea un semplice editor di immagini che:

- Chieda il percorso di un file immagine (`.jpg` o `.png`)
- Verifichi che il file esista e che l'estensione sia accettata
- Mostri il nome, l'estensione e la risoluzione dell'immagine

- Chieda all'utente se vuole modificare la risoluzione (mantenendo opzionalmente le proporzioni)
- Offra la possibilità di convertire l'immagine in bianco e nero
- Salvi l'immagine modificata nella stessa cartella dell'originale, con suffisso `_new`

Analisi del problema

L'esercizio richiede di costruire una piccola pipeline di elaborazione immagini che coinvolge cinque fasi ben distinte:

1. **Input e validazione**
   – ricevere un percorso assoluto o relativo;
   – verificare l'esistenza del file e che l'estensione appartenga all'insieme {`jpg`, `png`}. In caso negativo ripetere la richiesta.
2. **Ispezione iniziale**
   – caricare l'immagine con `Pillow`;
   – estrarre nome, estensione e risoluzione;
   – visualizzare tali informazioni all'utente.
3. **Ridimensionamento opzionale**
   – domandare se ridimensionare;
   – in caso affermativo, scegliere se mantenere le proporzioni;
   – calcolare la nuova coppia larghezza × altezza e applicare `img.resize()`.
4. **Conversione in scala di grigi (opzionale)**
   – se richiesta, convertire l'immagine in modalità `"L"` (grayscale).
5. **Salvataggio e feedback**
   – scrivere il file nella stessa cartella, aggiungendo il suffisso `_new`;
   – stampare il riepilogo finale con percorso di destinazione e risoluzione risultante;
   – (ambiente notebook) visualizzare l'immagine prodotta per verifica.

Il flusso prevede perciò due punti di validazione (percorso ed estensione), due scelte opzionali dell'utente (ridimensionamento, conversione in B/N) e una fase conclusiva di I/O. La soluzione sfrutta `Pillow` per la parte grafica, `os` per la gestione dei percorsi e, se in Colab, `IPython.display` per l'anteprima dell'output.

Metodo 1 – Didattico e guidato

```
# Collego il mio Google Drive a Colab forzando il remount se
    necessario
```

```
from google.colab import drive
drive.mount('/content/drive', force_remount=True)
```

```
# Importazione moduli
import os
import sys
from PIL import Image
from IPython.display import Image as IPyImage, display

# -------- Sezione 1: Caricamento e verifica del file --------

# Ciclo che si ripete finché non viene fornito un percorso valido
# Qui usiamo "while True" per creare un ciclo teoricamente infinito.
# Il ciclo si interromperà solo quando raggiungiamo il comando "break", cioè quando tutte le condizioni sono verificate (file esistente, estensione corretta).
while True:
    path = input("Inserisci il percorso dell'immagine: ")

    # Verifico che il file esista
    if not os.path.isfile(path):
        print("Il percorso inserito non esiste! Riprova.\n")
        continue  # torna all'inizio del ciclo

    # Verifico che l'estensione sia corretta
    _, estensione = os.path.splitext(path)
    estensione = estensione[1:].lower()
    estensioni_corrette = ["jpg", "png"]

    if estensione not in estensioni_corrette:
        print("Il file non è un'immagine JPG o PNG! Riprova.\n")
        continue

    # Se tutto è valido, esco dal ciclo
    break

# ------- Sezione 2: Informazioni sull'immagine -----------

# Carico l'immagine e ottengo dimensioni
img = Image.open(path)
larghezza, altezza = img.size

# Estraggo il solo nome del file (senza percorso), ad esempio: "immagine.png"
```

```
# Poi divido il nome e l'estensione (es: "immagine" e ".png") e salvo solo il nome
nome_file = os.path.basename(path)
nome_senza_ext = os.path.splitext(nome_file)[0]

# Stampo le info iniziali
print(
    f"Immagine: {nome_file}\n"
    f"Estensione: {estensione}\n"
    f"Risoluzione: {larghezza}x{altezza}\n"
)

# -------- Sezione 3: Modifica della risoluzione ------------

r_input = input("Vuoi cambiare la risoluzione? (y/n): ").lower()
if r_input == "y":
    mantieni_proporzioni = input(
        "Vuoi mantenere le proporzioni originali? (y/n): ").lower()

    if mantieni_proporzioni == "y":
        rapporto = larghezza / altezza  # Rapporto larghezza/altezza dell'immagine originale

        # L'utente sceglie quale dimensione modificare
        scelta = input(
            "Vuoi modificare la larghezza o l'altezza? (l/a): ").lower()
        if scelta == "l":
            nuova_larghezza = int(input("Inserisci la nuova larghezza: "))
            nuova_altezza = round(nuova_larghezza / rapporto)
        elif scelta == "a":
            nuova_altezza = int(input("Inserisci la nuova altezza: "))
            nuova_larghezza = round(nuova_altezza * rapporto)
        else:
            print("Scelta non valida. Uso la risoluzione originale.")
            nuova_larghezza, nuova_altezza = larghezza, altezza

        print(f"Nuova risoluzione calcolata: {nuova_larghezza}x{nuova_altezza}")

    else:
        # Se NON vuole mantenere le proporzioni, chiedo entrambe le
```

```
    dimensioni nel formato "larghezzaxaltezza"
77          while True:
78              nuova_risoluzione = input(
79                  "Inserisci la nuova risoluzione (LARGHEZZAxALTEZZA):
     ")
80              try:
81                  nuova_larghezza, nuova_altezza = map(
82                      int, nuova_risoluzione.lower().split("x"))
83                  break  # Uscita dal ciclo solo se l'input è corretto
84              except:
85                  # Se l'input non è nel formato corretto, il ciclo
    ricomincia
86                  print(
87                      "Formato non valido! Deve essere nel formato"
88                      " LARGHEZZAxALTEZZA (es: 150x100)")
89  else:
90      # Se non vuole cambiare nulla, mantengo le dimensioni originali
91      nuova_larghezza, nuova_altezza = larghezza, altezza
92
93  # Ridimensiono l'immagine (operazione comune, fatta una sola volta
    alla fine)
94  img = img.resize((nuova_larghezza, nuova_altezza))
95
96  # ------------- Sezione 4: Bianco e nero ----------------
97
98  converti_bn = input(
99      "Vuoi convertire l'immagine in bianco e nero? (y/n): ").lower()
100 if converti_bn == "y":
101     img = img.convert("L")  # Modalità "L" = grayscale
102     bianco_e_nero = "Immagine convertita in bianco e nero"
103
104 # ------ Sezione 5: Salvataggio e visualizzazione --------
105
106 # Prendo la cartella di origine del file per salvare il nuovo nella
    stessa posizione
107 cartella_destinazione = os.path.dirname(path)
108
109 # Costruisco il percorso del nuovo file
110 percorso_output = os.path.join(
111     cartella_destinazione, f"{nome_senza_ext}_new.{estensione}"
112 )
113
114 # Salvo l'immagine modificata
115 img.save(percorso_output)
116
```

```
# Messaggio finale con info
colore = "bianco e nero" if converti_bn == "y" else "a colori"
print(f"L'immagine salvata ha risoluzione {nuova_larghezza}x{nuova_altezza}, "
      f"salvata in {colore} in: {percorso_output}")

# Visualizzo l'immagine dentro notebook
display(IPyImage(filename=percorso_output))
```

Output del programma

```
Inserisci il percorso dell'immagine: /content/drive/MyDrive/Colab Notebooks/Modulo 1: Programmazione con Python/Esercitazioni/Esercitazione_10_PyPI_e_PIP/Immagini/image.jpg
Immagine: image.jpg
Estensione: jpg
Risoluzione: 1200x817

Vuoi cambiare la risoluzione? (y/n): y
Vuoi mantenere le proporzioni originali? (y/n): y
Vuoi modificare la larghezza o l'altezza? (l/a): a
Inserisci la nuova altezza: 300
Nuova risoluzione calcolata: 441x300
Vuoi convertire l'immagine in bianco e nero? (y/n): y
L'immagine salvata ha risoluzione 441x300, salvata in bianco e nero in: /content/drive/MyDrive/Colab Notebooks/Modulo 1: Programmazione con Python/Esercitazioni/Esercitazione_10_PyPI_e_PIP/Immagini/image_new.jpg
```

Metodo 2 - Funzionale e compatto

```
# Versione alternativa dell'esercizio con uso di funzioni, assert, e struttura più compatta

from PIL import Image
from os.path import isfile
from sys import exit

# Funzione che verifica se il percorso contiene una delle estensioni accettate
# L'approccio funzionale rende il codice riutilizzabile, ad esempio per futuri controlli su altri file
```

```
9
10 def is_valid_img(img_path, accepted_ext=[".jpg", ".png"]):
11     for ext in accepted_ext:
12         if ext in img_path:
13             return True
14     return False
15
16 # Funzione che mantiene le proporzioni modificando solo una
   dimensione
17 # Qui si lavora direttamente sulla lista 'size', che viene
   aggiornata in base al rapporto
18 # Questo approccio evita il calcolo manuale in più punti del codice
19
20 def keep_aspect_ratio(size, original_ratio):
21     ratio = size[0] / size[1]  # calcola il rapporto della nuova
   immagine
22     if original_ratio > ratio:
23         size[0] = int(size[1] / original_ratio)  # adatta larghezza
24     else:
25         size[1] = int(size[0] * original_ratio)  # adatta altezza
26     return size
27
28 # Input del percorso immagine da tastiera
29 img_path = input("Inserisci il percorso all'immagine: ")
30
31 # Uso di assert: se la condizione è False, Python solleva
   automaticamente un'eccezione con il messaggio specificato. Questo
    sostituisce l'uso di if/else + sys.exit().
32 # È più compatto, ma meno esplicito per utenti non esperti.
33
34 assert isfile(img_path), f"{img_path} non esiste!"
35 assert is_valid_img(img_path), f"{img_path} non è un'immagine valida
   !"
36
37 # Tecnica per ottenere nome file, estensione e cartella:
38 # img_path.split("/")[-1] prende l'ultima parte del percorso, cioè
   il nome del file
39 # img_path.replace(img_name, "") rimuove il nome dal percorso,
   lasciando solo la cartella
40 # img_path.split(".")[-1] prende l'estensione come stringa (senza
   punto iniziale)
41
42 img_name = img_path.split("/")[-1]        # estrae solo il nome del
   file
43 img_folder = img_path.replace(img_name, "")  # ottiene la cartella
```

```
44 img_ext = img_path.split(".")[-1]          # estrae estensione (es. "
   jpg")
45
46 # Apertura dell'immagine
47 img = Image.open(img_path)
48
49 # Informazioni su nome, estensione, risoluzione
50 print("\n")
51 print(f"File: {img_name}")
52 print(f"Formato: {img_ext}")
53 print(f"Risoluzione: {img.width}x{img.height}")
54 print("\n")
55
56 # Input della nuova risoluzione in formato "LARGHEZZAxALTEZZA"
57 new_size = input("Nuova risoluzione: ")
58
59 # Se viene inserita una nuova risoluzione, la elaboro
60 if new_size != "":
61     new_width, new_height = list(map(int, new_size.split("x")))
62     original_ratio = img.width / img.height
63
64     # Se la proporzione cambia, chiedo se mantenerla
65     if original_ratio != new_width / new_height:
66         keep_ratio = input("Mantenere le proporzioni? [si/no default
   =no]: ")
67         if keep_ratio == "si":
68             # Uso della funzione per correggere le dimensioni
   mantenendo il rapporto
69             new_width, new_height = keep_aspect_ratio([new_width,
   new_height], original_ratio)
70             print(f"Nuova risoluzione {new_width}x{new_height}")
71
72     # Ridimensionamento con filtro anti-aliasing (migliora qualità)
73     img = img.resize((new_width, new_height), Image.ANTIALIAS)
74
75 # Conversione in bianco e nero, ma in modalità '1' (bianco o nero
   netti, senza sfumature)
76 bw_mode = input("Convertire in bianco e nero? [si/no default=no]: ")
77 if bw_mode == "si":
78     img = img.convert('1')
79
80 # Costruzione del nuovo percorso per il file modificato
81 # Qui si usa una concatenazione manuale piuttosto che os.path.join
82 new_img_path = img_folder + "/" + img_name.split(".")[0] + "_new." +
    img_ext
```

```
83 img.save(new_img_path)
84
85 print(f"\nNuova immagine salvata in {new_img_path}")
```

Librerie esterne utilizzate in questo capitolo

Libreria	Funzionalità principali / Metodi utilizzati	Tipo
`Pillow (PIL)`	`Image.open()` – apre un'immagine (metodo) `Image.resize()` – ridimensiona l'immagine (metodo) `Image.convert()` – converte in bianco e nero o altri modi (metodo) `Image.save()` – salva l'immagine modificata (metodo)	Libreria per immagini
`IPython.display`	`display()` e `Image()` per mostrare immagini in Colab o Jupyter	Utility Colab