🔷 Modulo 5: Collezioni

20. **Nested loops** 👓

Obiettivi di apprendimento:

- Comprendere la logica dei cicli `for` annidati.
- Creare pattern geometrici e matrici bidimensionali.
- Gestire il flusso di esecuzione (ciclo interno vs. ciclo esterno).
- Introdurre il concetto di complessità $O(n^2)$.

Contenuto teorico:

Un **nested loop** (ciclo annidato) è un ciclo che si trova all'interno del corpo di un altro ciclo. Il ciclo interno viene eseguito completamente per **ogni singola iterazione** del ciclo esterno. Sono essenziali per lavorare con strutture dati bidimensionali (come le matrici) e per generare pattern complessi.

- **Flusso di esecuzione:** Se il ciclo esterno esegue N iterazioni e il ciclo interno esegue M iterazioni, il codice all'interno del ciclo interno verrà eseguito un totale di $N \times M$ volte.
- **Tempo di esecuzione:** Questo porta a una complessità temporale di **$O(n^2)$** se N e M sono approssimativamente uguali.

```
def ciclo_annidato_base():
    righe = 3
    colonne = 4

    print("=== MATRICE 3x4 ===")

    # Il ciclo esterno gestisce le righe
    for riga in range(righe):

        # Il ciclo interno gestisce le colonne
        for colonna in range(colonne):
            # Stampa le coordinate e usa end="" per rimanere sulla stessa
riga
            print(f"({riga}, {colonna})", end=" ")

        # Stampa un newline dopo che il ciclo interno (la riga) è
terminato
        print()

ciclo_annidato_base()
```

Generatore di Pattern Avanzati (Triangolo rettangolo):

```
def pattern_triangolo(altezza):
    print(f"\n=== TRIANGOLO ALTEZZA {altezza} ===")

    # Il ciclo esterno definisce la riga (e il numero di * da stampare)
```

```
    for i in range(1, altezza + 1):

        # Il ciclo interno stampa i caratteri per la riga corrente
        for j in range(i):
            print("*", end="")

        # Passa alla riga successiva
        print()

pattern_triangolo(5)
```

Generatore di Pattern Inverso (Triangolo rettangolo capovolto):

```
def pattern_triangolo_inverso(altezza):
    print(f"\n=== TRIANGOLO INVERSO ALTEZZA {altezza} ===")

    # Il ciclo esterno definisce la riga, partendo dall'alto
    for i in range(altezza, 0, -1):

        # Stampa 'i' caratteri (5, 4, 3, 2, 1)
        for j in range(i):
            print("#", end="")

        print()

pattern_triangolo_inverso(5)
```

Esempio Pratico: Moltiplicazione a Griglia

```
def moltiplicazione_griglia(dimensione):
    print(f"\n=== GRIGLIA DI MOLTIPLICAZIONE {dimensione}x{dimensione}
===")

    # Stampa l'intestazione della colonna
    print("   |", end="")
    for i in range(1, dimensione + 1):
        print(f"{i:4}", end="")
    print()
    print("---" + "-" * (4 * dimensione))

    # Ciclo esterno (riga)
    for i in range(1, dimensione + 1):
        print(f"{i:2} |", end="") # Intestazione riga

        # Ciclo interno (colonna)
        for j in range(1, dimensione + 1):
            prodotto = i * j
            print(f"{prodotto:4}", end="") # Stampa il risultato
```

```
        print()

moltiplicazione_griglia(7)
```

---

21. **Lists, Sets, and Tuples** 🍎

Obiettivi di apprendimento:

- Distinguere tra i tre tipi principali di collezioni sequenziali in Python.
- Conoscere le proprietà di **mutabilità**, **ordinamento** e **unicità** per ciascuna.
- Eseguire operazioni comuni (aggiunta, rimozione, accesso) su ciascun tipo.

Contenuto teorico:

Python offre diverse strutture dati integrate per raggruppare dati. Le tre più comuni sono: **List** (Lista), **Tuple** (Tupla) e **Set** (Insieme).

Caratteristica	List (Lista)	Tuple (Tupla)	Set (Insieme)
Sintassi	`[1, "a", True]`	`(1, "a", True)`	`{1, "a", True}`
Mutabile?	**Sì** (modificabile)	**No** (immutabile)	**Sì** (aggiungi/rimuovi elementi)
Ordinata?	**Sì** (mantiene l'ordine di inserimento)	**Sì** (mantiene l'ordine di inserimento)	**No** (non ha un ordine definito)
Duplicati?	**Sì** (può contenere duplicati)	**Sì** (può contenere duplicati)	**No** (elementi unici)
Uso Tipico	Raccolta generica, stack, code.	Dati fissi, coordinate, record.	Appartenenza, eliminazione duplicati.

📌 **Lists (Liste)**

Le liste sono la struttura dati più flessibile. Sono mutabili e ordinate.

```
# Creazione
frutta = ["mela", "banana", "kiwi", "banana"]
print(f"Lista originale: {frutta}")

# Accesso
print(f"Primo elemento: {frutta[0]}") # mela

# Mutabilità
frutta[1] = "arancia"
print(f"Dopo modifica: {frutta}") # ['mela', 'arancia', 'kiwi', 'banana']

# Operazioni comuni
frutta.append("uva") # Aggiunge alla fine
```

```
frutta.insert(1, "pera") # Inserisce a un indice specifico
frutta.remove("mela") # Rimuove la prima occorrenza
elemento_rimosso = frutta.pop() # Rimuove e restituisce l'ultimo elemento

print(f"Lista finale: {frutta}")
```

📌 Tuples (Tuple)

Le tuple sono simili alle liste ma sono **immutabili**. Sono usate per dati che non devono cambiare.

```
# Creazione
coordinate = (10.5, 20.7)
colori_rgb = (255, 0, 0) # Rosso

print(f"\nTuple coordinate: {coordinate}")

# Accesso (funziona come le liste)
print(f"X: {coordinate[0]}")

# Immutabilità (genera errore)
# coordinate[0] = 5.0 # Tenterà di modificare la tupla e fallirà
(TypeError)
```

📌 Sets (Insiemi)

I set sono collezioni non ordinate di **elementi unici**. Sono efficienti per operazioni matematiche come unione e intersezione.

```
# Creazione
numeri = {1, 5, 3, 5, 2, 1}
print(f"\nSet numeri (nota l'unicità e l'ordine variabile): {numeri}") #
{1, 2, 3, 5}

# Aggiunta e rimozione
numeri.add(8)
numeri.remove(5)
print(f"Set aggiornato: {numeri}") # {1, 2, 3, 8}

# Operazioni sui Set
A = {1, 2, 3, 4}
B = {3, 4, 5, 6}

unione = A.union(B)
intersezione = A.intersection(B)
differenza = A.difference(B)

print(f"Unione (A ∪ B): {unione}") # {1, 2, 3, 4, 5, 6}
```

```
print(f"Intersezione (A ∩ B): {intersezione}") # {3, 4}
print(f"Differenza (A \ B): {differenza}") # {1, 2}
```

022_Shopping_Cart_Program

---

23. **2D Collections**

Obiettivi di apprendimento:

- Rappresentare dati bidimensionali (matrici) usando liste di liste.
- Accedere e modificare elementi in una matrice.
- Iterare su matrici con cicli annidati per la stampa e l'elaborazione.

Contenuto teorico:

Una **collezione 2D** (o matrice) è una lista in cui ogni elemento è esso stesso una lista. Questo permette di rappresentare dati in formato griglia o tabella, con righe e colonne.

- **Rappresentazione:** `matrice = [[riga0_col0, riga0_col1], [riga1_col0, riga1_col1]]`
- **Accesso:** Si usano due indici: `matrice[indice_riga][indice_colonna]`.

```
def matrici_base():
    # Rappresenta una mappa 3x3
    mappa = [
        ["X", "O", "X"], # Riga 0
        ["O", "O", "X"], # Riga 1
        ["X", "X", "O"]  # Riga 2
    ]

    print("=== MAPPA 2D ===")

    # Accesso
    print(f"Elemento (0, 1): {mappa[0][1]}") # O

    # Modifica (Le liste sono mutabili)
    mappa[1][1] = " " # Svuota la cella centrale
    print(f"Elemento (1, 1) modificato: {mappa[1][1]}")

    # Stampa la matrice usando cicli annidati
    print("\nStampa della mappa:")
    for riga in mappa:
        # Il ciclo esterno itera sulle sottoliste (righe)
        for elemento in riga:
            # Il ciclo interno itera sugli elementi della riga (colonne)
            print(f"{elemento}", end=" ")
        print() # Newline per la riga successiva

matrici_base()
```

```
# Creazione dinamica di una matrice (già visto con List Comprehensions Annidate)
righe, colonne = 4, 5
matrice_vuota = [[None for _ in range(colonne)] for _ in range(righe)]
print(f"\nMatrice 4x5 vuota:\n{matrice_vuota}")
```

024_Quiz_Game

---

25. **Dictionaries** 📙

Obiettivi di apprendimento:

- Comprendere la struttura di un dizionario: **coppie chiave-valore**.
- Conoscere le proprietà di **mutabilità** e **non ordinamento** (prima di Python 3.7).
- Eseguire operazioni comuni: accesso, aggiunta, modifica e rimozione di elementi.
- Iterare su chiavi, valori o entrambi (item).

Contenuto teorico:

I **Dizionari** sono la struttura dati fondamentale per le mappature in Python. Memorizzano i dati come coppie **chiave: valore**.

- **Chiavi:** Devono essere **uniche** e **immutabili** (stringhe, numeri o tuple).
- **Valori:** Possono essere di qualsiasi tipo e possono essere duplicati.
- **Mutabili:** I dizionari possono essere modificati dopo la creazione (aggiungere, cambiare, eliminare coppie).
- **Accesso:** L'accesso agli elementi avviene tramite la **chiave**, non tramite l'indice numerico.

```
# Creazione di un Dizionario
studente = {
    "nome": "Luca Rossi",
    "eta": 22,
    "matricola": "M1001",
    "citta": "Roma",
    "voti": [8, 7.5, 9]
}

print("=== DIZIONARI BASE ===")
print(f"Dizionario completo: {studente}")

# Accesso ai valori (usando la chiave)
print(f"Nome studente: {studente['nome']}")
print(f"Voti: {studente['voti']}")

# Aggiunta di una nuova coppia chiave-valore
studente["corso"] = "Ingegneria"
print(f"Aggiunta 'corso': {studente['corso']}")
```

```
# Modifica di un valore esistente
studente["eta"] = 23
print(f"Età aggiornata: {studente['eta']}")

# Rimozione di una coppia chiave-valore
del studente["citta"]
# studente.pop("citta") # Alternativa: pop() rimuove e restituisce il
valore

# Iterazione su un Dizionario
print("\n=== ITERAZIONE ===")
print("Chiavi:")
for chiave in studente.keys():
    print(f"- {chiave}")

print("\nValori:")
for valore in studente.values():
    print(f"- {valore}")

print("\nCoppie Chiave-Valore (Item):")
for chiave, valore in studente.items():
    print(f"{chiave:<10}: {valore}")

# Metodo .get() per accesso sicuro (evita KeyError)
print(f"Telefono (get): {studente.get('telefono', 'N/A')}")
```

026_Concession_Stand_Program

---

27. **Random Numbers** 🎲

Obiettivi di apprendimento:

- Importare il modulo `random`.
- Generare numeri interi casuali in un intervallo (`randint`).
- Generare numeri float casuali tra 0 e 1 (`random`).
- Scegliere un elemento casuale da una sequenza (`choice`).

Contenuto teorico:

Il modulo **`random`** è essenziale per qualsiasi operazione che richieda la casualità, dai giochi ai test statistici.

```
import random

print("=== MODULO RANDOM ===")

# 1. random.randint(a, b) -> Genera un intero casuale N tale che a <= N <=
b (inclusivo)
numero_casuale_inclusivo = random.randint(1, 10)
```

```
print(f"Intero tra 1 e 10 (inclusivo): {numero_casuale_inclusivo}")

# 2. random.random() -> Genera un float casuale tra 0.0 (incluso) e 1.0
(escluso)
float_casuale = random.random()
print(f"Float tra 0.0 e 1.0: {float_casuale:.4f}")

# 3. random.uniform(a, b) -> Genera un float casuale tra a e b
(inclusivo/esclusivo a seconda dell'implementazione)
float_personalizzato = random.uniform(10.0, 20.0)
print(f"Float tra 10.0 e 20.0: {float_personalizzato:.4f}")

# 4. random.choice(sequenza) -> Sceglie un elemento casuale da una Lista,
Tupla o Stringa
frutti = ["mela", "banana", "kiwi", "arancia"]
scelta_casuale = random.choice(frutti)
print(f"Frutto scelto casualmente: {scelta_casuale}")

# 5. random.shuffle(lista) -> Mescola gli elementi di una lista (in-place)
carte = ["A", "K", "Q", "J", "10"]
random.shuffle(carte)
print(f"Carte mescolate: {carte}")
```

028_Number_Guessing_Game

029_Rock_Paper_Scissors_Game

030_Dice_Roller_Program