Week-8 programs

In [2]:

```
class BankAccount:
    def __init__(self, acc, name, bal, pwd):
        self.__acc, self.__name, self.__bal, self.__pwd = acc, name, bal, pwd

    def get_account_number(self): return self.__acc
    def get_owner_name(self): return self.__name
    def get_balance(self): return self.__bal

    def deposit(self, amt):
        if amt <= 0: raise ValueError("Invalid deposit")
        self.__bal += amt
        print(f"Deposited {amt}, Balance: {self.__bal}")

    def withdraw(self, amt, pwd):
        if pwd != self.__pwd: raise PermissionError("Wrong password")
        if amt <= 0 or amt > self.__bal: raise ValueError("Invalid withdrawal")
        self.__bal -= amt
        print(f"Withdrew {amt}, Balance: {self.__bal}")

    def transfer(self, to, amt, pwd):
        if pwd != self.__pwd: raise PermissionError("Wrong password")
        if amt <= 0 or amt > self.__bal: raise ValueError("Invalid transfer")
        self.__bal -= amt
        to._BankAccount__bal += amt
        print(f"Transferred {amt} to {to.get_owner_name()}, Balance: {self.__bal}")

    def display(self):
        print(f"Acc: {self.__acc}, Owner: {self.__name}, Balance: {self.__bal}")


class SavingsAccount(BankAccount):
    def withdraw(self, amt, pwd):
        super().withdraw(amt, pwd)
        if self.get_balance() < 1000:
            self._BankAccount__bal -= 100
            print("Penalty 100 applied")


class CurrentAccount(BankAccount):
    def withdraw(self, amt, pwd):
        if pwd != self._BankAccount__pwd: raise PermissionError("Wrong password")
        if amt <= 0 or amt > self.get_balance() + 5000: raise ValueError("Overdraft ex
        self._BankAccount__bal -= amt
        print(f"Withdrew {amt}, Balance: {self.get_balance()}")


def main():
    accs = {}
    while True:
        print("\n1.Create Savings  2.Create Current  3.Deposit  4.Withdraw  5.Transfer
        c = input("Choice: ")
        try:
            if c == "1":
                a, n, b, p = input("Acc: "), input("Name: "), float(input("Bal: ")), i
```

```
                accs[a] = SavingsAccount(a, n, b, p)
            elif c == "2":
                a, n, b, p = input("Acc: "), input("Name: "), float(input("Bal: ")), i
                accs[a] = CurrentAccount(a, n, b, p)
            elif c == "3":
                a, amt = input("Acc: "), float(input("Amt: "))
                accs[a].deposit(amt)
            elif c == "4":
                a, amt, p = input("Acc: "), float(input("Amt: ")), input("Pwd: ")
                accs[a].withdraw(amt, p)
            elif c == "5":
                f, t, amt, p = input("From: "), input("To: "), float(input("Amt: ")),
                accs[f].transfer(accs[t], amt, p)
            elif c == "6":
                a = input("Acc: ")
                accs[a].display()
            elif c == "7":
                break
        except Exception as e:
            print("Error:", e)


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

```
1.Create Savings  2.Create Current  3.Deposit  4.Withdraw  5.Transfer  6.Display  7.E
xit
Choice: 1
Acc: 1
Name: kamal
Bal: 10000
Pwd: 1

1.Create Savings  2.Create Current  3.Deposit  4.Withdraw  5.Transfer  6.Display  7.E
xit
Choice: 4
Acc: 1
Amt: 100
Pwd: 1
Withdrew 100.0, Balance: 9900.0

1.Create Savings  2.Create Current  3.Deposit  4.Withdraw  5.Transfer  6.Display  7.E
xit
Choice: 2
Acc: 2
Name: sanjay
Bal: 99999
Pwd: 2

1.Create Savings  2.Create Current  3.Deposit  4.Withdraw  5.Transfer  6.Display  7.E
xit
Choice: 3
Acc: 2
Amt: 1
Deposited 1.0, Balance: 100000.0

1.Create Savings  2.Create Current  3.Deposit  4.Withdraw  5.Transfer  6.Display  7.E
xit
Choice: 4
Acc: 1
Amt: 1
Pwd: 1
Withdrew 1.0, Balance: 9899.0

1.Create Savings  2.Create Current  3.Deposit  4.Withdraw  5.Transfer  6.Display  7.E
xit
Choice: 5
From: 2
To: 1
Amt: 100
Pwd: 2
Transferred 100.0 to kamal, Balance: 99900.0

1.Create Savings  2.Create Current  3.Deposit  4.Withdraw  5.Transfer  6.Display  7.E
xit
Choice: 6
Acc: 2
Acc: 2, Owner: sanjay, Balance: 99900.0

1.Create Savings  2.Create Current  3.Deposit  4.Withdraw  5.Transfer  6.Display  7.E
xit
Choice: 7
```

In [3]:
```
from abc import ABC, abstractmethod
import math
```

```python
class Shape(ABC):
    @abstractmethod
    def area(self): pass

    @abstractmethod
    def perimeter(self): pass


class SolidShape(Shape):
    @abstractmethod
    def volume(self): pass


class Rectangle(Shape):
    def __init__(self, length, width):
        if length <= 0 or width <= 0: raise ValueError("Invalid dimensions")
        self.length, self.width = length, width

    def area(self): return self.length * self.width
    def perimeter(self): return 2 * (self.length + self.width)


class Circle(Shape):
    def __init__(self, radius):
        if radius <= 0: raise ValueError("Invalid radius")
        self.radius = radius

    def area(self): return math.pi * self.radius ** 2
    def perimeter(self): return 2 * math.pi * self.radius


class Cylinder(SolidShape):
    def __init__(self, radius, height):
        if radius <= 0 or height <= 0: raise ValueError("Invalid dimensions")
        self.radius, self.height = radius, height

    def area(self): return 2 * math.pi * self.radius * (self.radius + self.height)
    def perimeter(self): return 2 * math.pi * self.radius
    def volume(self): return math.pi * self.radius ** 2 * self.height


class Cube(SolidShape):
    def __init__(self, side):
        if side <= 0: raise ValueError("Invalid side length")
        self.side = side

    def area(self): return 6 * self.side ** 2
    def perimeter(self): return 12 * self.side
    def volume(self): return self.side ** 3


def display_shape_info(shape):
    print(f"\nShape: {shape.__class__.__name__}")
    print("Area:", shape.area())
    print("Perimeter:", shape.perimeter())
    if isinstance(shape, SolidShape):
        print("Volume:", shape.volume())
```

```python
def main():
    shapes = []
    while True:
        print("\n1.Rectangle  2.Circle  3.Cylinder  4.Cube  5.Display Shapes  6.Compar
        c = input("Choice: ")
        try:
            if c == "1":
                l, w = float(input("Length: ")), float(input("Width: "))
                shapes.append(Rectangle(l, w))
            elif c == "2":
                r = float(input("Radius: "))
                shapes.append(Circle(r))
            elif c == "3":
                r, h = float(input("Radius: ")), float(input("Height: "))
                shapes.append(Cylinder(r, h))
            elif c == "4":
                s = float(input("Side: "))
                shapes.append(Cube(s))
            elif c == "5":
                for s in shapes: display_shape_info(s)
            elif c == "6":
                if len(shapes) < 2: print("Need at least 2 shapes")
                else:
                    areas = [(s.area(), s.__class__.__name__) for s in shapes]
                    areas.sort(reverse=True)
                    print("Areas (largest first):")
                    for a, n in areas: print(f"{n}: {a}")
            elif c == "7":
                break
        except Exception as e:
            print("Error:", e)


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

```
1.Rectangle  2.Circle  3.Cylinder  4.Cube  5.Display Shapes  6.Compare Areas  7.Exit
Choice: 1
Length: 1000
Width: 111

1.Rectangle  2.Circle  3.Cylinder  4.Cube  5.Display Shapes  6.Compare Areas  7.Exit
Choice: 2
Radius: 90

1.Rectangle  2.Circle  3.Cylinder  4.Cube  5.Display Shapes  6.Compare Areas  7.Exit
Choice: 3
Radius: 90
Height: 1

1.Rectangle  2.Circle  3.Cylinder  4.Cube  5.Display Shapes  6.Compare Areas  7.Exit
Choice: 4
Side: 44

1.Rectangle  2.Circle  3.Cylinder  4.Cube  5.Display Shapes  6.Compare Areas  7.Exit
Choice: 5

Shape: Rectangle
Area: 111000.0
Perimeter: 2222.0

Shape: Circle
Area: 25446.900494077323
Perimeter: 565.4866776461628

Shape: Cylinder
Area: 51459.28766580081
Perimeter: 565.4866776461628
Volume: 25446.900494077323

Shape: Cube
Area: 11616.0
Perimeter: 528.0
Volume: 85184.0

1.Rectangle  2.Circle  3.Cylinder  4.Cube  5.Display Shapes  6.Compare Areas  7.Exit
Choice: 6
Areas (largest first):
Rectangle: 111000.0
Cylinder: 51459.28766580081
Circle: 25446.900494077323
Cube: 11616.0

1.Rectangle  2.Circle  3.Cylinder  4.Cube  5.Display Shapes  6.Compare Areas  7.Exit
Choice: 7
```

In [ ]:

WEEK 9

Movies CSV Processing

Question:
Given a `movies.csv` file with columns: `Title, Year, Genre, Rating`, clean the data by:

- Removing leading/trailing spaces in all fields
- Splitting multiple genres (e.g., Action|Sci-Fi) into separate columns (`Genre1, Genre2, …`)
- Ensuring `Year` and `Rating` are numeric types

Save the cleaned dataset to `movies_clean.csv`. Generate a summary text file `genre_stats.txt` with the average rating per genre.

In [6]:
```
import pandas as pd

df = pd.read_csv("movies.csv")
df = df.applymap(lambda x: x.strip() if isinstance(x, str) else x)
genres_split = df["Genre"].str.split("|", expand=True)
genres_split.columns = [f"Genre{i+1}" for i in range(genres_split.shape[1])]
df = pd.concat([df.drop(columns=["Genre"]), genres_split], axis=1)
df["Year"] = pd.to_numeric(df["Year"])
df["Rating"] = pd.to_numeric(df["Rating"])
df.to_csv("movies_clean.csv", index=False)

print("\nmovies_clean.csv content:")
print(df)

genres = pd.melt(df, id_vars=["Title", "Year", "Rating"], value_vars=genres_split.colu
genres = genres.dropna(subset=["Genre"])
genre_stats = genres.groupby("Genre")["Rating"].mean().reset_index()

with open("genre_stats.txt", "w") as f:
    for _, row in genre_stats.iterrows():
        f.write(f"{row['Genre']} : {row['Rating']:.2f}\n")

print("\ngenre_stats.txt content:")
with open("genre_stats.txt", "r") as f:
    print(f.read())
```

```
movies_clean.csv content:
        Title  Year  Rating  Genre1   Genre2
0  The Matrix  1999     8.7  Action   Sci-Fi
1   Inception  2010     8.8  Sci-Fi   Action
2  Fight Club  1999     8.8   Drama     None

genre_stats.txt content:
 Action : 8.80
Action : 8.70
Drama : 8.80
Sci-Fi : 8.75
```

Students CSV Processing

Question:
Given a `students.csv` file with columns: `Name, Subject, Marks`, perform the following:

- Convert student names into Title Case (e.g., alice → Alice)
- Group marks by subject and calculate:
  - Average marks per subject
  - Highest scorer in each subject

Write the results into `summary.csv` in the format:
`Subject, Average, Topper, TopperMarks`

In [7]:
```
import pandas as pd

df = pd.read_csv("students.csv")
df["Name"] = df["Name"].str.title()
avg_marks = df.groupby("Subject")["Marks"].mean().reset_index(name="Average")
topper = df.loc[df.groupby("Subject")["Marks"].idxmax()]
summary = avg_marks.merge(topper, on="Subject")
summary = summary.rename(columns={"Name": "Topper", "Marks": "TopperMarks"})
summary.to_csv("summary.csv", index=False)

print("\nsummary.csv content:")
print(summary)
```

```
summary.csv content:
   Subject  Average   Topper  TopperMarks
0     Math     91.5  Charlie           95
1  Science     72.0      Bob           72
```

Article Text Processing

Question:
Given a `.txt` file containing an article or book chapter, perform the following:

- Remove punctuation
- Convert all text to lowercase
- Tokenize the text into words
- Count the frequency of each word and store it in a dictionary

Write the top 20 most frequent words to `output.txt` in the format:
`word, frequency`

In [8]:
```
import string
from collections import Counter

with open("article.txt", "r") as f:
    text = f.read()

text = text.lower().translate(str.maketrans("", "", string.punctuation))
words = text.split()
freq = Counter(words)
top20 = freq.most_common(20)

with open("output.txt", "w") as f:
    for word, count in top20:
        f.write(f"{word},{count}\n")
```

```
print("\noutput.txt content:")
with open("output.txt", "r") as f:
    print(f.read())
```

```
output.txt content:
data,4
science,2
is,2
the,2
future,2
of,1
bright,1
everywhere,1
but,1
not,1
enough,1
insight,1
```

WEEK 10

Design a Python application to simulate a banking system involving two shared bank accounts, Account A (starting balance: $1000) and Account B ($starting balance$:500). Write a function, transfer_money(sender_account, receiver_account, amount), and launch five threads that concurrently attempt to call this function with random amounts between $1 and 100.

1. Implement Necessary Synchronization: Use two separate threading.Lock objects (one for each account) to protect the integrity of the individual account balances.
2. Demonstrate Deadlock (Optional): Show how a deadlock can occur if the two locks are acquired in different orders by different threads (e.g., Thread 1 acquires Lock A then Lock B, while Thread 2 acquires Lock B then Lock A). The final combined balance (A+B) must always equal the initial combined balance ($1500), regardless of the transfer order.

In [104…

```
import threading
import random
import time

account_A = {"name": "Account A", "balance": 1000}
account_B = {"name": "Account B", "balance": 500}

lock_A = threading.Lock()
lock_B = threading.Lock()

def transfer_money(sender, receiver, amount):
    # Always lock accounts in a fixed order to prevent deadlock
    first_lock, second_lock = (lock_A, lock_B) if id(lock_A) < id(lock_B) else (lock_B
    with first_lock:
        with second_lock:
            if sender["balance"] >= amount:
                sender["balance"] -= amount
                receiver["balance"] += amount

def simulate_transfer():
```

```
    for _ in range(50):
        amount = random.randint(1, 100)
        if random.choice([True, False]):
            transfer_money(account_A, account_B, amount)
        else:
            transfer_money(account_B, account_A, amount)

threads = [threading.Thread(target=simulate_transfer, name=f"T{i+1}") for i in range(5

print("Starting Transfers...")
start = time.time()
for t in threads:
    t.start()
for t in threads:
    t.join()
end = time.time()

print("\n--- Final Results ---")
print(f"Account A Balance: ${account_A['balance']}")
print(f"Account B Balance: ${account_B['balance']}")
print(f"Total Combined: ${account_A['balance'] + account_B['balance']}")
print(f"Expected Total: $1500")
print(f"Time Taken: {end - start:.4f} sec")
```

```
Starting Transfers...

--- Final Results ---
Account A Balance: $947
Account B Balance: $553
Total Combined: $1500
Expected Total: $1500
Time Taken: 0.0023 sec
```

A company wants to analyze large text files. You are asked to design a Python program that: Splits a given text file into five equal parts. Creates five threads, where each thread:

a. Reads one part of the file,

b. Counts the number of words in its chunk,

c. Stores the count in a shared list.

After all threads complete, the main thread should sum up the word counts and display the total number of words in the file.

In [105…

```
import threading
import time

sample_text = ("Python multithreading is powerful and efficient. " * 1000).strip()
with open("sample_text.txt", "w") as f:
    f.write(sample_text)

word_counts = []
lock = threading.Lock()

def count_words_in_chunk(chunk):
    count = len(chunk.split())
    with lock:
```

```
        word_counts.append(count)

def threaded_word_count(filename, num_threads=5):
    with open(filename, "r") as f:
        text = f.read()
    size = len(text)
    chunk_size = size // num_threads
    threads = []
    start_time = time.time()
    for i in range(num_threads):
        start_idx = i * chunk_size
        end_idx = None if i == num_threads - 1 else (i + 1) * chunk_size
        chunk = text[start_idx:end_idx]
        t = threading.Thread(target=count_words_in_chunk, args=(chunk,))
        threads.append(t)
        t.start()
    for t in threads:
        t.join()
    total_words = sum(word_counts)
    print(f"Total word count: {total_words}")
    print(f"Processed in: {time.time() - start_time:.4f} seconds")

threaded_word_count("sample_text.txt")
```

```
Total word count: 6003
Processed in: 0.0008 seconds
```

Design a Python program to perform matrix multiplication using multithreading. You are given two square matrices A and B of size N x N. Create N threads, where each thread computes one row of the result matrix C = A × B. Use the threading module to launch the threads and join them after completion. Store the results in a shared 2D list (matrix C). Ensure synchronization so that no race condition occurs when multiple threads update the shared matrix. Compare the performance of the threaded version with a sequential version for large matrices.

In [106…

```
import threading
import random
import time

N = 4
A = [[random.randint(1, 10) for _ in range(N)] for _ in range(N)]
B = [[random.randint(1, 10) for _ in range(N)] for _ in range(N)]
C = [[0 for _ in range(N)] for _ in range(N)]

lock = threading.Lock()

def multiply_row(row_idx):
    global C
    for j in range(N):
        value = sum(A[row_idx][k] * B[k][j] for k in range(N))
        with lock:
            C[row_idx][j] = value

def threaded_matrix_multiply():
    threads = []
    start = time.time()
    for i in range(N):
        t = threading.Thread(target=multiply_row, args=(i,))
        threads.append(t)
```

```
        t.start()
    for t in threads:
        t.join()
    end = time.time()
    print(f"Threaded multiplication took: {end - start:.6f} sec")

def sequential_matrix_multiply():
    start = time.time()
    result = [[sum(A[i][k] * B[k][j] for k in range(N)) for j in range(N)] for i in ra
    end = time.time()
    print(f"Sequential multiplication took: {end - start:.6f} sec")
    return result

threaded_matrix_multiply()
seq_result = sequential_matrix_multiply()

print("\nMatrix A:", A)
print("Matrix B:", B)
print("\nResult (Threaded):", C)
print("Result (Sequential):", seq_result)
```

```
Threaded multiplication took: 0.000817 sec
Sequential multiplication took: 0.000041 sec

Matrix A: [[10, 2, 10, 10], [8, 9, 10, 7], [2, 4, 5, 5], [4, 9, 1, 10]]
Matrix B: [[6, 10, 4, 9], [2, 8, 8, 2], [6, 3, 5, 3], [10, 3, 10, 10]]

Result (Threaded): [[224, 176, 206, 224], [196, 203, 224, 190], [100, 82, 115, 91],
[148, 145, 193, 157]]
Result (Sequential): [[224, 176, 206, 224], [196, 203, 224, 190], [100, 82, 115, 91],
[148, 145, 193, 157]]
```