**Q1. What is multithreading in Python? Why is it used? Name the module used to handle threads in Python.**

**Multithreading** is a technique in Python where multiple threads run concurrently within a single process. Each thread is a separate flow of execution, allowing tasks to run seemingly at the same time.

**Why is it used?**

* To achieve concurrency for tasks that involve waiting, such as I/O-bound tasks (file operations, web scraping, network requests).
* To improve the performance of programs by efficiently utilizing CPU resources.
* To keep the main program responsive during time-consuming tasks.

**Module used to handle threads:**  
The threading module in Python is used for creating and managing threads.

**Q2. Why is the threading module used?**

The threading module is used to create and manage threads easily in Python. It provides methods and classes to perform concurrent programming, such as creating threads, synchronizing threads, and managing thread lifecycle.

**Functions:**

1. **activeCount()**
   * Returns the number of thread objects currently alive.
   * Example:
   * import threading
   * print(threading.activeCount())
2. **currentThread()**
   * Returns the current thread object.
   * Example:
   * import threading
   * print(threading.currentThread().name)
3. **enumerate()**
   * Returns a list of all currently active threads.
   * Example:
   * import threading
   * print(threading.enumerate())

**Q3. Explanation of functions:**

1. **run()**
   * Contains the code that the thread should execute. This method is called when the thread starts.
   * Example:
   * class MyThread(threading.Thread):
   * def run(self):
   * print("Thread is running")
2. **start()**
   * Starts a thread by calling its run() method.
   * Example:
   * t = MyThread()
   * t.start()
3. **join()**
   * Waits for the thread to complete execution.
   * Example:
   * t.join()
4. **isAlive()**
   * Checks whether a thread is still running.
   * Example:
   * print(t.isAlive())

**Q4. Python program to create two threads:**

import threading

def print\_squares(numbers):

print("Squares:")

for num in numbers:

print(num \*\* 2)

def print\_cubes(numbers):

print("Cubes:")

for num in numbers:

print(num \*\* 3)

numbers = [1, 2, 3, 4, 5]

# Creating threads

thread1 = threading.Thread(target=print\_squares, args=(numbers,))

thread2 = threading.Thread(target=print\_cubes, args=(numbers,))

# Starting threads

thread1.start()

thread2.start()

# Joining threads

thread1.join()

thread2.join()

print("Both threads have finished execution.")

**Q5. Advantages and Disadvantages of Multithreading:**

**Advantages:**

1. Improves performance for I/O-bound tasks.
2. Enhances responsiveness of applications.
3. Efficient utilization of CPU by running tasks concurrently.
4. Simplifies the design of applications that need to handle multiple tasks simultaneously.

**Disadvantages:**

1. Python's Global Interpreter Lock (GIL) limits true parallelism in CPU-bound tasks.
2. Can lead to race conditions if shared data is not handled properly.
3. Debugging multithreaded programs is complex.
4. Increased memory usage and overhead for context switching.

**Q6. Deadlocks and Race Conditions:**

1. **Deadlocks:**
   * Occur when two or more threads are waiting for each other to release resources, causing them to be stuck indefinitely.
   * Example: Thread A locks Resource 1 and waits for Resource 2, while Thread B locks Resource 2 and waits for Resource 1.
2. **Race Conditions:**
   * Occur when two or more threads access shared data concurrently, and the outcome depends on the order of execution.
   * Example: Incrementing a shared counter without synchronization can lead to unexpected results.

**Solutions:**

* Use locks, semaphores, or thread-safe data structures to prevent race conditions.
* Use proper locking mechanisms to avoid deadlocks (e.g., lock ordering).