Q1

What is multithreading in python? hy is it used? Name the module used to handle threads in python

This article covers the basics of multithreading in Python programming language. Just like multiprocessing, multithreading is a way of achieving multitasking. In multithreading, the concept of threads is used. "Threading" module is used to handle threads

Q2

Why threading module used? write the use of the following functions

1. activeCount()
2. currentThread()
3. enumerate()

Python threading allows you to have different parts of your program run concurrently and can simplify your design.

1. The threading.active_count() is an inbuilt method of the threading module, it is used to return the number of Thread objects that are active at any instant.
2. threading.currentThread() − Returns the number of thread objects in the caller's thread control
3. The threading.enumerate() is an inbuilt method of the threading module, it is used to return the list of all the Thread class objects which are currently alive. It also includes daemonic threads, the main thread, and dummy thread objects created by current_thread().

Q3

3. Explain the following functions
4. run()
5. start()
6. join()
7. isAlive()

run() : is a method that is often defined or overridden in a class that implements the Runnable interface or inherits from the Thread class in threading libraries. It represents the code that will be executed when the thread is started. When you start a thread, the run() method is invoked, and the logic within it is executed in the context of that thread.

start(): is a method that starts the execution of a thread. When you call start(), it creates a new thread and invokes the run() method in that new thread. This is the recommended way to start a thread rather than directly calling the run() method, as calling run() directly won't create a new thread and will run the code in the current thread.

join(): is a method used to wait for a thread to complete its execution. When you call join() on a thread, the calling thread will be blocked until the target thread finishes its work. It is useful when you want to ensure that certain operations in your program are executed only after a particular thread has finished its task.

isAlive(): is a method used to check if a thread is currently running. It returns a boolean value

Q4

Write a python program to create two threads. Thread one must print the list of squares and thread two must print the list of cubes

In [34]:
```
import threading

def sq(numbers):
    for num in numbers:
        print(f"the square of {num}:{num**2}")
def cb(numbers):
    for num in numbers:
        print(f"the cube of {num}:{num**3}")
def main():
    numbers=list(range(5))

    thrd1= threading.Thread(target=sq, args=(numbers,))
    thrd2= threading.Thread(target=cb, args=(numbers,))
    thrd1.start()
    thrd2.start()
    thrd1.join()
    thrd2.join()

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

```
the square of 0:0the cube of 0:0
the cube of 1:1
the cube of 2:8
the cube of 3:27
the cube of 4:64

the square of 1:1
the square of 2:4
the square of 3:9
the square of 4:16
```

In [ ]:
```
```

Advantages:

improved Performance: Multithreading allows a program to execute multiple tasks concurrently, taking advantage of multiple processor cores. This can lead to improved performance and faster execution times, especially in CPU-bound tasks.

Responsiveness: Multithreading can enhance the responsiveness of an application, particularly in user interfaces. By offloading time-consuming tasks to separate threads, the main thread can remain responsive and continue handling user interactions.

Resource Sharing: Threads within a process share the same memory space, making it easier to share data between threads without the need for complex inter-process communication mechanisms.

Modular Design: Multithreading allows developers to design applications with modular components that can run independently as separate threads, making the codebase more maintainable and easier to understand.

Parallelism: Multithreading enables parallelism, which is crucial in tasks that can be broken down into smaller, independent subtasks. This can significantly speed up the execution of such tasks.

Disadvantages:

Complexity: Writing multithreaded code can be complex and error-prone due to issues like race conditions, deadlocks, and thread synchronization. Debugging and testing multithreaded programs can also be more challenging.

Race Conditions: Race conditions occur when multiple threads access shared resources concurrently, leading to unpredictable behavior and data corruption if not handled correctly.

Deadlocks: Deadlocks can occur when two or more threads are waiting for each other to release resources, resulting in a state where none of the threads can proceed, causing the program to freeze.

Resource Contentions: Threads competing for shared resources can lead to resource contentions, slowing down the overall performance of the application.

Increased Memory Usage: Each thread has its own stack and thread context, which can lead to increased memory consumption, especially when creating numerous threads.

Difficulty in Debugging: Identifying and resolving issues in multithreaded applications can be more complex than single-threaded programs, as debugging becomes more challenging with non-deterministic behavior.

Limited Scaling: While multithreading can improve performance on multi-core processors, there is a limit to how much performance can be gained from adding more threads due to diminishing returns and overhead from thread management.

Q6

Deadlocks: A deadlock is a situation in which two or more threads or processes are unable to proceed because each is waiting for the other to release a resource that they need. In other words, the threads are stuck in a circular wait, where each thread is holding a resource that another thread requires to proceed. As a result, none of the threads can make any progress, and the application can become unresponsive or even crash.

Race Conditions: A race condition is a situation in which the behavior of a program depends on the relative timing of events, particularly when multiple threads or processes access shared resources without proper synchronization. It occurs when the outcome of the program depends on the order of execution of operations, which can lead to unpredictable or erroneous results.

In [ ]: