MIT World Peace University

Python Programming
Second Year B. Tech, Semester 4

Learning Basics of the *threading module*

Assignment No. 10

Prepared By

Krishnaraj Thadesar
Cyber Security and Forensics
Batch A1, PA 20

April 30, 2023

Contents

1 Aim

Write a Python program to implement multithreading scenarios

2 Problem Statement

Create two threads to display cube and square of 5 numbers from list. Threads can simultaneously execute a Cube and square functions from program to access the shared list of 5 numbers.

3 Objectives

1. To learn and implement functions of the threading library.

4 Theory

4.1 Threads

A Thread is a single sequential flow of control within a program. The concept of threads is similar to that of processes.

- Threads are sometimes called lightweight processes because they have their own stack but can access shared data. Because threads share the same address space as the process and other threads within the process, the operational cost of communication between the threads is low, which is an advantage. The disadvantage is that a problem with one thread in a process will certainly affect other threads and the viability of the process itself.
- Threads allow the application to be more responsive to the user. Threads can also take advantage of multiprocessor architectures as processes can be assigned to different processors. The operating system manages the threads within a process. Also see multitasking, multiprocessing, multithreading.
- A thread is a dispatchable unit of work. Threads are sometimes called lightweight processes. Both processes and threads provide an execution environment, but creating a new thread requires fewer resources than creating a new process. Threads exist within a process — every process has at least one. Threads share the process's resources, including memory and open files. This makes for efficient, but potentially problematic, communication.

Example:

```
t1 = threading.Thread(target=print_cube, args=(10,))
t2 = threading.Thread(target=print_square, args=(10,))
t1.start()
t2.start()
```

4.2 Process

A Process is a collection of one or more threads that share the same virtual memory, code segment, data segment, and operating-system resources, such as open files and signals.

-

- A process is controlled by the operating system, which allocates resources and regulates execution. Threads are not controlled by the operating system, but rather are driven by plain Java code, with some help from the JVM.
- A process is an execution of a program, but a thread is a single execution sequence within a process. A process can contain multiple threads. A thread is sometimes called a lightweight process.
- A process is an instance of a program running in a computer. It is close in meaning to task , a term used in some operating systems. In UNIX and some other operating systems, a process is started when a program is initiated (either by a user entering a shell command or by another program). Like a task, a process is a running program with which a particular set of data is associated so that the process can be kept track of. Each process can be assigned a certain priority. Because the operating system must manage the allocation of resources to processes, it must also keep track of where each process resides in memory and where and how much memory is available for each process. This activity is called process management.

4.3 Multithreading

Multithreading is the ability of a program or an operating system process to manage its use by more than one user at a time and to even manage multiple requests by the same user without having to have multiple copies of the programming running in the computer.

- It is a process of executing multiple threads simultaneously.
- It enables a program to perform more than one task at a time, by using the maximum CPU time.
- The main benefit of multithreading is that it enables a program to operate more efficiently by doing multiple things at the same time, rather than one at a time.
- Under the hood, most operating systems use preemptive multitasking, which means each running program is allowed to run for a short time before another program gets a turn. This creates the illusion that multiple programs are running simultaneously, even though only one program is running at a time.

4.4 Threading Module

The threading module builds on the low-level features of thread to make working with threads even easier and more pythonic. It offers a higher-level interface than the thread module: threading.Thread is a class that represents a thread of control.

- The threading module provides a number of functions and classes to manage threads.
- It allows to create simple threaded programs.
- It provides a way to run multiple functions at the same time and also provides a mechanism to synchronize the threads.

4.5 Functions of the Threading Module

There are many functions in the threading module. Some of them are listed below:

1. **threading.activeCount()**: Returns the number of thread objects that are active.
2. **threading.currentThread()**: Returns the number of thread objects in the caller's thread control.
3. **threading.enumerate()**: Returns a list of all thread objects that are currently active.
4. **threading.LineLock()**: This function is used to block the current thread until the lock is released.
5. **threading.Lock()**: This function is used to acquire a lock, and it is used to release the lock.
6. **threading.Condition()**: This function is used to wait until notified or until a timeout occurs.
7. **threading.Semaphore(value=1)**: This function is used to release the semaphore counter.
8. **threading.Thread(group=None, target=None, name=None, args=(), kwargs=, *, daemon=None)**: This function is used to represent an activity that is run in a separate thread of control.
9. **threading.Timer**: This function is used to run a function after a specific interval of time.

Here are some functions used to manage threads: **Here are some functions used to manage threads:**

1. **threading.start()**: This function is used to start a thread.
2. **threading.run()**: This function is used to run a thread.
3. **threading.join([timeout])**: This function is used to wait until the thread terminates.
4. **threading.isAlive()**: This function is used to check whether a thread is still executing.
5. **threading.getName()**: This function is used to return the name of the thread.
6. **threading.setName()**: This function is used to set the name of the thread.

5 Input and Output

5.1 Input

Creating array of 5 numbers

5.2 Output

Display cube and square of 5 numbers with two separate threads resp .

6 Requirements

1. Python 3.7 or above
2. Numpy

7 Code

[1]:
```
import threading

import time

def calc_square(numbers):
    print("calculate square numbers")
    for n in numbers:
        time.sleep(0.2)
        print('square:', n*n)

def cal_cube(numbers):
    print("calculate cube of numbers")
    for n in numbers:
        time.sleep(0.2)
        print('cube:', n*n*n)

arr = [2,3,8,9]
```

7.1 Let us try to run it without threads

[4]:
```
%%time
calc_square(arr)
cal_cube(arr)
```

```
calculate square numbers
square: 4
square: 9
square: 64
square: 81
calculate cube of numbers
cube: 8
cube: 27
cube: 512
cube: 729
CPU times: user 11.3 ms, sys: 222 µs, total: 11.5 ms
Wall time: 1.6 s
```

7.2 Now let us try to use threads

[15]:
```
thread_1 = threading.Thread(target=calc_square, args=(arr,))
thread_2 = threading.Thread(target=cal_cube, args=(arr,))
```

[16]:
```
%%time
thread_1.start()
thread_2.start()
thread_1.join()
```

```
print("Done with thread 1")
thread_2.join()
print("Done with thread 2")
```

```
calculate square numbers
calculate cube of numbers
square: 4
cube: 8
square: 9
cube: 27
square: 64
cube: 512
square: 81
Done with thread 1
cube: 729
Done with thread 2
CPU times: user 1.99 ms, sys: 8.22 ms, total: 10.2 ms
Wall time: 804 ms
```

[ ]:

8 Conclusion

Studied python threading library, functions and its benefits. Implemented threading in python using threading library.

9 FAQ

1. **What's the difference between a process and a thread ?**
   The difference between a process and a thread is that a process is an executing program with its own address space, whereas a thread is a single execution sequence within the process. A process can contain multiple threads. A thread is sometimes referred to as a lightweight process. Some other differences between a process and a thread include:

   - Processes are isolated by default whereas threads share memory.
   - Processes have their own address space, whereas threads share the address space.
   - Process creation is slow, whereas thread creation is fast.
   - Processes are independent, whereas threads are not independent.
   - Process termination is slow, whereas thread termination is fast.
   - Processes don't share memory with other processes, whereas threads share memory with other threads of the same process.
   - Processes are heavyweight operations, whereas threads are light weight operations.
   - Process switching needs interaction with the kernel, whereas thread switching does not need interaction with the kernel.
   - Processes are independent, so if one process is blocked, it doesn't affect the execution of other processes. Threads share the same address space, so if one thread is blocked, it affects the execution of the other threads of the same process.
   - Processes don't need to be synchronised, whereas threads need to be synchronised.

2. **When should use Multithreading.**

   We should use Multithreading in the following cases:

   - When we want to perform multiple tasks at a time so that we can reduce the time taken to perform the tasks.
   - When we want to perform multiple I/O tasks at a time so that there is no wastage of CPU cycles.
   - When we want to perform multiple tasks that are not dependent on each other.