Practical title: Multithreading in Python.
Learning outcomes: Learn the basics of multithreading and thread synchronization in Python.
Module: CMT202
Lecturer: Padraig Corcoran

Part 1 - Creating threads in Python

A process is a program in execution. A single program in execution will consist of a single process. A thread is also a program in execution. A single program in execution may consist of one or more threads.

Multithreading is defined as the ability of a processor to execute multiple threads concurrently. In a simple single-core CPU, it is achieved using frequent switching between threads.

Let us consider an example using the Python threading module.

```
import threading # import the threading module

def print_cube(num):
    # function to print cube of given num
    print("Cube: {}".format(num * num * num))

def print_square(num):
    # function to print square of given num
    print("Square: {}".format(num * num))

if __name__ == "__main__":
    # creating thread
    t1 = threading.Thread(target=print_square, args=(10,))
    t2 = threading.Thread(target=print_cube, args=(10,))

    t1.start() # starting thread 1
    t2.start() # starting thread 2

    t1.join() # wait until thread 1 is completely executed
    t2.join() # wait until thread 2 is completely executed

    print("Done!") # both threads completely executed
```

threading_1.py

Let us try to understand the above code:

To import the threading module, we do:

```
    import threading
```

To create a new thread, we create an object of Thread class. It takes the following arguments:

target: the function to be executed by thread
args: the arguments to be passed to the target function

In the above example, we created 2 threads with different target functions:

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

To start a thread, we use the start method of Thread class.

```
t1.start()
t2.start()
```

Once the threads start, the current program (you can think of it like a main thread) also keeps on executing. In order to stop execution of current program until a thread is complete, we use join method:

```
t1.join()
t2.join()
```

As a result, the current program will first wait for the completion of t1 and t2. Once they are finished, the remaining statements of the current program are executed.

Part 2 - Race Conditions

All threads in a given program in execution share global variables within that program. Thread synchronization is defined as a mechanism which ensures that two or more concurrent threads do not simultaneously execute some particular program segment known as critical section. A *critical section* is a part of the program where the shared resource is accessed. Please read the following Wikipedia article for more info https://en.wikipedia.org/wiki/Critical_section

Concurrent accesses to a shared resource can lead to a race condition. A race condition occurs when two or more threads can access shared data and they try to change it at the same time. As a result, the values of variables may be unpredictable and vary depending on the timings of context switches of the threads. Please read the following Wikipedia article for more info https://en.wikipedia.org/wiki/Race_condition

To illustrate the concept of race condition run the following program.

```
import threading
x = 0 # global variable x

def increment():
   # function to increment global variable x
   global x
   x += 1

def thread_task():
   # task for thread
```

```
    for _ in range(100000):
        increment()

def main_task():
    global x
    x = 0 # setting global variable x as 0

    # creating threads
    t1 = threading.Thread(target=thread_task)
    t2 = threading.Thread(target=thread_task)

    # start threads
    t1.start()
    t2.start()

    # wait until threads finish their job
    t1.join()
    t2.join()

if __name__ == "__main__":
    for i in range(10):
        main_task()
        print("Iteration {0}: x = {1}".format(i,x))
```

threading_2.py

In the above program two threads t1 and t2 are created in main_task function and the global variable x is set to 0. Each thread has a target function thread_task in which the increment function is called 100000 times. This increment function will increment the global variable x by 1 in each call.

Since we have two threads and each thread attempts to increment the global variable x 100000 times, the expected final value of x is 200000. However what we get in 10 iterations of main_task function is some different values. This happens due to the concurrent accesses of the shared variable x by the threads. This unpredictability in value of x is the consequence of a race condition.

Figure 1 below illustrates how a race condition can occur in the above program. In this figure there are two threads entitled Thread 1 and Thread 2. There is also an integer entitled x. In this figure time increases as we move left to right. Thread 1 attempts to add one to the value of x which equals 10. It does this by reading the value of x, adding one to this value before writing the new value back which equals 11.

Thread 2 also attempts to add one to the value of x in the same way as Thread 1. Thread 2 reads the value of x after Thread 1 has read its value but before Thread 1 has written the new value back.

Since both Thread 1 and Thread 2 attempt to add one to the value of x, the new expected value of x is 12; that is x=10+1+1. However, due to the race condition, it turns out to be 11! The race conditon in question is the fact that Thread 2 reads the value of x after Thread 1 has read its value and before Thread 1 has written the new value back.

Therefore, we need a tool for synchronization between multiple threads and prevent race conditions.

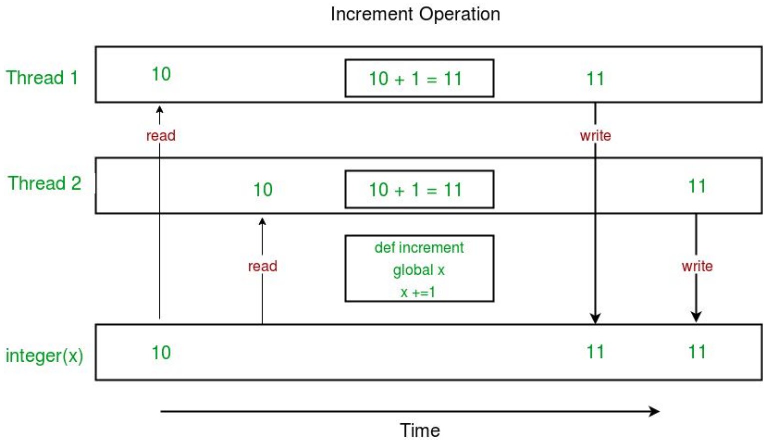

Figure 1. Example of a race condition in the program threading_2.py.

Race conditions can lead to erros or bugs in code which are very difficult to fit. Explain why this is the case (Hint: do a little research on the concept of reproducibility).

Part 3 - Using Locks

The Python threading module provides a Lock class to deal with the race conditions. A lock (also known as a semaphore) is used to control access to a common resource by multiple threads and prevent race conditions. Please read the following Wikipedia article for more info https://en.wikipedia.org/wiki/Semaphore_(programming)

A lock is in one of two states: "locked" or "unlocked". It is created in an unlocked state. It has two basic methods, acquire() and release(). When the state is unlocked, acquire() changes the state to locked and returns immediately. When the state is locked, acquire() blocks until a call to release() in another thread changes it to unlocked, then the acquire() call resets it to locked and returns.

The release() method should only be called in the locked state; it changes the state to unlocked and returns immediately. If an attempt is made to release an unlocked lock, a RuntimeError will be raised.

To illustrate the concept of locks run the following program.

```
import threading

x = 0 # global variable x

def increment():
    # function to increment global variable x
    global x
    x += 1
```

```
def thread_task(lock):
    # task for thread. calls increment function 100000 times.
    for _ in range(100000):
        lock.acquire()
        increment()
        lock.release()

def main_task():
    global x
    x = 0 # setting global variable x as 0

    lock = threading.Lock() # creating a lock

    # creating threads
    t1 = threading.Thread(target=thread_task, args=(lock,))
    t2 = threading.Thread(target=thread_task, args=(lock,))

    # start threads
    t1.start()
    t2.start()

    # wait until threads finish their job
    t1.join()
    t2.join()

if __name__ == "__main__":
    for i in range(10):
        main_task()
        print("Iteration {0}: x = {1}".format(i,x))
```

Threading_3.py

Let us try to understand the above code step by step:

Firstly, a Lock object is created using:

```
lock = threading.Lock()
```

Then, lock is passed as target function argument:

```
t1 = threading.Thread(target=thread_task, args=(lock,))
t2 = threading.Thread(target=thread_task, args=(lock,))
```

In the critical section of the target function, we apply lock using the lock.acquire() method. As soon as a lock is acquired, no other thread can access the critical section (here, increment function) until the lock is released using lock.release() method.

```
lock.acquire()
increment()
```

```
        lock.release()
```

As you can see in the results, the final value of x comes out to be 200000 every time (which is the expected final result).

The diagram below in Figure 2 illustrates the implementation of locks in the above program. In this diagram we can see that Thread 1 reads and locks x. Locking prevents the situation illustrated in Figure 1 where Thread 2 reads the value of x after Thread 1 has read its value but before Thread 1 has written the new value back.

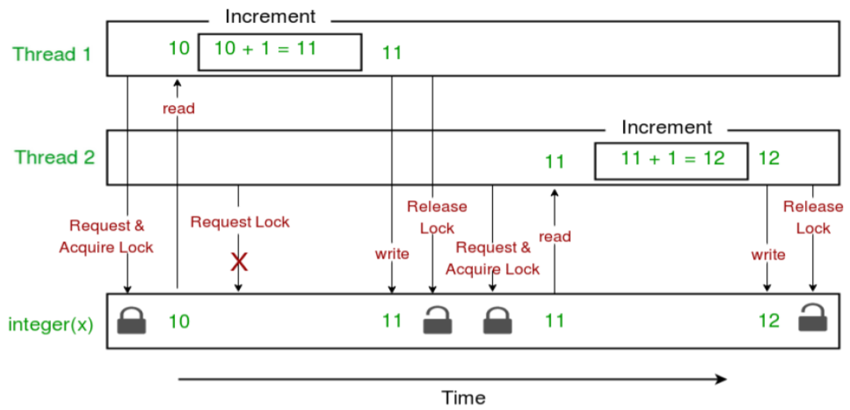

Figure 2. Example of lock in program threading_3.py.

Part 4 - Reflection
Compare the running times of the Python programs threading_2.py and threading_3.py. Are the running times different and if so why?

Review the lecture slides from the section entitled *Processes*. Describe how the theory in these slides relates to the computer programs you have considered during this lab.

References
https://docs.python.org/3.8/library/threading.html
https://www.geeksforgeeks.org/multithreading-python-set-1/
https://www.geeksforgeeks.org/multithreading-in-python-set-2-synchronization/