Session 3 : Threading

Razmig Kéchichian

Learning outcomes

The aim of this session is to familiarize you with:

- multithreading in Python, and
- task parallelism with threads.

I. Python threads

Recall that threads are separate execution paths within a single process. As such, they share program code, data and external resources, such as open files. Also recall that CPython, the reference and most widely-used Python implementation, uses a global interpreter lock (GIL) the consequence of which is that only one thread can execute Python code at a time, that is, Python threads run *concurrently* and not in parallel. In most situations, this doesn't constitute a limitation; first, multiprocessing is a better choice for CPU-bound tasks in which case the GIL has no impact, and second, multithreading is a valid approach for IO-bound tasks since a thread releases the GIL when it has to wait for IO allowing other threads to execute. To sum up, the usual use case for threads is to improve the responsiveness of especially user-facing applications by running tasks, usually IO, in the background. It is not uncommon to design complex programs as multiple processes some multi others single-threaded with interprocess communication (IPC).

Python thread creation and synchronization tools are defined in the standard module `threading`. The API of this module is rather similar to that of the `multiprocessing` module. The abstraction representing a thread is the

`threading.Thread` class which, similarly to `multiprocessing.Process`, allows you to create a new thread in your Python program either by:

1. subclassing it and defining its `run()` method, or
2. simply by creating a `threading.Thread` object and passing it the name and the arguments of the function to be executed in a separate thread.

To illustrate the use of `threading.Thread`, here's a multithreaded solution of the Fibonacci series generation exercise from Session 1 (without error checking).

```
import sys
import threading

def fibonacci(n):
    print("Starting thread:", threading.current_thread().name)
    res = [0]
    a, b = 0, 1
    i = 0
    while i < n:
        a, b = b, a+b
        res.append(a)
        i += 1
    print(res)
    print("Ending thread:", threading.current_thread().name)

if __name__ == "__main__":
    print("Starting thread:", threading.current_thread().name)
    index = int(sys.argv[1])
    thread = threading.Thread(target=fibonacci, args=(index,))
    thread.start()
    thread.join()
    print("Ending thread:", threading.current_thread().name)
```

Execute this program and observe its behavior. As you have probably guessed, the method `thread.start()` starts the thread's execution, and the method `thread.join()` blocks the calling thread until `thread` terminates. 'print' statements serve the purpose of logging thread start/end on the standard output. Before moving to the first exercise, have a quick look on the documentation of the `threading.Thread` class.

Exercise 1: How much is π ?

A classic approach to estimate π is using the Monte Carlo method which involves randomization. This works as follows. Frist, assume a circle of radius equal to 1, inscribed within a square and centered on the origin of the Cartesian plane as in the figure above. Next, generate a series of random points (x, y) which fall within the coordinates of the square. Some of the generated points will occur within the circle. Then π is estimated by:

$$\pi = 4 \times (\text{number of points in circle})/(\text{total number of points})$$

Write a multithreaded program that creates a separate thread to generate a number of random points. The thread will count the number of points that occur within the circle and store the result in a global variable. When this thread exits, the main thread will calculate and output the estimated value of π. Experiment with the number of random points generated.

Hints:

- To assert that a point (x, y) falls within a circle of radius R, it suffices that $x^2 + y^2 <= R$.
- You can generate random real numbers falling into the interval $[0, 1)$ via `random.random()`

II. Task parallelism with threads

Recall that *task parallelism* involves distributing operations across multiple workers (threads or processes), each performing a unique operation on often the same data. In contrast, *data parallelism* distributes subsets of data across multiple workers each performing the same operation. In practice, however, few applications strictly follow either data or task parallelism. In most cases, a hybrid of these two strategies is used.

The `threading` module provides a number of useful tools to synchronize the concurrent execution of threads most of which will be addressed in due course

during later sessions. For the purposes of the following exercise, we shall look at the simplest of these synchronization tools, the `Event` class which allows for a thread to signal the occurrence of an event for other threads waiting for it. Have a look on the documentation of this class to understand how it works.

The `queue` standard module provides thread-safe queue objects with similar API and patterns of use to the `multiprocessing.Queue` class we encountered during the last session. You may want to review that information to refresh your memory. In the following exercise, we shall use the class `queue.Queue`.

The following program illustrates the use of both `Event` and `Queue` objects. A main thread creates a worker thread which waits on an event object. Next, the main thread reads some data from the standard input and puts it in the queue. Once data is available, the main thread signals the event allowing the worker thread to proceed with reading it from the queue.

```
import threading
from queue import Queue

def worker(queue, data_ready):
    print("Starting thread:", threading.current_thread().name)
    data_ready.wait()
    value = queue.get()
    print("got value:", value)
    print("Ending thread:", threading.current_thread().name)

if __name__ == "__main__":
    print("Starting thread:", threading.current_thread().name)

    queue = Queue()
    data_ready = threading.Event()

    thread = threading.Thread(target=worker, args=(queue, data_ready))
    thread.start()

    value = input("give me some value:")
    queue.put(value)
    data_ready.set()

    thread.join()

    print("Ending thread:", threading.current_thread().name)
```

Exercise 2: the INSEE office

We would like to calculate the following statistics on numeric data: min, max, median, mean and standard deviation. Write a multithreaded program where the main thread creates a number of workers equal to the number of statistics to perform, reads data from the terminal and signals data availability to its workers which proceed to calculate and display statistics. **Do not** define 5 different functions to be executed in worker threads, define a single worker function and pass operations as an argument in a `queue.Queue`.

Hints:

- To calculate statistics, use built-in functions `min` and `max`, and `statistics` module functions `median`, `mean` and `stdev`.
- Every thread displays the result of its computation. You may also choose to return computation results to the main thread in a `queue.Queue`.
- To read data from the standard input until end-of-file (EOF) is encountered (`Ctrl+D` on Unix/Linux, `Ctrl+Z` on Windows), you can use the following code snippet.

```
data = []
input_str = sys.stdin.read().split()
for s in input_str:
    try:
        x = float(s)
    except ValueError:
        print("bad number", s)
    else:
        data.append(x)
```

- In your initial solution, display the calculated statistic in the worker thread. Extend this version to put statistics calculated in worker threads into a queue, the contents of which are displayed by the main thread.