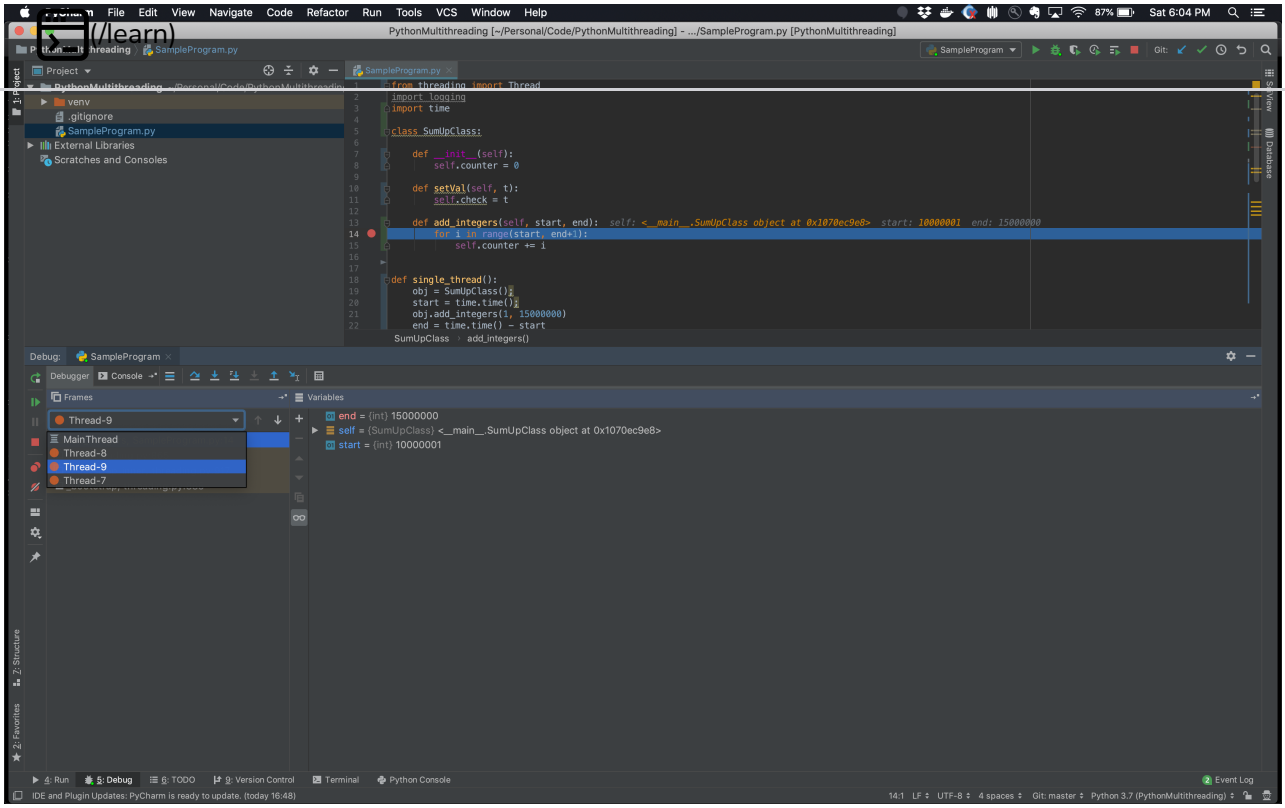
Introduction

This lesson introduces concurrency and provides motivational examples to further our understanding of concurrent systems.

Introduction

Understanding of concurrency and its implementation models using either threads or coroutines exhibits maturity and technical depth in a candidate, which can be an important differentiator in landing a higher leveling offer at a company. More importantly, grasp over concurrency lets us write performant and efficient programs. In fact, you'll inevitably run into the guts of the topic when working on any sizeable project.

We'll start with threads as they are the most well-known and familiar concepts in the context of concurrency. Threads, like most computer science concepts, aren't physical objects. The closest tangible manifestation of a thread can be seen in a debugger. The screen-shot below shows the threads of an example program suspended in the debugger.



The simplest example of a concurrent system is a single-processor machine running your favorite IDE. Say you edit one of your code files and click save. That clicking of the button will initiate a workflow which will cause bytes to be written out to the underlying physical disk. However, IO is an expensive operation, and the CPU will be idle while bytes are being written out to the disk.

Whilst IO takes place, the idle CPU could work on something useful. Here is where threads come in - the IO thread is ***switched out*** and the UI thread gets scheduled on the CPU so that if you click elsewhere on the screen, your IDE is still responsive and does not appear hung or frozen.

Threads can give the illusion of multitasking even though the CPU is executing only one thread at any given point in time. Each thread gets a slice of time on the CPU and then gets switched out either because it

initiates a task which requires waiting and not utilizing the CPU or it completes its time slot on the CPU. There are many nuances and intricacies on how thread scheduling works but what we just described forms the basis of it.

With advances in hardware technology, it is now common to have multi-core machines. Applications can take advantage of these architectures and have a dedicated CPU run each thread. However, as we'll learn in this course, the standard version of Python is limited in its ability to leverage multiple processors in a system when running a multithreaded-program.

Benefits of Threads

1. **Higher throughput**, though in some pathetic scenarios it is possible to have the overhead of context switching among threads steal away any throughput gains and result in worse performance than a single-threaded scenario. However, such cases are unlikely and are exceptions rather than the norm. Also, multithreaded programs can utilize multiple cores in a machine but a single-threaded program limits itself to a single core even if multiple are available.

2. **Responsive applications** that give the illusion of multitasking on single core machines.

3. **Efficient utilization of resources**. Note that thread creation is lightweight in comparison to spawning a brand new process. Web servers that use threads instead of creating new processes when fielding web requests consume far fewer resources. Moreover, because all threads within a process share the same address space, they need not use shared memory, unlike processes.

All other benefits of multi-threading are extensions of or indirect

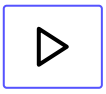
All other benefits of multi-threading are extensions of or indirect benefits of the above.

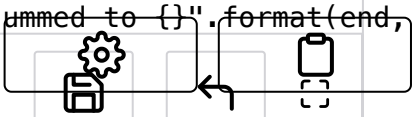
Performance Gains via Multi-Threading

As a concrete example, consider the example code below. The task is to ***compute the sum of all the integers from 0 to 300000000 (30 million)***. In the first scenario, we have a single thread doing the summation while in the second scenario, we split the range into two parts and have one thread sum for each range. Once both the threads are complete, we add the two half sums to get the combined sum. Finally, we repeat the previous exercise using processes instead of threads. We measure the time taken for each scenario and print it.

```
from threading import Thread
from multiprocessing import Process

from multiprocessing.managers import BaseManager
from queue import Queue
import time
import multiprocessing


class SumUpClass:

    def __init__(self):
        self.counter = 0

    def add_integers(self, start, end):
        for i in range(start, end + 1):
            self.counter += i

    def get_counter(self):
        return self.counter


def single_thread():
    obj = SumUpClass();
    start = time.time();
    obj.add_integers(1, 30000000)
    end = time.time() - start
```

```
28      print("single threaded took : {} seconds and summed to {}".format(end, o
```





The table below presents the time in seconds it takes for various scenarios to sum up integers from 1 to 30 million.

Setup	**Time (secs)**
Single Thread	9.9
Multiple Threads	12.3
Multiple Processes	13.1

The results are interesting and counterintuitive. We would expect the multithreaded scenario to perform better than the single-threaded one since two threads can work in parallel if the system has at least two CPUs. The relatively poor performance in comparison to the single-threaded scenario can be explained by (standard) Python's Achilles' heel, the **Global Interpreter Lock**, an entity within the Python framework that allows a single thread to execute even in the presence of more than one idle CPUs. We'll have more to say on that subject in later sections. Multithreaded scenarios may not experience any performance gains in case of CPU-intensive tasks such as the one in our example because threads don't execute in parallel and in fact incur an additional cost of

management and coalescing partial results.

With multiple processes we can expect each process to be scheduled onto a separate CPU and work in parallel. However, there's the additional overhead of creating and tearing down processes, which is higher than doing the same for threads. Additionally, we utilize Python's inter-process communication machinery, which uses the proxy-pattern and adds network latency to the multiprocessing scenario. Collectively these headwinds explain why the multiprocessing scenario performs no better than a single-threaded scenario.

However, in general tasks involving blocking operations such as network and disk I/O, can see significant performance gains when migrated to a multithreaded or multiprocessor architecture.

Problems with Threads

There's no free lunch in life. The premium for using threads manifests in the following forms:

1. ***It's usually very hard to find bugs***, some that may only rear their heads in production environments. Bugs can't be reproduced consistently and may depend on the sequence in which various threads execute.
2. ***Higher cost of code maintenance*** since the code inherently becomes harder to reason about.
3. ***Increased utilization of system resources***. Creation of each thread consumes additional memory, CPU cycles for book-keeping, and waste of time in context switches.
4. ***Programs may experience slowdown*** as coordination amongst threads comes at a price. Acquiring and releasing locks adds to program execution time. Threads fighting over acquiring locks

cause lock contention.

With this backdrop lets delve into more details of concurrent programming which you are likely to be quizzed about in an interview.

Next →

Program vs Process vs Thread

Mark as Completed