The Why, When, and How of Using Python Multi-threading and Multi-Processing

This guide aims to explain why multi-threading and multi-processing are needed in Python, when to use one over the other, and how to use them in your programs. As an AI researcher, I use them extensively when preparing data for my models!

Thilina Rajapakse

Image by Parker_West from Pixabay

A long time ago in a galaxy far, far away….

A wise and powerful wizard lives in a small village in the middle of nowhere. Let's call him Dumbledalf. Not only is he wise and powerful, but he's also happy to help anyone who asks and this means that people come from far and wide to ask the wizard for aid. Our story begins when on one fine day, a young traveler brings a magical scroll to the wizard. The traveler has no idea what the scroll contains, but he knows that if anyone can decipher the secrets of the scroll, it will be the great wizard, Dumbledalf.

Chapter 1: Single-threaded, single-process

If you haven't guessed already, my rather soppy analogy is talking about a CPU and its functions. Our wizard is the CPU and the magical scroll is a list of *URLs* which leads to the power of Python and the knowledge to wield that power.

The wizard's first thought, having deciphered the scroll without too much trouble, was to send his trusted friend *(Haragorn? I know, I know, that's terrible)* to each of the locations given in the scroll to see and bring back what he can find.

```
In [1]: import urllib.request
        from concurrent.futures import ThreadPoolExecutor

In [2]: urls = [
          'http://www.python.org',
          'https://docs.python.org/3/',
          'https://docs.python.org/3/whatsnew/3.7.html',
          'https://docs.python.org/3/tutorial/index.html',
          'https://docs.python.org/3/library/index.html',
          'https://docs.python.org/3/reference/index.html',
          'https://docs.python.org/3/using/index.html',
          'https://docs.python.org/3/howto/index.html',
          'https://docs.python.org/3/installing/index.html',
          'https://docs.python.org/3/distributing/index.html',
          'https://docs.python.org/3/extending/index.html',
          'https://docs.python.org/3/c-api/index.html',
          'https://docs.python.org/3/faq/index.html'
          ]

In [3]: %%time

        results = []
        for url in urls:
            with urllib.request.urlopen(url) as src:
                results.append(src)
```

for_loop.ipynb hosted with ❤ by GitHub view raw

As you can see, we are simply plodding through the *URLs* one by one using a `for loop` and reading the response. Thanks to `%%time` the magic from *IPython,* we can

see that it takes about 12 seconds with my deplorable internet.

Chapter 2: Multi-threading

Not for naught was the wizard's wisdom famed across the land, and he quickly comes up with a much more efficient method. Instead of sending one person to each of the locations in order, why not get together a bunch of (trustworthy) people and send them separately to each of the locations, *at the same time!* The wizard can simply combine everything they bring once they all come back.

That's right, instead of looping through the list one by one, we can use `multithreading` to access multiple *URLs* at the same time.

```
In [1]: import urllib.request
        from concurrent.futures import ThreadPoolExecutor

In [2]: urls = [
          'http://www.python.org',
          'https://docs.python.org/3/',
          'https://docs.python.org/3/whatsnew/3.7.html',
          'https://docs.python.org/3/tutorial/index.html',
          'https://docs.python.org/3/library/index.html',
          'https://docs.python.org/3/reference/index.html',
          'https://docs.python.org/3/using/index.html',
          'https://docs.python.org/3/howto/index.html',
          'https://docs.python.org/3/installing/index.html',
          'https://docs.python.org/3/distributing/index.html',
          'https://docs.python.org/3/extending/index.html',
          'https://docs.python.org/3/c-api/index.html',
          'https://docs.python.org/3/faq/index.html'
          ]

In [4]: %%time

        with ThreadPoolExecutor(4) as executor:
            results = executor.map(urllib.request.urlopen, urls)

        CPU times: user 122 ms, sys: 8.27 ms, total: 130 ms
        Wall time: 3.83 s
```

multithreading.ipynb hosted with ❤ by GitHub view raw

Much better! Almost like.. magic. Using multiple threads can significantly speed up many tasks that are ***IO-bound***. Here, the vast portion of the time taken to read the *URLs* is due to the network delay. ***IO-bound*** programs spend most of their time waiting for, you guessed it, input/output *(Similar to how the wizard needs to wait for his friend/friends to go to the locations given in the scroll and come back)*. This may be I/O from a network, a database, a file, or even a user. This I/O tends to take a

significant amount of time, as the source itself may need to perform its own processing before passing on the I/O. For example, the CPU works much, *much* faster than a network connection can shuttle data *(Think Flash vs your grandma)*.

Note: `Multithreading` *can be very useful in tasks like web scraping.*

Chapter 3: Multi-processing

As the years rolled on and our wizard's fame grew, so did the envy of one rather unpleasant dark wizard (Sarudort? Voldeman?). Armed with devious cunning and driven by jealousy, the dark wizard performed a terrible curse on Dumbledalf. As soon as the curse settled, Dumbledalf knew that he had mere moments to break it. Tearing through his spellbooks in desperation, he finds a counter-spell that looks like it might do the trick. The only problem is that it requires him to calculate the sum of all prime numbers below 1000000. Weird spell, but it is what it is.

Now, the wizard knows that calculating the value will be trivial given enough time but time is not a luxury that he has. Wizard though he is, even he is limited by his humanity and he can only calculate one number at a time. If he were, to sum up the prime numbers one by one, it would take far too long. With seconds left to reverse the curse, he suddenly remembers the `multiprocessing` spell he learned from the magic scroll years ago. This spell would allow him to make copies of himself, and splitting up the numbers between his copies would allow him to check if multiple numbers are primes, simultaneously. Finally, all he has to do is add up all the prime numbers that he and his copies discover.

In [1]:
```
from multiprocessing import Pool
```

In [2]:
```
def if_prime(x):
    if x <= 1:
        return 0
    elif x <= 3:
        return x
    elif x % 2 == 0 or x % 3 == 0:
        return 0
    i = 5
    while i**2 <= x:
        if x % i == 0 or x % (i + 2) == 0:
            return 0
        i += 6
    return x
```

In [17]:
```
%%time

answer = 0
```

```
for i in range(1000000):
    answer += if_prime(i)
```

```
CPU times: user 3.48 s, sys: 0 ns, total: 3.48 s
Wall time: 3.48 s
```

multiprocessing.ipynb hosted with ❤ by GitHub view raw

With modern CPU's generally having more than a single core, we can speed up ***CPU bound*** tasks by using the `multiprocessing` module. ***CPU bound*** tasks are programs that spend most of their time performing calculations in the CPU (mathematical computations, image processing, etc.). If the calculations can be performed independently of each other, we can split them up among the available CPU cores thereby gaining a significant boost to processing speed.

All you have to do is;

1. Define the function to be applied
2. Prepare a list of items that the function is to be applied on
3. Spawn processes using `multiprocessing.Pool`. The number passed to `Pool()` will be the number of processes spawned. Embedding inside a `with` statement ensures that the processes are killed after finishing execution.
4. Combine the outputs using the `map` function of a Pool process. The inputs to the `map` function are the function to be applied to each item, and the list of items.

Note: The function can be defined so as to perform any task that can be done in parallel. For example, the function may contain code to write the result of a computation to a file.

So, why do we need separate `multiprocessing` and `multithreading`? If you tried to use `multithreading` to improve the performance of a ***CPU bound*** task, you might notice that what you actually get is a ***degradation*** in performance. Heresy! Let's see why this happens.

Much like the wizard being limited by his human nature and only being able to calculate one number at a time, Python comes with something called the **Global Interpreter Lock (GIL).** Python will happily let you spawn as many `threads` as you

like, but the **GIL** ensures that only *one* of those `threads` will ever be *executing* at any given time.

For an ***IO-bound*** task, that is perfectly fine. One `thread` fires off a request to a URL and while it is waiting for a response, that `thread` can be swapped out for another `thread` that fires another request to another URL. Since a `thread` doesn't have to do anything until it receives a response, it doesn't really matter that only one `thread` is *executing* at a given time.

For a ***CPU bound*** task, having multiple `threads` is about as useful as nipples on a breastplate. Because only one `thread` is being executed at a time, even if you spawn multiple `threads` with each having their own number to be checked for *prime-ness*, the CPU is still only going to be dealing with one `thread` at a time. In effect, the numbers will still be checked one after the other. The overhead in dealing with multiple `threads` will contribute to the performance degradation you may observe if you use `multithreading` in a ***CPU bound*** task.

To get around this 'limitation', we use the `multiprocessing` module. Instead of using `threads`, `multiprocessing` uses, well, multiple `processes`. Each `process` gets its own interpreter and memory space, so the GIL won't be holding things back. In essence, each `process` uses a different CPU core to work on a different number, *at the same time*. Sweet!

You may notice that CPU utilization goes much higher when you are using `multiprocessing` compared to using a simple for loop, or even `multithreading`. That is because multiple CPU cores are being used by your program, rather than just a single core. This is a good thing!

Keep in mind that `multiprocessing` comes with its own overhead to manage multiple `processes`, which typically tends to be heavier than `multithreading` overhead. (`Multiprocessing` spawns a separate interpreter, and assigns a separate memory space for each `process`, so duh!). This means that, as a rule of thumb, it is better to use the lightweight `multithreading` when you can get away with it (read: ***IO-bound*** tasks). When CPU processing becomes your bottleneck, it's generally time to summon the `multiprocessing` module. But remember, *with great power comes great responsibility*.

If you spawn more `processes` than your CPU can handle at a time, you will notice your performance starting to drop. This is because the operating system now has to do more work swapping `processes` in and out of the CPU cores since you have more `processes` than cores. The reality might be more complicated than a simple explanation, but that's the basic idea. You can see a drop-off in performance on my system when we reach 16 `processes`. This is because my CPU only has 16 logical cores.

Chapter 4: TLDR;

- For ***IO-bound*** tasks, using `multithreading` can improve performance.
- For ***IO-bound*** tasks, using `multiprocessing` can also improve performance, but the overhead tends to be higher than using `multithreading`.
- The Python GIL means that only `thread` can be executing at any given time in a Python program.
- For ***CPU bound*** tasks, using `multithreading` can actually worsen the performance.
- For ***CPU bound*** tasks, using `multiprocessing` can improve performance.
- Wizards are awesome!

That concludes this introduction to `multithreading` and `multiprocessing` in Python. Go forth and conquer!