

★ Get unlimited access to the best of Medium for less than \$1/week. [Become a member](#)



Multithreading VS Multiprocessing in Python

Clearing up misconceptions with live experiments



Amine Baatout · [Follow](#)

Published in Contentsquare Engineering

7 min read · Dec 5, 2018



Listen



Share

... More

This blog has moved! Click [here](#) to read it in our [new website](#).

In this article, I will try to discuss some misconceptions about Multithreading and explain why they are false.

All experiments are conducted on a machine with 4 cores (EC2 c5.xlarge).

Open in app ↗



Search



Pythons enjoying a nice thread-pool party.

I've been dealing with parallelism in python for quite a while, and I was constantly reading articles and stackoverflow threads in order to improve my understanding of the subject. Normally the more you search the more you learn. However in the case of multithreading/multiprocessing, the more I searched the more I got confused. Here's an example:

Multiprocessing vs Threading Python

600 I am trying to understand the advantages of [multiprocessing](#) over [threading](#). I know that **multiprocessing** gets around the Global Interpreter Lock, but what other advantages are there, and can **threading** not do the same thing?

python multiprocessing multiprocessing

top answer ↓

524 The `threading` module uses threads, the `multiprocessing` module uses processes. The difference is that threads run in the same memory space, while processes have separate memory. This makes it a bit harder to share objects between processes with multiprocessing. Since threads use the same memory, precautions have to be taken or two threads will write to the same memory at the same time. This is what the global interpreter lock is for.

✓ Spawning processes is a bit slower than spawning threads. Once they are running, there is not much difference.

😞 Not quite true...

While the first part of the answer is correct, the last is completely false.

I'm not attacking the person who wrote the answer, on the contrary: I have the upmost respect for anyone who tries to help other people. I only used this example to show that some explanations about multithreading can be misleading. Moreover, some other explanations employ advanced terms and could make things harder than they really are not.

⚠️ *PS: I'll try to keep things simple: so no talking about GIL, Memory, Pickling, Overhead. (Although I'll talk about overhead just a little bit).*

Let's get started!

Multiprocessing and Multithreading are basically the same thing.

✗ FALSE!

[\[Link the whole experiment code\]](#)

I'm going to start with a simple experiment and I will borrow the code from [this article](#) written by [Brendan Fortuner](#) which is a great read by the way.

Suppose we have this task which we will execute many times.

```
def cpu_heavy(x):
    print('I am', x)
    count = 0
```

```
for i in range(10**8):
    count += i
```

Next we will try both Multiprocessing or Multithreading

```
from concurrent.futures import ProcessPoolExecutor,
ThreadPoolExecutor

def multithreading(func, args, workers):
    with ThreadPoolExecutor(workers) as ex:
        res = ex.map(func, args)
    return list(res)

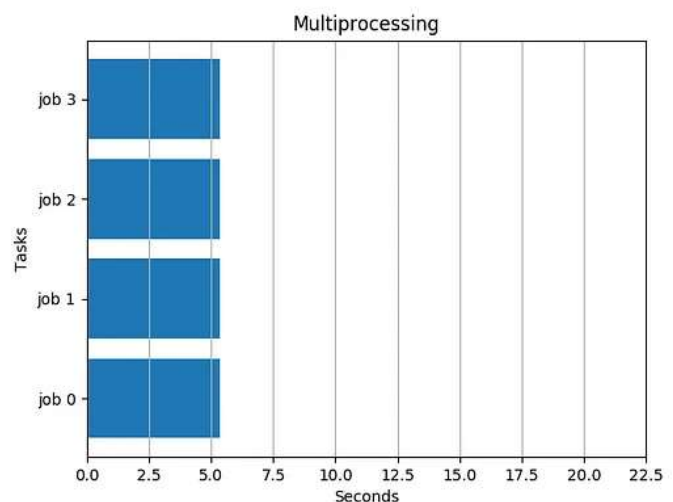
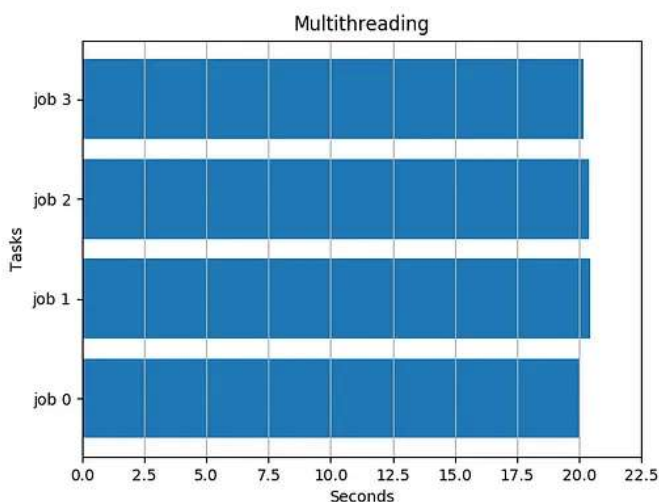
def multiprocessing(func, args, workers):
    with ProcessPoolExecutor(workers) as ex:
        res = ex.map(func, args)
    return list(res)
```

⚠ Note that if you implement:

- *Multiprocessing with multiprocessing OR concurrent.futures ...*
 - *Multithreading with threading OR multiprocessing.dummy OR concurrent.futures ...*
- it won't affect our experiments.

Without further due, let's run some code:

```
visualize_runtimes(multithreading(cpu_heavy, range(4), 4))
visualize_runtimes(multiprocessing(cpu_heavy, range(4), 4))
```



While Multithreading took 20 seconds, Multiprocessing took only 5 seconds.

So now that we are convinced that **they're not the same**, we would like to know **why**.
For that let's move to the next misconception about multithreading. 🙋

In Multithreading, threads run in parallel.

✗ FALSE !

Actually in a ThreadPool, only one thread is being executed at any given time t.

[\[Link the whole experiment code\]](#)

I don't know about you but for me it was a shocker! 🤖

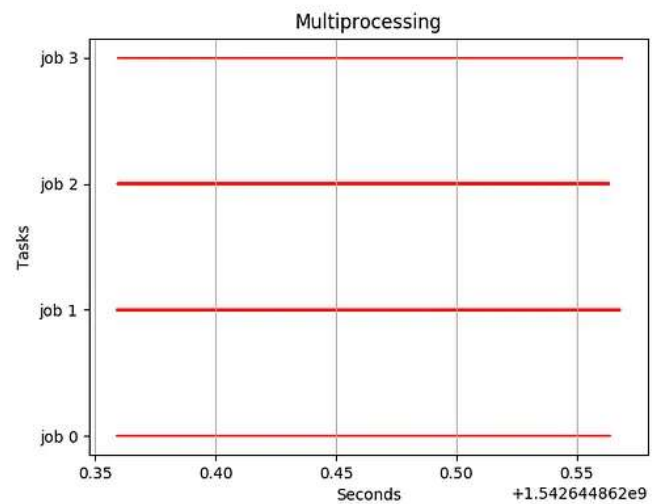
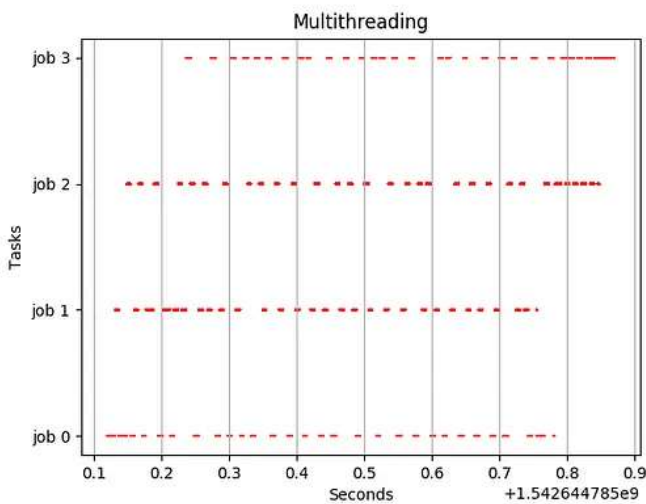
I always thought that threads execute code simultaneously, but this is totally untrue in Python.

Let's do a little experiment. Unlike the previous one, we will not only track the start and stop of each job but also every point in time where the job is running:

```
def live_tracker(x):  
    print('I am', x)  
    l = []  
    for i in range(10**6):  
        l.append(time.time())  
    return l
```

Like before we will run our experiment and produce new graphs.

```
visualize_live_runtimes(multithreading(live_tracker, range(4), 4))  
visualize_live_runtimes(multiprocessing(live_tracker, range(4), 4))
```



Actually threads neither run in parallel nor in sequence. They run concurrently! Each time one job will be executed a little and then the other takes on.

*Concurrency and parallelism are related terms but not the same, and often misconceived as the similar terms. The crucial difference between concurrency and parallelism is that **concurrency** is about dealing with a lot of things at same time (gives the illusion of simultaneity) or handling concurrent events **essentially hiding latency**. On the contrary, **parallelism** is about doing a lot of things at the same time for **increasing the speed**. [Source: techdifferences.com]*

With this said, if you have a cpu heavy task, and you want to make it faster use multiprocessing!

For example if you have 4 cores like I did in my tests, with multithreading each core will be at around 25% capacity while with multiprocessing you will get 100% on each core. This means that with 100% on 4 cores you will get a speedup by 4. How about multithreading's 25% ? will we get any speedup? Answer in next section. 🙌

Multithreading is always faster than serial.

✗ FALSE !

Actually for cpu heavy tasks, multithreading will not only bring nothing good. Worst: it'll make your code even slower!

[\[Link the whole experiment code\]](#)

Dispatching a cpu heavy task into multiple threads **won't speed up the execution.** On the contrary it might degrade overall performance.

Imagine it like this: if you have 10 tasks and each takes 10 seconds, serial execution will take 100 seconds in total. However with multithreading, since only one thread is executed at any given time t, it will be like serial execution **PLUS the time spent to switch between the threads.**

So for the experiment I'm launching 4 heavy cpu jobs, on 4 threads on a 4-cores machine (EC2 c5.xlarge) and comparing it with serial execution.

```
def cpu_heavy(x):  
    count = 0  
    for i in range(10**10):  
        count += i  
  
n_jobs = 4  
  
marker = time.time()  
for i in range(n_jobs): cpu_heavy(i)  
print("Serial spent", time.time() - marker)  
marker = time.time()  
multithreading(cpu_heavy, range(n_jobs), 4)  
print("Multithreading spent", time.time() - marker)
```

Outputs:

```
amine@c5-xlarge:~$ python3 experiment.py  
Serial spent 1658.8452804088593  
Multithreading spent 1668.8857419490814
```

So Multithreading is 10 seconds slower than Serial on cpu heavy tasks, even with 4 threads on a 4 cores machine.

Actually the difference is negligible because it's 10 seconds on a 27 minutes job (0.6% slower), but still, it shows that multithreading is useless in this case.

Is multithreading any good then ?

Multithreading is useless.

 FALSE !

Actually for cpu heavy tasks, multithreading is useless indeed. However it's perfect for IO.

[\[Link the whole experiment code\]](#)

For IO tasks, like querying a database or loading a webpage the CPU is just doing nothing but waiting for an answer. Let's try to query 16 urls, serially than using 4 threads, then using 8:

```
urls = [...] # 16 urls

def load_url(x):
    with urllib.request.urlopen(urls[x], timeout=5) as conn:
        return conn.read()

n_jobs = len(urls)

marker = time.time()
for i in range(n_jobs): load_url(i)
print("Serial spent", time.time() - marker)
marker = time.time()
multithreading(load_url, range(n_jobs), 4)
print("Multithreading 4 spent", time.time() - marker)
marker = time.time()
multithreading(load_url, range(n_jobs), 8)
print("Multithreading 8 spent", time.time() - marker)
```

Ouputs

```
amine@c5-xlarge:~$ python3 serial_comparaison_io.py
Serial spent 7.8587799072265625
Multithreading with 4 spent 2.5494980812072754
Multithreading with 8 spent 1.1110448837280273
Multithreading with 16 spent 0.6199102401733398
```

Notice we have we have gained a significant speedup with multithreading in comparaison to serial! Note also that the more threads you have, the faster your

execution. Of course there is no point of having more threads than the number of urls, this is why I stopped at 16 threads for 16 urls.

Also bear in mind that in your best case scenario, the time of execution with multithreading is equal to the maximum time spent loading one url: If you have 16 urls with one that takes 10 seconds to load and 15 others that take 0.1 second each, using a thread pool of 8 threads will make your program last at least 10 seconds, while in serial it would last 11.5 seconds. So in this case there isn't a huge speedup.

Okay now we know that even though multithreading is bad for CPU, it performs remarkably well for IO.

If multithreading is bad for CPU and good for IO, does this mean that multiprocessing is good for CPU and bad for IO ?

Answer in next section. 🙌

Multiprocessing is bad for IO.

 FALSE !

When it comes to IO, Multiprocessing is overall as good as multithreading. It just has more overhead because popping processes is more expensive than popping threads.

If you like to do an experiment, just replace multithreading with multiprocessing in the previous one.

```
amine@c5-xlarge:~$ python3 serial_comparaison_io.py
Serial spent 5.325972080230713
Multiprocessing 4 spent 1.2662420272827148
Multiprocessing 8 spent 0.8015711307525635
Multiprocessing 16 spent 0.5572431087493896
```

[Bonus] Multiprocessing is always faster than serial.

 TRUE, but only if you do it right ⚠️

For example if you have 1000 cpu heavy task and only 4 cores, don't pop more than 4 processes otherwise they will **compete** for CPU resources.

(compete => competition => concurrency)

Conclusion

- There can only be **one thread running** at any given time in a python process.
- Multiprocessing is parallelism. Multithreading is concurrency.
- Multiprocessing is for increasing speed. Multithreading is for hiding latency.
- Multiprocessing is best for computations. Multithreading is best for IO.
- If you have CPU heavy tasks, use multiprocessing with `n_process = n_cores` and never more. Never!
- If you have IO heavy tasks, use multithreading with `n_threads = m * n_cores` with `m` a number bigger than 1 that you can tweak on your own. Try many values and choose the one with the best speedup because there isn't a general rule. For instance the default value of `m` in `ThreadPoolExecutor` is set to 5 [\[Source\]](#) which honestly feels quite random in my opinion.

That's it. 🎉 🔄

The end.

References

Intro to Threads and Processes in Python

Beginner's guide to parallel programming

medium.com

concurrent.futures - Launching parallel tasks - Python 3.7.1 documentation

When using , this method chops iterables into a number of chunks which it submits to the pool as separate tasks. The...

docs.python.org

baatout/multithreading-vs-multiprocessing

Multithreading VS Multiprocessing in Python. Contribute to baatout/multithreading-vs-multiprocessing development by...

github.com

Python

Parallel Computing

Software Engineering

AI

Machine Learning

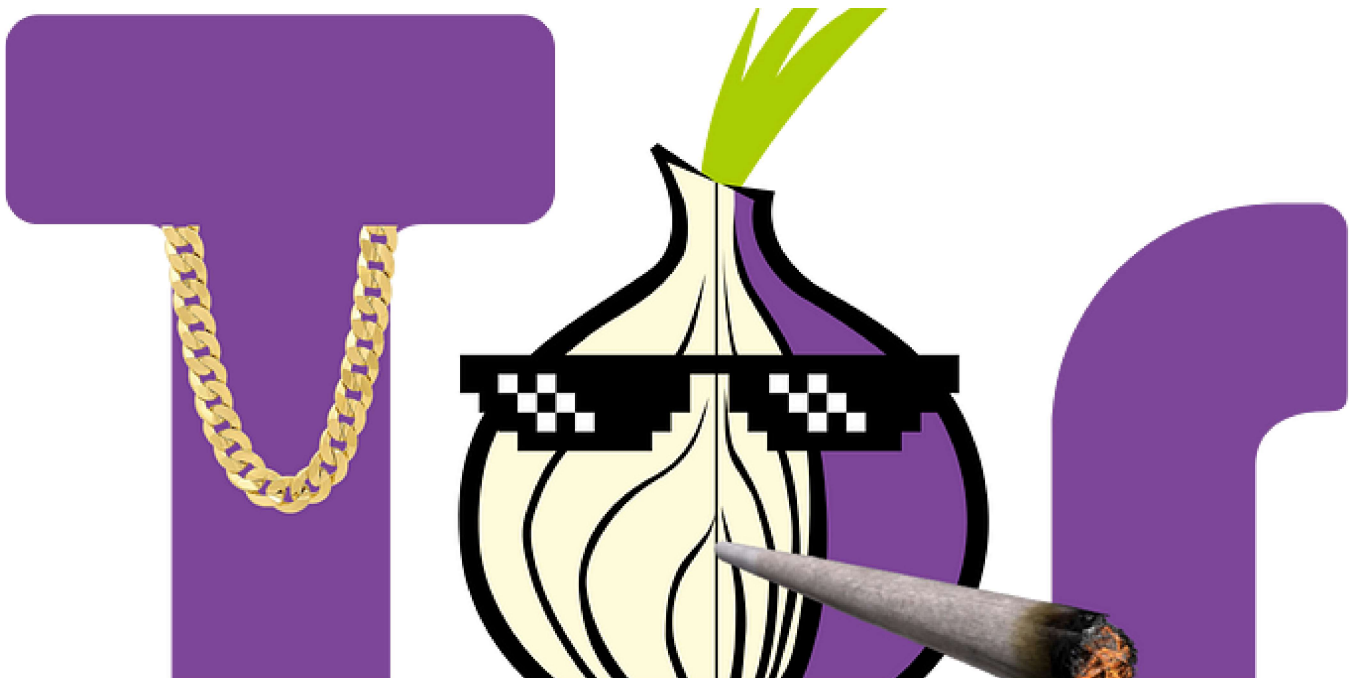


Follow

Written by Amine Baatout

247 Followers · Writer for Contentsquare Engineering

More from Amine Baatout and Contentsquare Engineering





Amine Baatout

Crawling the web with TOR

In this article I will present an example of a resilient crawler that is able to change its IP on demand.

4 min read · Jul 17, 2018

 510  1



 Vincent Chenal in Contentsquare Engineering

Kafka topics sizing: how much messages do I store?

At ContentSquare Engineering , Kafka is a central component in our data collection pipeline. We want it to be hassle free

5 min read · Oct 15, 2019

 588  1



 Julien Dumazert in Contentsquare Engineering

Understanding the layout of webpages using automatic zone recognition

9 min read · Mar 6, 2017

 299  4



 Amine Baatout in Towards Data Science

A 10-line proof of back propagation

A vectorised proof with no summations nor indices.

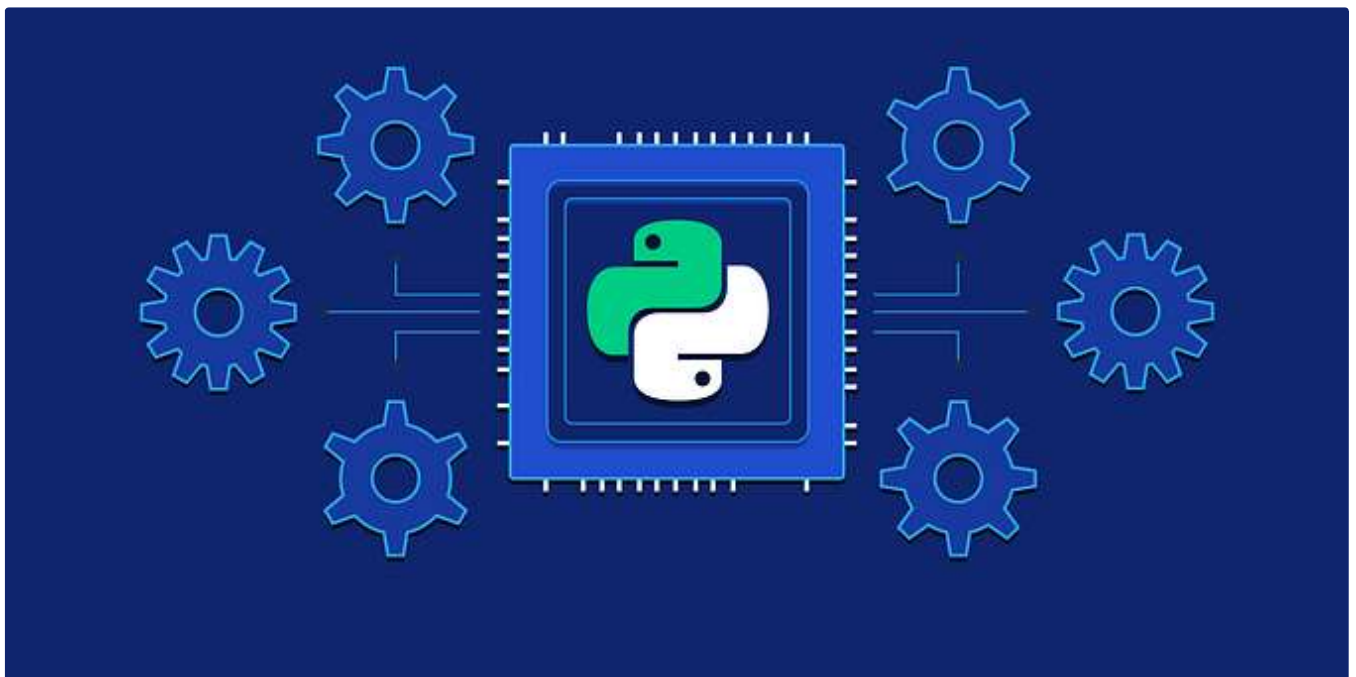
7 min read · Nov 29, 2019



220

[See all from Amine Baatout](#)[See all from Contentsquare Engineering](#)

Recommended from Medium



Daniel Wu

Exploring Concurrency in Python: asyncio, Multithreading, and Multiprocessing

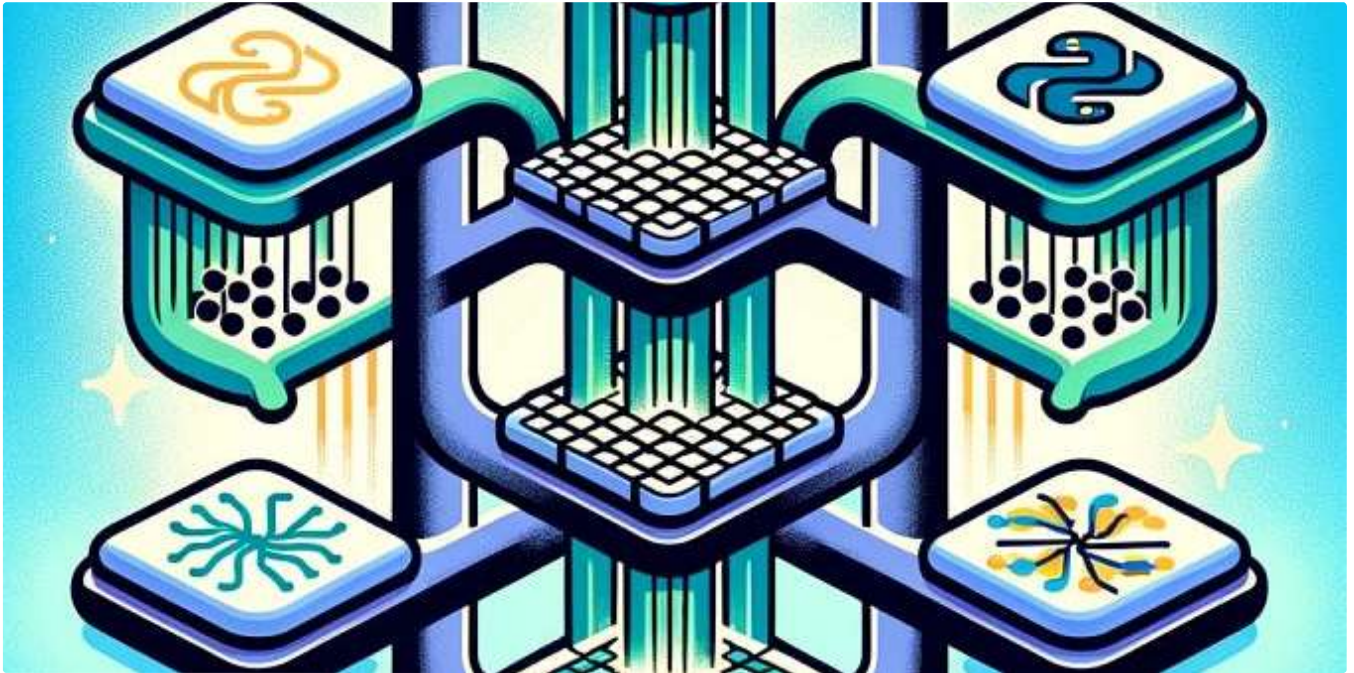
Introduction

6 min read · Oct 28, 2023



15





Utkarsh Singh in Stackademic

Advanced Guide to Asyncio, Threading, and Multiprocessing in Python

Python offers diverse paradigms for concurrent and parallel execution: Asyncio for asynchronous programming, Threading for concurrent...

3 min read · Dec 17, 2023



11



1



Lists



Coding & Development

11 stories · 444 saves



Predictive Modeling w/ Python

20 stories · 894 saves



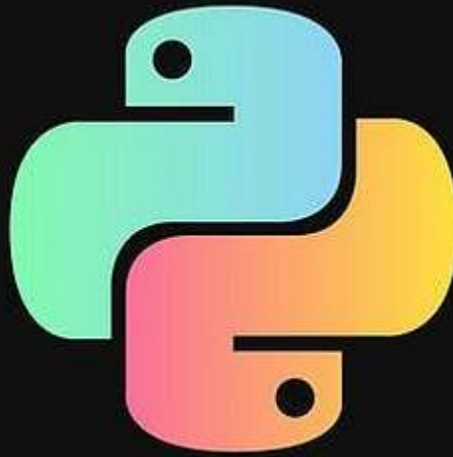
General Coding Knowledge

20 stories · 912 saves



Stories to Help You Grow as a Software Developer

19 stories · 805 saves

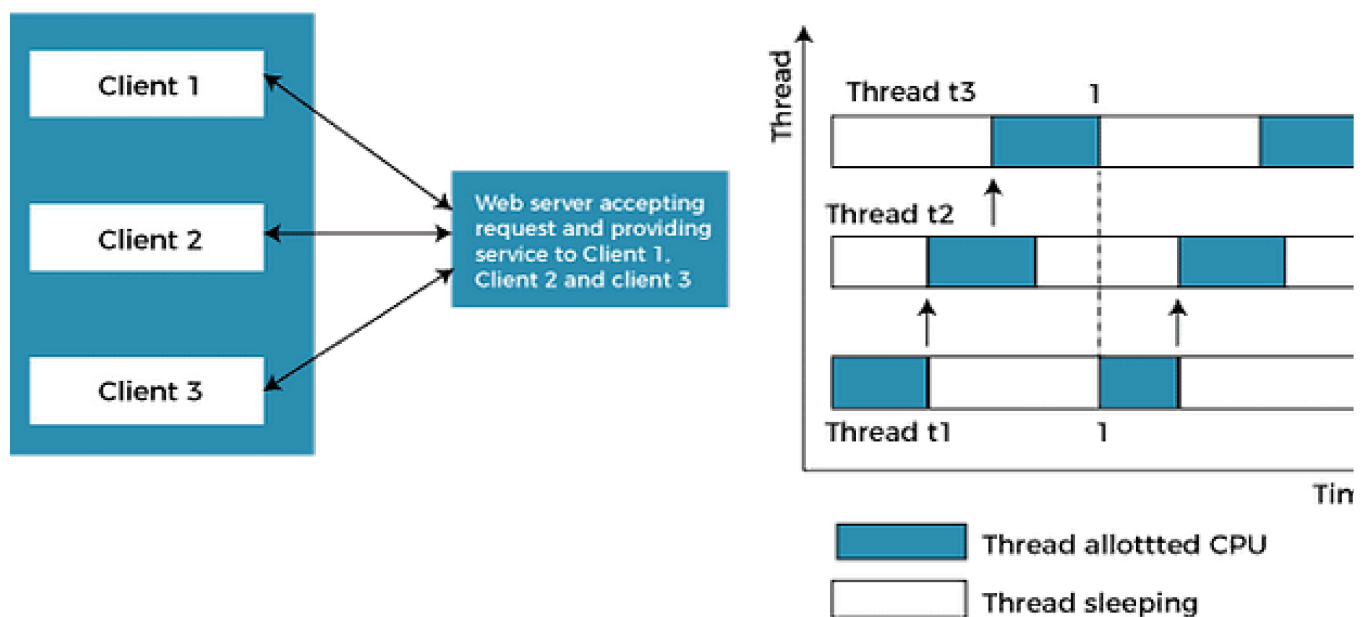


 Tomas Svojanovsky in Python in Plain English

Asyncio: Tasks, Futures

Diving into Asyncio: Tasks, Futures, and Coroutines Explained

🌟 · 4 min read · Sep 7, 2023

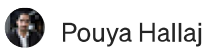


 Suryansh Shrivastava

Concurrency, Asynchronous Programming and Multithreading

Concurrency, asynchronous programming, and multithreading are closely related concepts, often used to optimise performance and...

3 min read · Aug 14, 2023



Pouya Hallaj

Python Asyncio: A Guide to Asynchronous Programming

A Guide to Python's Asynchronous Programming.

4 min read · Sep 18, 2023






 J Kerr

Python: multiprocessing or threading?

A short comparison and walkthrough of a threading + queue template you can use.

2 min read · Sep 27, 2023

 53 

See more recommendations