# The Three Python Concurrency Patterns: Which one to choose?

# Introduction

Name: Iyanuoluwa Ajao

https://twitter.com/iyanuashiri

https://github.com/iyanuashiri

https://iyanuashiri.hashnode.dev

https://theriseofai.substack.com

Software Engineer

# What is Concurrency

Concurrency is when two or more tasks run in an overlapping manner. Examples are Threading(1st) and Asyncio(2nd)

ANALOGY. Watching two movies using two movie players but you pause and play while the first or second one is buffering.

# What is Parallelism

The difference between parallelism and concurrency.

Parallelism is when two or more tasks run at the same time. Most times, the different tasks make use of different CPUs. An example is multiprocessing(3rd Python concurrency pattern). Parallelism is a specific type of concurrency.

ANALOGY. Watching two movies using two movie players without pause and play.

# Problems well suited for concurrency

- CPU-bound problems. Example is mathematical calculations

- IO-bound problems. Example is making a network call


import requests
import time
def get_url(url, session):
with session.get(url) as response:
print(len(response.content))
def get_all_urls(urls):
with requests.Session() as session:
for url in urls:
get_url(url, session)
if __name__ == "__main__":
urls = [
"https://punchng.com",
"http://iyanuashiri.hashnode.dev",
] * 10
start_time = time.time()
get_all_urls(urls)
duration = time.time() - start_time
print(f"Downloaded {len(urls)} in {duration} seconds")

We use a **session** object from **requests** library.

This is a feature in requests. When "*you're making several requests to the same host, the underlying TCP connection will be reused, which can result in a significant performance increase.*"

session = requests.Session()

session.get() == requests.get()

```python
import concurrent.futures
import requests
import threading
import time

thread_local = threading.local()

def get_session():
    if not hasattr(thread_local, "session"):
        thread_local.session = requests.Session()
    return thread_local.session

def get_url(url):
    session = get_session()
    with session.get(url) as response:
        print(len(response.content))

def get_all_urls(urls):
    with concurrent.futures.ThreadPoolExecutor(max_workers=5) as executor:
        futures = [executor.submit(get_url, url) for url in urls]
        for future in concurrent.futures.as_completed(futures):
            future.result()

if __name__ == "__main__":
    urls = [
        "https://punch.ng",
        "http://iyanuashiri.hashnode.dev",
    ] * 10

    start_time = time.time()
    get_all_urls(urls)
    duration = time.time() - start_time
    print(f"Downloaded {len(urls)} sites in {duration:.2f} seconds")
```

# Threading

A thread is a different flow of execution. The threads are not executed at the same time. There are two major differences from the first program.

ThreadPoolExecutor. This creates a Thread, a Pool of threads that run concurrently, and  an Executor that manages how the threads run.

The **with** keyword shows it implements a context manager.

*with open(newfile ‘w’) as file:*

The **with** allows ThreadPoolExecutor to create and free up the threads.

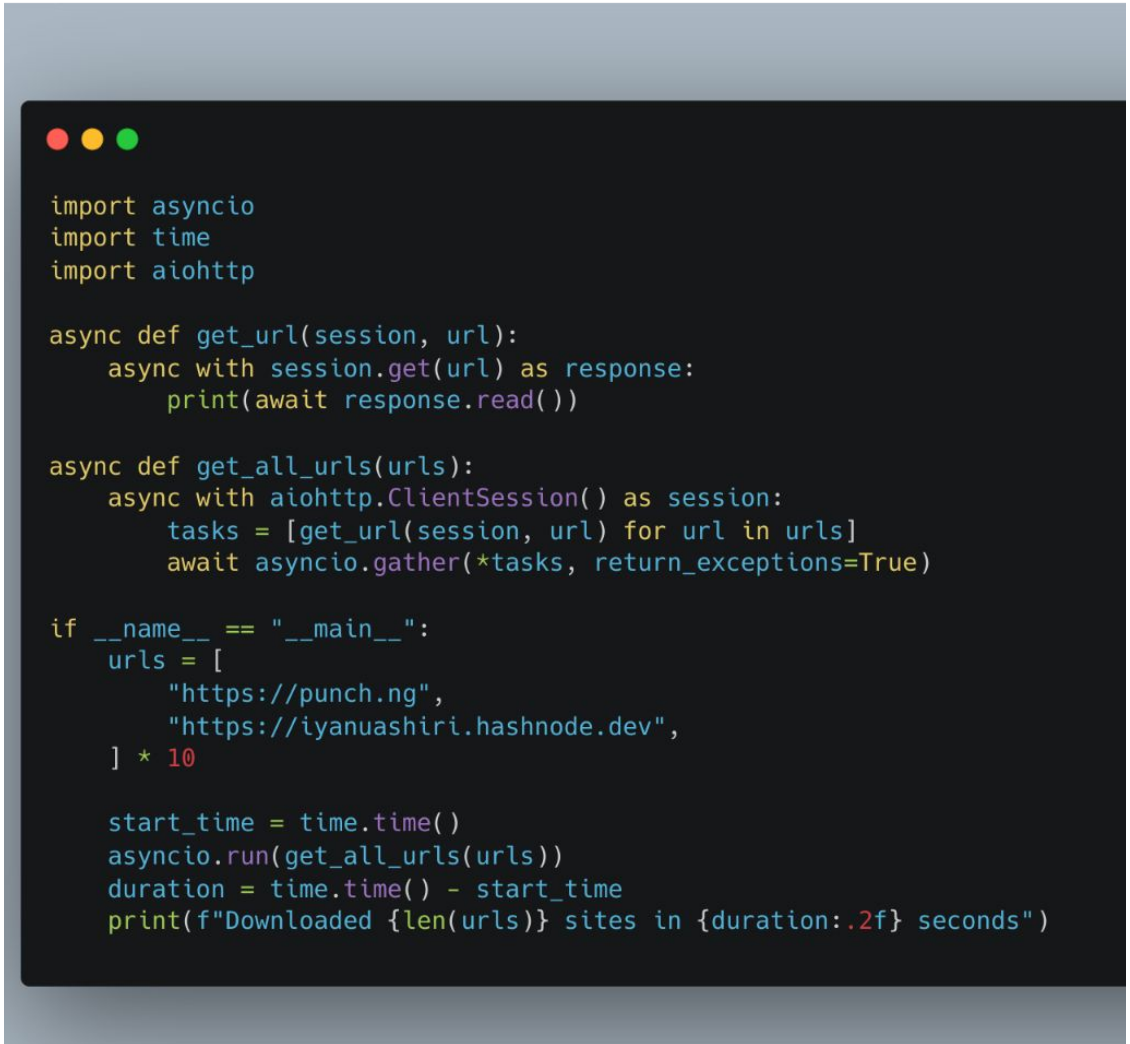
The **get_session()** function for creating a session object.

Creating a session object manually is not thread-safe, so we are employing this method.

This is one of the disadvantages of Threading.

The operating system determines when a task is paused and another is played. So data shared between threads must must be protected or thread-safe.

**requests.Session()** is not thread-safe.



```python
import asyncio
import time
import aiohttp

async def get_url(session, url):
    async with session.get(url) as response:
        print(await response.read())

async def get_all_urls(urls):
    async with aiohttp.ClientSession() as session:
        tasks = [get_url(session, url) for url in urls]
        await asyncio.gather(*tasks, return_exceptions=True)

if __name__ == "__main__":
    urls = [
        "https://punch.ng",
        "https://iyanuashiri.hashnode.dev",
    ] * 10

    start_time = time.time()
    asyncio.run(get_all_urls(urls))
    duration = time.time() - start_time
    print(f"Downloaded {len(urls)} sites in {duration:.2f} seconds")
```

# Asyncio

The major difference between this program and the non-concurrent program is that we ditch **requests** library for **aiohttp.** To take advantage of asyncio, you need async compatible libraries.

**requests 3** is currently being developed with asyncio in mind. But until then, we use **aiohttp**.

We notice the **async/await** keywords for defining the function and the with keyword. **await** means that a task will take a while and it should give up control to the event loop. You will get syntax errors if your function has an **await** in it but not marked with **async.**

**async with** works just like **await** but for creating a context manager.

Asyncio has nothing like thread-safety so we can create a session object here unlike the example in Threading.

**asyncio.run()** # Python 3.7

The **asyncio.get_event_loop()** manages how and when the tasks are run.

```python
import requests
import multiprocessing
import time

session = None


def set_global_session():
    """Initialize a session for each worker process."""
    global session
    if session is None:
        session = requests.Session()


def get_url(url):
    """Fetch the URL content length using a process-wide session."""
    global session
    try:
        with session.get(url) as response:
            print(len(response.content))
    except requests.RequestException as e:
        print(f"Error fetching {url}: {e}")


def get_all_urls(urls):
    """Use multiprocessing pool to fetch all URLs concurrently."""
    with multiprocessing.Pool(initializer=set_global_session) as pool:
        pool.map(get_url, urls)


if __name__ == "__main__":  # Required for Windows compatibility
    urls = [
        "https://punch.ng",
        "http://iyanuashiri.hashnode.dev",
    ] * 10  # Multiply list for testing concurrency

    start_time = time.time()
    get_all_urls(urls)
    duration = time.time() - start_time
    print(f"Downloaded {len(urls)} sites in {duration:.2f} seconds")
```

# Multiprocessing

**multiprocessing.Pool()** creates and can determine the number of pools to create. By default, it creates the equivalent of the number of CPUs in your computer.

# Threading vs Asyncio

- Implementing threads well, you need to understand some other things. E.g Lock to prevent data race in Threads and Queue to manage multiple Threads. This has been abstracted with ThreadPoolExecutor.
- Threads require RAM memory.
- Starting a Thread is expensive.
- Asyncio requires async compatible libraries
- Asyncio uses less resources when compared with Threading.
- Asyncio is faster.
- **Asyncio wins**

# Multiprocessing vs Asyncio or Threading

- Choose Multiprocessing when what you want to do CPU-bound tasks not IO-bound tasks.
- Multiprocessing runs on multiple CPUs.
- Asyncio is when you want to do IO-bound tasks.
- Asyncio and Threading runs on a single CPU.

Thank You!