PYTHON ASYNCIO

DEFINE

Python's asyncio library allows you to run single threaded concurrent code using coroutines using an event loop.

The event loop is designed to I/O over sockets other resources, especially good for working with client/server network connections.

Python >= 3.4(best features and performance in 3.6)

BENEFITS

- The event loop allows you to handle a larger number of network connections at once.
- No connection blocks, so you can have long running connections with very little performance impact(HTML5 sockets for example)

HOW WEB SERVERS TYPICALLY ARE DESIGNED

- (Pyramid, Flash, Plone, etc)
- Processes X Threads = Total number of concurrent connections that can be handled at once
- Client makes a request to web server, request is assigned thread, thread handle request and sends response
- If no threads available, request is blocked, waiting for an open thread
- Threads are expensive(CPU), Processes are expensive on RAM

ASYNCIO WEB SERVER

- All requests are thrown on thread loop
- Since we don't block on network traffic, we can juggle many requests at the same time
- Modern web application servers connect with many different services that can potentially block on network traffic—BAD
- Limiting factor is maxed out CPU, not costly thread switching between requests—GOOD

WHERE DO WE USE NETWORK TRAFFIC?

- Web Client/App Server
- App Server/Database
- App Server/Caching(redis)
- App Server/OAUTH
- App Server/Cloud storage
- App Server/APIs(gdrive, m$, slack, etc)

DETAILS…

- In order to benefit, the whole stack needs to be asyncio aware
- Anywhere in your application server that is not and does network traffic WILL BLOCK all other connections while it is doing it's network traffic(example: using requests library instead of aiohttp)

BASICS

1. Get active event loop or create new one

2. Run coroutine inside event loop with asyncio.run_until_complete

BASICS(CODE)

```
import asyncio



async def hello():
    print('hi')


event_loop = asyncio.get_event_loop()
event_loop.run_until_complete(hello())
```

BASIC CONTINUED

- asyncio.run_until_complete automatically wraps your coroutine into a Future object and waits for it to finish

- asyncio.ensure_future will wrap a coroutine in a future and return it to you

- So you can schedule multiple coroutines that can run at the same time

BASIC CONTINUED(CODE)

```python
import asyncio


async def hello1():
    await asyncio.sleep(0.5)
    print('hi 1')


async def hello2():
    print('hi 2')


event_loop = asyncio.get_event_loop()
future1 = asyncio.ensure_future(hello1(), loop=event_loop)
future2 = asyncio.ensure_future(hello2(), loop=event_loop)
event_loop.run_until_complete(future2)
event_loop.run_until_complete(future1)
```

LONG RUNNING TASKS

- You can also schedule long running tasks on the event loop

- The tasks can run forever…

- "Task" objects and the same as "Future" objects(well, close)

LONG RUNNING TASKS(CODE)

```python
import asyncio
import random


async def hello_many():
    while True:
        number = random.randint(0, 3)
        await asyncio.sleep(number)
        print('Hello {}'.format(number))


event_loop = asyncio.get_event_loop()
task = asyncio.Task(hello_many())
print('task running now...')
event_loop.run_until_complete(asyncio.sleep(10))
print('we waited 10 seconds')
task.cancel()
print('task cancelled')
```

GOTCHA: EVERYTHING MUST BE ASYNC

If you want part of your code to be async(say a function), the complete stack of the caller must be async and running on the event loop

ASYNC EVERYTHING(CODE)

```python
import asyncio


async def print_foobar1():
    print('foobar1')


async def print_foobar2():
    print('foobar2')


async def foobar():
    await print_foobar1()
    print_foobar2()  # won't work, never awaited


event_loop = asyncio.get_event_loop()
event_loop.run_until_complete(foobar())
print_foobar1()  # won't work, never awaited
await print_foobar1()  # error, not running in event loop
```

SINGLE THREADED

- Only 1 event loop can run in a thread at a time
- Running multi-threaded code with asyncio code running in a thread loop is unsafe
- You can multi-(process|thread) and run multiple threads at the same time

```python
import asyncio
import threading


class PrintThread(threading.Thread):

    def __init__(self, id):
        self.id = id
        super().__init__(target=self)

    def __call__(self):
        self._loop = asyncio.new_event_loop()
        self._loop.run_until_complete(self._run())

    async def _run(self):
        for idx in range(3):
            await asyncio.sleep(idx)
            print('Hello {} from thread {}'.format(
                idx, self.id
            ))


threads = []
for i in range(5):
    thread = PrintThread(i)
    threads.append(thread)
    thread.start()
print(f'Waiting to finish')
for thread in threads:
    thread.join()
print('done')
```

"MULTI" PROCESSING IN ASYNCIO

asyncio.gather allows you to run multiple coroutines at the same time, waiting for all of them to finish

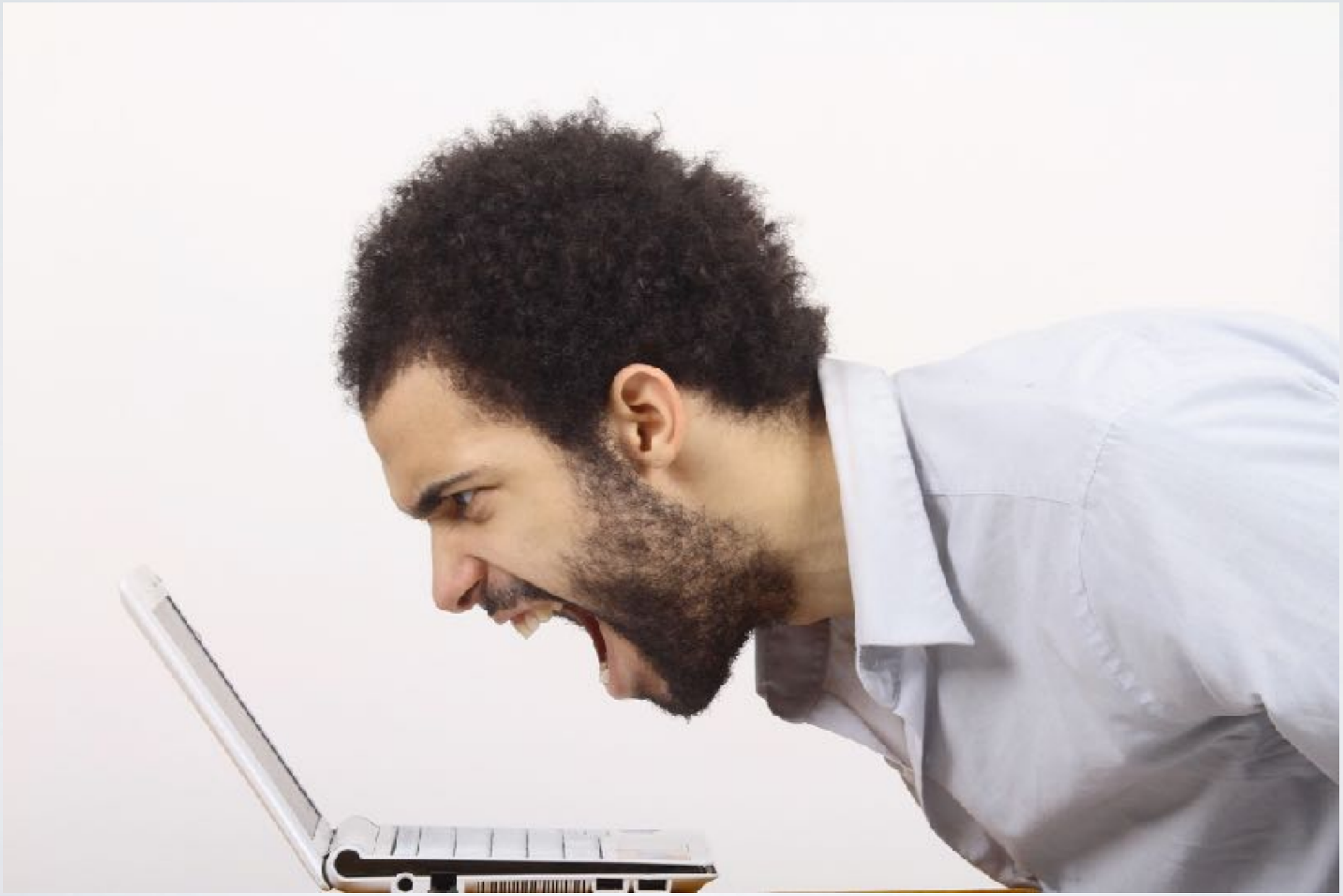
MULTI PROCESS WITH GATHER

```python
import asyncio
import aiohttp


async def download_url(url):
    async with aiohttp.ClientSession() as session:
        resp = await session.get(url)
        text = await resp.text()
        print(f'Downloaded {url}, size {len(text)}')


event_loop = asyncio.get_event_loop()
event_loop.run_until_complete(asyncio.gather(
    download_url('https://www.google.com'),
    download_url('https://www.facebook.com'),
    download_url('https://www.twitter.com'),
    download_url('https://www.stackoverflow.com')
))
```

LOOPS

We can also utilize asyncio in for loops, giving up the loop for every iteration of the loop

IS IT “YELLED”?



OR YIELD

ASYNC LOOP(CODE)

```python
import asyncio


async def yelleding():
    for idx in range(5):
        print(f'Before yelleding {idx}')
        yield idx


async def foobar2():
    async for idx in yelleding():
        print(f"Yay, I've been yelleded {idx}")


event_loop = asyncio.get_event_loop()
event_loop.run_until_complete(foobar2())
```

SCHEDULING

- loop.call_later: arrange to call on a delay
- loop.call_at: arrange function to be called at specified time

SCHEDULING(CODE)

```
import asyncio



def delayed_print(txt):
    print(txt)


event_loop = asyncio.get_event_loop()
event_loop.call_later(1, delayed_print, 'Hello')
event_loop.call_at(event_loop.time() + 1, delayed_print, 'Hello timed')
event_loop.run_until_complete(asyncio.sleep(2))
```

EXECUTORS

- An executor is available to use when you have non-async code that needs to be made async

- A typical executor is a thread executor. This means, anything you run in an executor is being thrown in a thread to run.

- Try to avoid but it's a tool available

- It's worse to have non-async code than to use thread executors

EXECUTORS(CODE)

```
import asyncio
n/misc/asyncio-presentation/basic.py
import concurrent.futures


def download_url(url):
    resp = requests.get(url)
    text = resp.content
    print(f'Downloaded {url}, size {len(text)}')


async def foobar():
    print('foobar')


executor = concurrent.futures.ThreadPoolExecutor(max_workers=5)

event_loop = asyncio.get_event_loop()
event_loop.run_until_complete(asyncio.gather(
    event_loop.run_in_executor(executor, download_url, 'https://www.google.com'),
    event_loop.run_in_executor(executor, download_url, 'https://www.facebook.com'),
    event_loop.run_in_executor(executor, download_url, 'https://www.twitter.com'),
    event_loop.run_in_executor(executor, download_url, 'https://www.stackoverflow.com'),
    foobar()
))
```

SUBPROCESS

```python
import asyncio


async def run_cmd(cmd):
    print(f'Executing: {" ".join(cmd)}')
    process = await asyncio.create_subprocess_exec(*cmd, stdout=asyncio.subprocess.PIPE)
    out, error = await process.communicate()
    print(out.decode('utf8'))


event_loop = asyncio.get_event_loop()
event_loop.run_until_complete(asyncio.gather(
    run_cmd(['sleep', '1']),
    run_cmd(['echo', 'hello'])
))
```

LOOP IMPLEMENTATIONS

- uvloop
- gevent
- eventlet

AIO LIBRARIES

- aiohttp: client and server library
- aioes: elastic search
- asyncpg: postgresql
- aioredis
- aiobotocore
- aiosmtpd: smtp
- and many more. Check out https://github.com/aio-libs

DEBUGGING

- Debugging is more difficult than regular sequential programs
- pdb(debugger) statements do not properly skip over await calls(because thread loop causes debugging to pay attention to something else)
- pdb also causes thread loop to halt right there, there is no way to manually execute task in loop in a pdb prompt either
- Making matters more annoying, python prompt doesn't work with asyncio

DEBUGGING TOOLS

- aioconsole: allows you to have a python prompt with asyncio loop already setup for you. So you can run await statements! Guillotina runs it's shell command in an aioconsole.

- aiomonitor: attach to already running event loop and get info on running tasks. Also integrated with guillotina(run `g -m`)

MORE RESOURCES

- guillotina_hive: not too complicated client/server implementation

- guillotina: full web application built with asyncio technologies. Lots of great examples

- guillotina_rediscache: simple long running task example

QUESTION TIME