



Python 3.5+ Async

An Easier Way to do Concurrency

Concurrency in Python



Concurrency

- Doing lots of slow things at once
- Reasons
 - Single-threaded, single-process, non-event-loop programming is much easier, but also slow
 - Doesn't take advantage of the system resources (RAM, CPU, bandwidth)
- Implementations
 - No Concurrency (no threads, no event loop, single process)
 - Processes
 - Threads
 - Event Loop

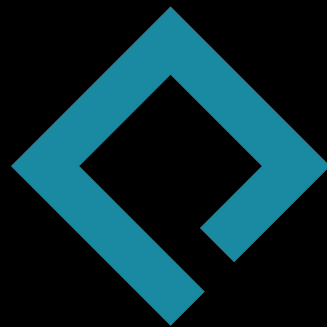
Example: Website Checker

- Check a list of websites to see what status codes they return
 - Classic I/O bound problem
 - Relatively easy to break into concurrent chunks

<https://goo.gl/DivuRA>

Naive Implementation

`naive-checker.py`



No Concurrency Implementation

```
def website_statuses(websites):
    statuses = {}
    for website in websites:
        response = requests.get(website)
        status = response.status_code
        if not statuses.get(status):
            statuses[status] = 0
        statuses[status] += 1
    return statuses

if __name__ == '__main__':
    with open(sys.argv[1], 'r') as f:
        websites = f.readlines()
    t0 = time.time()
    print(json.dumps(website_statuses(websites)))
    t1 = time.time()
    print("getting website statuses took {0:.1f} seconds".format(t1-t0))
```

No Concurrency Implementation

```
def website_statuses(websites):
    statuses = {}
    for website in websites:
        response = requests.get(website)
        status = response.status_code
        if not statuses.get(status):
            statuses[status] = 0
        statuses[status] += 1
    return statuses

if __name__ == '__main__':
    with open(sys.argv[1], 'r') as f:
        websites = f.readlines()
        t0 = time.time()
        print(json.dumps(website_statuses(websites)))
        t1 = time.time()
        print("getting website statuses took {0:.1f} seconds".format(t1-t0))
```

No Concurrency Implementation

```
def website_statuses(websites):
    statuses = {}
    for website in websites:
        response = requests.get(website)
        status = response.status_code
        if not statuses.get(status):
            statuses[status] = 0
        statuses[status] += 1
    return statuses

if __name__ == '__main__':
    with open(sys.argv[1], 'r') as f:
        websites = f.readlines()
    t0 = time.time()
    print(json.dumps(website_statuses(websites)))
    t1 = time.time()
    print("getting website statuses took {0:.1f} seconds".format(t1-t0))
```


No Concurrency Implementation

```
$ python3 naive-checker.py list.txt  
{"200": 31}  
getting website statuses took 35.1 seconds
```

Pros

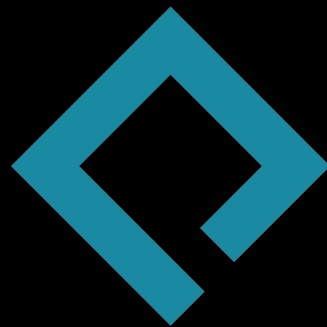
- No splitting/combining work
- No locks/semaphores
- Easy to code

Cons

- Slow to execute

Multi-Process Implementation

`subprocess-checker.py`



Concurrency Using Processes

```
if __name__ == '__main__':
    with open(sys.argv[1], 'r') as f:
        websites = f.readlines()
        number_of_processes = int(sys.argv[2])
        per_process = math.ceil(len(websites) / number_of_processes)
        # split up the work based on number of processes
        ...
        t0 = time.time()
        processes = []
        for i in range(number_of_processes):
            p = subprocess.Popen(
                ["python3", "naive-checker.py", "/tmp/list-{}.txt".format(i)],
                stdout=subprocess.PIPE)
            processes.append(p)
        # gather the results
        ...
        print(combined)
        t1 = time.time()
        print("getting website statuses took {0:.1f} seconds".format(t1-t0))
```

Concurrency Using Processes

```
if __name__ == '__main__':
    with open(sys.argv[1], 'r') as f:
        websites = f.readlines()
        number_of_processes = int(sys.argv[2])
        per_process = math.ceil(len(websites) / number_of_processes)
        # split up the work based on number of processes
        ...
    t0 = time.time()
    processes = []
    for i in range(number_of_processes):
        p = subprocess.Popen(
            ["python3", "naive-checker.py", "/tmp/list-{}.txt".format(i)],
            stdout=subprocess.PIPE)
        processes.append(p)
    # gather the results
    ...
    print(combined)
    t1 = time.time()
    print("getting website statuses took {0:.1f} seconds".format(t1-t0))
```

Concurrency Using Processes

```
if __name__ == '__main__':  
    with open(sys.argv[1], 'r') as f:  
        websites = f.readlines()  
        number_of_processes = int(sys.argv[2])  
        per_process = math.ceil(len(websites) / number_of_processes)  
        # split up the work based on number of processes  
        ...  
        t0 = time.time()  
        processes = []  
        for i in range(number_of_processes):  
            p = subprocess.Popen(  
                ["python3", "naive-checker.py", "/tmp/list-{}.txt".format(i)],  
                stdout=subprocess.PIPE)  
            processes.append(p)  
        # gather the results  
        ...  
        print(combined)  
        t1 = time.time()  
        print("getting website statuses took {0:.1f} seconds".format(t1-t0))
```

Concurrency Using Processes

```
if __name__ == '__main__':
    with open(sys.argv[1], 'r') as f:
        websites = f.readlines()
        number_of_processes = int(sys.argv[2])
        per_process = math.ceil(len(websites) / number_of_processes)
        # split up the work based on number of processes
        ...
        t0 = time.time()
        processes = []
        for i in range(number_of_processes):
            p = subprocess.Popen(
                ["python3", "naive-checker.py", "/tmp/list-{}.txt".format(i)],
                stdout=subprocess.PIPE)
            processes.append(p)
        # gather the results
        ...
        print(combined)
        t1 = time.time()
        print("getting website statuses took {0:.1f} seconds".format(t1-t0))
```

Concurrency Using Processes

```
$ python3 subprocess-checker.py list.txt 3  
{'200': 31}  
getting website statuses took 9.6 seconds
```

Pros

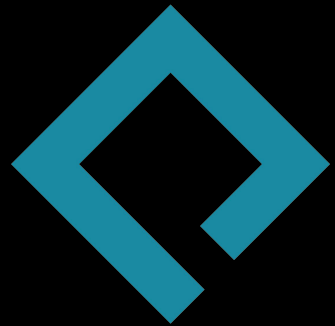
- Faster!
- No locks/semaphores

Cons

- No shared memory
- Have to split/combine data yourself
- Tricky to code

Multi-Thread Implementation

`thread-checker.py`



Concurrency Using Threads

```
STATS = {}  
  
def get_website_status(url, lock):  
    response = requests.get(url)  
    status = response.status_code  
    if status != 200:  
        print(url)  
    lock.acquire()  
    if not STATS.get(status):  
        STATS[status] = 0  
    STATS[status] += 1  
    lock.release()  
  
if __name__ == '__main__':  
    ...  
    threads = []  
    lock = threading.Lock()  
    for website in websites:  
        t = threading.Thread(target=get_website_status, args=(website, lock))  
        threads.append(t)  
        t.start()  
    for t in threads:  
        t.join()  
    t1 = time.time()  
    print(json.dumps(STATS))  
    print("getting website statuses took {0:.1f} seconds".format(t1-t0))
```

Concurrency Using Threads

```
STATS = {}  
  
def get_website_status(url, lock):  
    response = requests.get(url)  
    status = response.status_code  
    if status != 200:  
        print(url)  
    lock.acquire()  
    if not STATS.get(status):  
        STATS[status] = 0  
    STATS[status] += 1  
    lock.release()  
  
if __name__ == '__main__':  
    ...  
    threads = []  
    lock = threading.Lock()  
    for website in websites:  
        t = threading.Thread(target=get_website_status, args=(website, lock))  
        threads.append(t)  
        t.start()  
    for t in threads:  
        t.join()  
    t1 = time.time()  
    print(json.dumps(STATS))  
    print("getting website statuses took {0:.1f} seconds.format(t1-t0))
```

Concurrency Using Threads

```
STATS = {}

def get_website_status(url, lock):
    response = requests.get(url)
    status = response.status_code
    if status != 200:
        print(url)
    lock.acquire()
    if not STATS.get(status):
        STATS[status] = 0
    STATS[status] += 1
    lock.release()

if __name__ == '__main__':
    ...
    threads = []
    lock = threading.Lock()
    for website in websites:
        t = threading.Thread(target=get_website_status, args=(website, lock))
        threads.append(t)
        t.start()
    for t in threads:
        t.join()
    t1 = time.time()
    print(json.dumps(STATS))
    print("getting website statuses took {0:.1f} seconds.format(t1-t0))
```

Concurrency Using Threads

```
STATS = {}  
def get_website_status(url, lock):  
    response = requests.get(url)  
    status = response.status_code  
    if status != 200:  
        print(url)  
    lock.acquire()  
    if not STATS.get(status):  
        STATS[status] = 0  
    STATS[status] += 1  
    lock.release()  
if __name__ == '__main__':  
    ...  
    threads = []  
    lock = threading.Lock()  
    for website in websites:  
        t = threading.Thread(target=get_website_status, args=(website, lock))  
        threads.append(t)  
        t.start()  
    for t in threads:  
        t.join()  
    t1 = time.time()  
    print(json.dumps(STATS))  
    print("getting website statuses took {0:.1f} seconds.format(t1-t0))
```

Concurrency Using Threads

```
$ python3 thread-checker.py list.txt  
{"200": 31}  
getting website statuses took 5.7 seconds
```

Pros

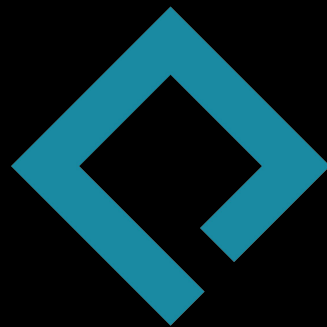
- Faster!
- Shared Memory

Cons

- Locks/Semaphores have to be managed
- Tricky to code
- GIL means you're not really using extra CPU power

Event Loop Implementation

`asyncio-checker.py`



Concurrency Using Async/Event Loop

```
async def get_statuses(websites):
    statuses = {}
    tasks = [get_website_status(website) for website in websites]
    for status in await asyncio.gather(*tasks):
        if not statuses.get(status):
            statuses[status] = 0
        statuses[status] += 1
    print(json.dumps(statuses))

async def get_website_status(url):
    response = await aiohttp.get(url)
    status = response.status
    response.close()
    return status

if __name__ == '__main__':
    with open(sys.argv[1], 'r') as f:
        websites = f.readlines()
    t0 = time.time()
    loop = asyncio.get_event_loop()
    loop.run_until_complete(get_statuses(websites))
    t1 = time.time()
    print("getting website statuses took {0:.1f} seconds".format(t1-t0))
```

Concurrency Using Async/Event Loop

```
async def get_statuses(websites):
    statuses = {}
    tasks = [get_website_status(website) for website in websites]
    for status in await asyncio.gather(*tasks):
        if not statuses.get(status):
            statuses[status] = 0
        statuses[status] += 1
    print(json.dumps(statuses))

async def get_website_status(url):
    response = await aiohttp.get(url)
    status = response.status
    response.close()
    return status

if __name__ == '__main__':
    with open(sys.argv[1], 'r') as f:
        websites = f.readlines()
        t0 = time.time()
        loop = asyncio.get_event_loop()
        loop.run_until_complete(get_statuses(websites))
        t1 = time.time()
        print("getting website statuses took {0:.1f} seconds".format(t1-t0))
```


Concurrency Using Async/Event Loop

```
async def get_statuses(websites):
    statuses = {}
    tasks = [get_website_status(website) for website in websites]
    for status in await asyncio.gather(*tasks):
        if not statuses.get(status):
            statuses[status] = 0
        statuses[status] += 1
    print(json.dumps(statuses))

async def get_website_status(url):
    response = await aiohttp.get(url)
    status = response.status
    response.close()
    return status

if __name__ == '__main__':
    with open(sys.argv[1], 'r') as f:
        websites = f.readlines()
    t0 = time.time()
    loop = asyncio.get_event_loop()
    loop.run_until_complete(get_statuses(websites))
    t1 = time.time()
    print("getting website statuses took {0:.1f} seconds".format(t1-t0))
```

Concurrency Using Async/Event Loop

```
async def get_statuses(websites):
    statuses = {}
    tasks = [get_website_status(website) for website in websites]
    for status in await asyncio.gather(*tasks):
        if not statuses.get(status):
            statuses[status] = 0
        statuses[status] += 1
    print(json.dumps(statuses))
```

```
async def get_website_status(url):
    response = await aiohttp.get(url)
    status = response.status
    response.close()
    return status
```

```
if __name__ == '__main__':
    with open(sys.argv[1], 'r') as f:
        websites = f.readlines()
    t0 = time.time()
    loop = asyncio.get_event_loop()
    loop.run_until_complete(get_statuses(websites))
    t1 = time.time()
    print("getting website statuses took {:.1f} seconds".format(t1-t0))
```

Concurrency Using Async/Event Loop

```
async def get_statuses(websites):
    statuses = {}
    tasks = [get_website_status(website) for website in websites]
    for status in await asyncio.gather(*tasks):
        if not statuses.get(status):
            statuses[status] = 0
        statuses[status] += 1
    print(json.dumps(statuses))

async def get_website_status(url):
    response = await aiohttp.get(url)
    status = response.status
    response.close()
    return status

if __name__ == '__main__':
    with open(sys.argv[1], 'r') as f:
        websites = f.readlines()
    t0 = time.time()
    loop = asyncio.get_event_loop()
    loop.run_until_complete(get_statuses(websites))
    t1 = time.time()
    print("getting website statuses took {0:.1f} seconds".format(t1-t0))
```

Concurrency Using Async/Event Loop

```
async def get_statuses(websites):
    statuses = {}
    tasks = [get_website_status(website) for website in websites]
    for status in await asyncio.gather(*tasks):
        if not statuses.get(status):
            statuses[status] = 0
        statuses[status] += 1
    print(json.dumps(statuses))

async def get_website_status(url):
    response = await aiohttp.get(url)
    status = response.status
    response.close()
    return status

if __name__ == '__main__':
    with open(sys.argv[1], 'r') as f:
        websites = f.readlines()
    t0 = time.time()
    loop = asyncio.get_event_loop()
    loop.run_until_complete(get_statuses(websites))
    t1 = time.time()
    print("getting website statuses took {0:.1f} seconds".format(t1-t0))
```

Concurrency Using Async/Event Loop

```
$ python3 asyncio-checker.py list.txt  
{"200": 31}  
getting website statuses took 2.8 seconds
```

Pros

- Faster!
- Shared Memory
- Utilize Extra Processors
(`run_in_executor`)
- Easier to code

Cons

- Not as many supported libraries
- Still harder than the naive approach

Other Concerns



Event Loop Considerations

- Python avoids callback hell using generators
- Very good for networking protocols (managing socket connections)
- Nginx vs Apache

Event Loop Implementations

- Pre-3.5
 - Twisted
 - Gevent
- 3.5+
 - Asyncio
 - Curio

Questions?

Email: jimmy.song@paxos.com

Twitter: @jimmysong

LinkedIn: <http://linkedin.com/in/jimmysong>

