Python Concurrency

Multiprocessing vs Threading

Ackwonledgement: https://nyu-cds.github.io/python-concurrency/

Task: fetch images from Imgur

```
import json
import os
from pathlib import Path
from urllib.request import urlopen, Request

types = {'image/jpeg', 'image/png', 'image/gif'}

def get_links(client_id):
    headers = {'Authorization': 'Client-ID {}'.format(client_id)}
    req = Request('https://api.imgur.com/3/gallery/random/random/', headers=headers, method='GET')
    with urlopen(req) as resp:
        data = json.loads(resp.read().decode('utf-8'))
    return [item['link'] for item in data['data'] if 'type' in item and item['type'] in types]

def download_link(directory, link):
    download_path = directory / os.path.basename(link)
    with urlopen(link) as image, download_path.open('wb') as f:
        f.write(image.read())

def setup_download_dir():
    download_dir = Path('images')
    if not download_dir.exists():
        download_dir.mkdir()
    return download_dir
```

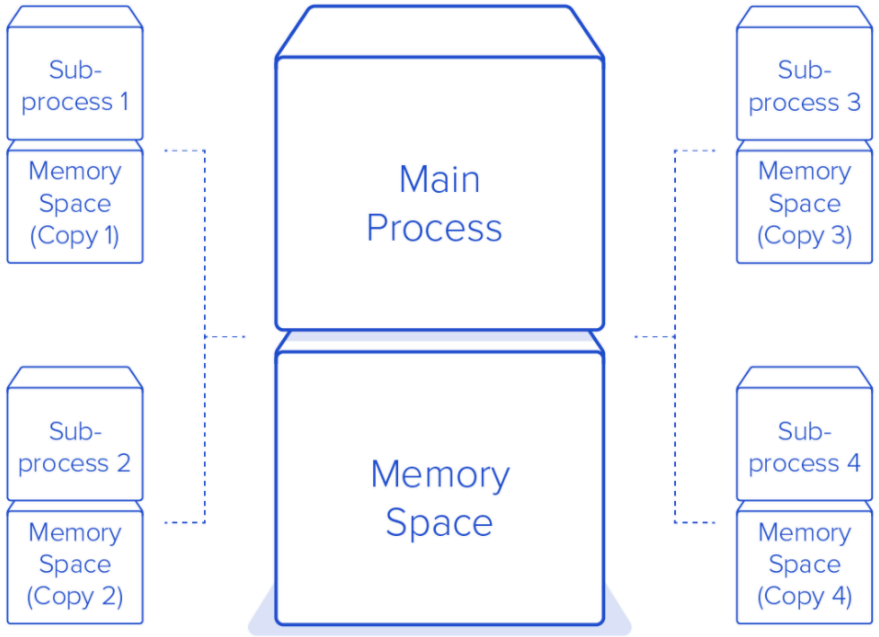
Python multiprocessing module

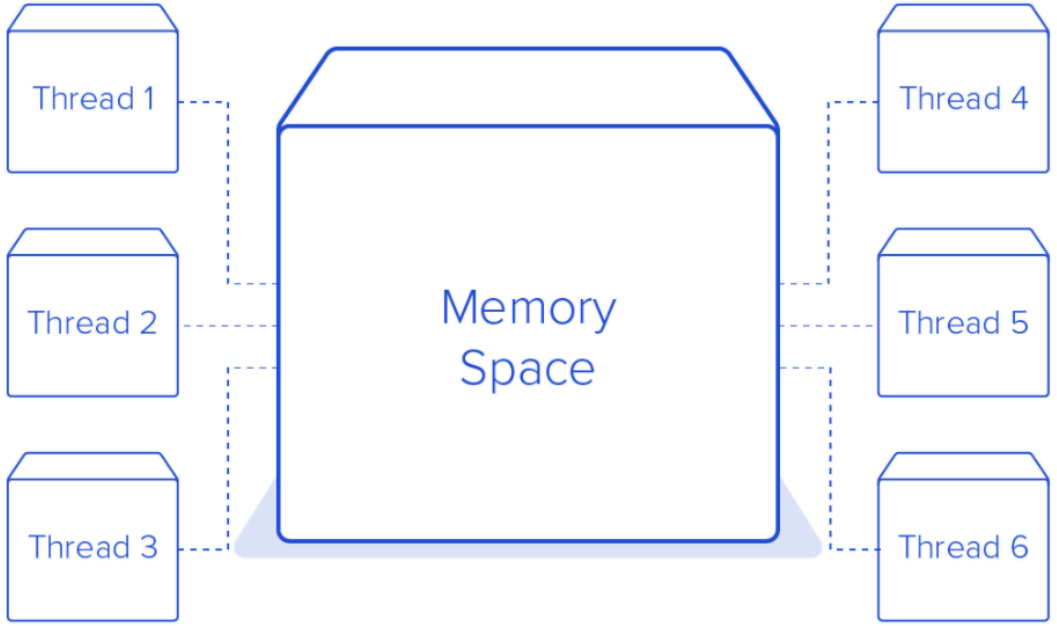
```
from time import time
from functools import partial
from multiprocessing.pool import Pool

from download import setup_download_dir, get_links, download_link

CLIENT_ID = 'replace with your client ID'

def main():
   ts = time()
   download_dir = setup_download_dir()
   links = [l for l in get_links(CLIENT_ID)]
   download = partial(download_link, download_dir)
   with Pool(8) as p:
       p.map(download, links)
   print('Took {}s'.format(time() - ts))

if __name__ == '__main__':
   main()
```

multiprocessing vs threading

Process	Thread
processes run in separate memory (process isolation)	threads share memory
uses more memory	uses less memory
more overhead	less overhead
slower to create and destroy	faster to create and destroy
easier to code and debug	can become harder to code and debug

multiprocessing vs threading





Python threading: **start()**

```python
import logging
import threading
import time

def thread_function(name):
    logging.info("Thread %s: starting", name)
    time.sleep(2)
    logging.info("Thread %s: finishing", name)

if __name__ == "__main__":
    format = "%(asctime)s: %(message)s"
    logging.basicConfig(format=format, level=logging.INFO,
                        datefmt="%H:%M:%S")

    logging.info("Main    : before creating thread")
    x = threading.Thread(target=thread_function, args=(1,))
    logging.info("Main    : before running thread")
    x.start()
    logging.info("Main    : wait for the thread to finish")
    # x.join()
    logging.info("Main    : all done")
```

```
$ ./single_thread.py
Main    : before creating thread
Main    : before running thread
Thread 1: starting
Main    : wait for the thread to finish
Main    : all done
Thread 1: finishing
```

You'll notice that the `Thread` finished after the `Main` section of your code did.

Python threading: **daemon, join()**

```python
import logging
import threading
import time

def thread_function(name):
    logging.info("Thread %s: starting", name)
    time.sleep(2)
    logging.info("Thread %s: finishing", name)

if __name__ == "__main__":
    format = "%(asctime)s: %(message)s"
    logging.basicConfig(format=format, level=logging.INFO,
                        datefmt="%H:%M:%S")

    logging.info("Main    : before creating thread")
    x = threading.Thread(target=thread_function, args=(1,), daemon=True)
    logging.info("Main    : before running thread")
    x.start()
    logging.info("Main    : wait for the thread to finish")
    # x.join()
    logging.info("Main    : all done")
```

```
$ ./daemon_thread.py
Main    : before creating thread
Main    : before running thread
Thread 1: starting
Main    : wait for the thread to finish
Main    : all done
```

The difference here is that the final line of the output is missing. `thread_function()` did not get a chance to complete. It was a `daemon` thread, so when `__main__` reached the end of its code and the program wanted to finish, the daemon was killed.

Working with multiple threads

```python
import logging
import threading
import time

def thread_function(name):
    logging.info("Thread %s: starting", name)
    time.sleep(2)
    logging.info("Thread %s: finishing", name)

if __name__ == "__main__":
    format = "%(asctime)s: %(message)s"
    logging.basicConfig(format=format, level=logging.INFO,
                        datefmt="%H:%M:%S")

    threads = list()
    for index in range(3):
        logging.info("Main    : create and start thread %d.", index)
        x = threading.Thread(target=thread_function, args=(index,))
        threads.append(x)
        x.start()

    for index, thread in enumerate(threads):
        logging.info("Main    : before joining thread %d.", index)
        thread.join()
        logging.info("Main    : thread %d done", index)
```

```
$ ./multiple_threads.py
Main    : create and start thread 0.
Thread 0: starting
Main    : create and start thread 1.
Thread 1: starting
Main    : create and start thread 2.
Thread 2: starting
Main    : before joining thread 0.
Thread 2: finishing
Thread 1: finishing
Thread 0: finishing
Main    : thread 0 done
Main    : before joining thread 1.
Main    : thread 1 done
Main    : before joining thread 2.
Main    : thread 2 done
```

If you walk through the output carefully, you'll see all three threads getting started in the order you might expect, but in this case they finish in the opposite order! Multiple runs will produce different orderings. Look for the `Thread x: finishing` message to tell you when each thread is done.

Deadlocks

Five philosophers are seated on a round table with five plates of spaghetti (a type of pasta) and five forks, as shown in the diagram.

At any given time, a philosopher must either be eating or thinking.

Moreover, a philosopher must take the two forks adjacent to him (i.e., the left and right forks) before he can eat the spaghetti. The problem of deadlock occurs when all five philosophers pick up their right forks simultaneously.

Since each of the philosophers has one fork, they will all wait for the others to put their fork down. As a result, none of them will be able to eat spaghetti.

Similarly, in a concurrent system, a deadlock occurs when different threads or processes (philosophers) try to acquire the shared system resources (forks) at the same time. As a result, none of the processes get a chance to execute as they are waiting for another resource held by some other process.

Dining Philosophers Problem

Race conditions

A race condition is an unwanted state of a program which occurs when a system performs two or more operations simultaneously. For example, consider this simple for loop:

```
i=0; # a global variable
for x in range(100):
    print(i)
    i+=1;
```

If you create **n** number of threads which run this code at once, you cannot determine the value of i (which is shared by the threads) when the program finishes execution. This is because in a real multithreading environment, the threads can overlap, and the value of i which was retrieved and modified by a thread can change in between when some other thread accesses it.

Python threading: **Lock**

To deal with race conditions, deadlocks, and other thread-based issues, the threading module provides the **Lock** object. The idea is that when a thread wants access to a specific resource, it acquires a lock for that resource. Once a thread locks a particular resource, no other thread can access it until the lock is released. As a result, the changes to the resource will be atomic, and race conditions will be averted.

```
import threading

# global variable x
x = 0

def increment():
    """
    function to increment global variable x
    """
    global x
    x += 1

def thread_task(lock):
    """
    task for thread
    calls increment function 100000 times.
    """
    for _ in range(100000):
        lock.acquire()
        increment()
        lock.release()

def main_task():
    global x
    # setting global variable x as 0
    x = 0

    # creating a lock
    lock = threading.Lock()

    # creating threads
    t1 = threading.Thread(target=thread_task, args=(lock,))
    t2 = threading.Thread(target=thread_task, args=(lock,))

    # start threads
    t1.start()
    t2.start()

    # wait until threads finish their job
    t1.join()
    t2.join()

if __name__ == "__main__":
    for i in range(10):
        main_task()
        print("Iteration {0}: x = {1}".format(i,x))
```

Global Interpreter Lock (GIL)

Global Interpreter Lock (GIL) in python is a process lock or a mutex used while dealing with the processes. It makes sure that one thread can access a particular resource at a time and it also prevents the use of objects and bytecodes at once. This benefits the single-threaded programs in a performance increase. GIL in python is very simple and easy to implement.

A lock can be used to make sure that only one thread has access to a particular resource at a given time.

One of the features of Python is that it uses a global lock on each interpreter process, which means that every process treats the python interpreter itself as a resource.

For example, suppose you have written a python program which uses two threads to perform both CPU and 'I/O' operations. When you execute this program, this is what happens:

1. The python interpreter creates a new process and spawns the threads
2. When thread-1 starts running, it will first acquire the GIL and lock it.
3. If thread-2 wants to execute now, it will have to wait for the GIL to be released even if another processor is free.
4. Now, suppose thread-1 is waiting for an I/O operation. At this time, it will release the GIL, and thread-2 will acquire it.
5. After completing the I/O ops, if thread-1 wants to execute now, it will again have to wait for the GIL to be released by thread-2.

Python threading: **ThreadPoolExecutor**

```
import concurrent.futures

# [rest of code]

if __name__ == "__main__":
    format = "%(asctime)s: %(message)s"
    logging.basicConfig(format=format, level=logging.INFO,
                        datefmt="%H:%M:%S")

    with concurrent.futures.ThreadPoolExecutor(max_workers=3) as executor:
        executor.map(thread_function, range(3))
```

The code creates a `ThreadPoolExecutor` as a context manager, telling it how many worker threads it wants in the pool. It then uses `.map()` to step through an iterable of things, in your case `range(3)`, passing each one to a thread in the pool.

The end of the `with` block causes the `ThreadPoolExecutor` to do a `.join()` on each of the threads in the pool. It is *strongly* recommended that you use `ThreadPoolExecutor` as a context manager when you can so that you never forget to `.join()` the threads.

Python threading: **ThreadPoolExecutor**

```python
from concurrent.futures import ThreadPoolExecutor
from functools import partial
from time import time

from download import setup_download_dir, get_links, download_link

CLIENT_ID = 'IMGUR_CLIENT_ID'


def main():
    ts = time()
    download_dir = setup_download_dir()
    links = get_links(CLIENT_ID)
    # By placing the executor inside a with block, the executors shutdown method
    # will be called cleaning up threads.

    with ThreadPoolExecutor() as executor:

        # Create a new partially applied function that stores the directory
        # argument.
        #
        # This allows the download_link function that normally takes two
        # arguments to work with the map function that expects a function of a
        # single argument.
        fn = partial(download_link, download_dir)

        # Executes fn concurrently using threads on the links iterable. The
        # timeout is for the entire process, not a single call, so downloading
        # all images must complete within 30 seconds.
        executor.map(fn, links, timeout=30)

    print(len(links), ' images', 'took the average time: {}s'.format((time() - ts) / len(links)))


if __name__ == '__main__':
    main()
```

Python multiprocess: **ProcessPoolExecutor**

```python
import logging
from pathlib import Path
from time import time
from functools import partial
from concurrent.futures import ProcessPoolExecutor
from PIL import Image

logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(name)s - %(levelname)s - %(message)s')
logger = logging.getLogger(__name__)
types = {'.jpg', '.png', '.gif'}


def create_thumbnail(size, path):
    print(path)
    path = Path(path)
    name = path.stem + '_thumbnail' + path.suffix
    thumbnail_path = path.with_name(name)
    image = Image.open(path)
    image.thumbnail(size)
    image.save(thumbnail_path)


def main():
    ts = time()
    # Partially apply the create_thumbnail method, setting the size to 128x128
    # and returning a function of a single argument.
    thumbnail_128 = partial(create_thumbnail, (128, 128))
    # Create the executor in a with block so shutdown is called when the block
    # is exited.
    paths = []
    for image_path in Path('images').iterdir():
        if image_path.suffix in types:
            paths.append(image_path)
    with ProcessPoolExecutor() as executor:
        executor.map(thumbnail_128, paths)
    logging.info('Took %s', time() - ts)


if __name__ == '__main__':
    main()
```

Message passing with Queue

```
import threading, queue

q = queue.Queue()

def worker():
    while True:
        item = q.get()
        print(f'Working on {item}')
        print(f'Finished {item}')
        q.task_done()

# turn-on the worker thread
threading.Thread(target=worker, daemon=True).start()

# send thirty task requests to the worker
for item in range(30):
    q.put(item)
print('All task requests sent\n', end='')

# block until all tasks are done
q.join()
print('All work completed')
```

Python threading with Queue

```
from time import time
from queue import Queue
from threading import Thread

from download import setup_download_dir, get_links, download_link

CLIENT_ID = 'replace with your client ID'

class DownloadWorker(Thread):
    def __init__(self, queue):
        Thread.__init__(self)
        self.queue = queue

    def run(self):
        while True:
            # Get the work from the queue and expand the tuple
            directory, link = self.queue.get()
            download_link(directory, link)
            self.queue.task_done()

def main():
    ts = time()
    download_dir = setup_download_dir()
    links = [l for l in get_links(CLIENT_ID)]
    # Create a queue to communicate with the worker threads
    queue = Queue()
    # Create 8 worker threads
    for x in range(8):
        worker = DownloadWorker(queue)
        # Setting daemon to True will let the main thread exit even though the workers are blocking
        worker.daemon = True
        worker.start()
    # Put the tasks into the queue as a tuple
    for link in links:
        print('Queueing {}'.format(link))
        queue.put((download_dir, link))
    # Causes the main thread to wait for the queue to finish processing all the tasks
    queue.join()
    print('Took {}'.format(time() - ts))

if __name__ == '__main__':
    main()
```

Python Multiprocess with queue

simple_queue2.py

```
#!/usr/bin/python

from multiprocessing import Queue, Process, current_process


def worker(queue):
    name = current_process().name
    print(f'{name} data received: {queue.get()}')


def main():

    queue = Queue()
    queue.put("wood")
    queue.put("sky")
    queue.put("cloud")
    queue.put("ocean")

    processes = [Process(target=worker, args=(queue,)) for _ in range(4)]

    for p in processes:
        p.start()

    for p in processes:
        p.join()

if __name__ == "__main__":
    main()
```

```
$ ./simple_queue2.py
Process-1 data received: wood
Process-2 data received: sky
Process-3 data received: cloud
Process-4 data received: ocean
```