Practical 1 - Parallel programming in Python with base libraries

Software prerequisites for this practical:

- Python3

A few notes:

- During this practical, we will use Python modules. That might be confusing, so if you need to get a refresher on how Python imports / hierarchies work, a good resource can be found here: http://python-notes.curiousefficiency.org/en/latest/python_concepts/import_traps.html
- We will also use the Python profiling. More information about that can be found here: https://docs.python.org/3/library/profile.html
- Lastly, we will use decorators. More information about that can be found here: https://wiki.python.org/moin/PythonDecorators

Before performing any optimizations, we always need to understand what the bottlenecks are, how can we get the most benefits with the least code changes. Therefore, even if our toy examples we will use a profiling decorator that will give us insights into the behaviour of the samples.

As mentioned in the lecture, Python 3 has rather sophisticated mechanism for managing concurrency by using processes and threads. Without the help of additional (3rd party) libraries, the possible ways to implement these behaviours are:

- subprocess - spawn additional processes (available since python 2)
- signal - asynchronous system events (available since python 2)
- threading - concurrent operations within a process (available since python 2)
- queues - used for first-in-first out or last-in-last-out stack-like implementations (available since python2, but reworked slightly in python 3)
- multiprocessing - manage processes as threads (available since python 2.6)
- concurrent.futures - pools of concurrent tasks (available since python 2.7)
- asyncio - asynchronous I/O, event loop, and concurrency tools (available only in python 3, the code we will work on requires python 3.5)

In order to measure the time elapsed in various functions, we will define the following function based on the cProfile module:

```
import cProfile

def do_profile(func):
    def profiled_func(*args, **kwargs):
        profile = cProfile.Profile()
        try:
            profile.enable()
            result = func(*args, **kwargs)
            profile.disable()
            return result
        finally:
            profile.print_stats()
    return profiled_func
```

By decorating functions with @do_profile, we will analyse the behaviour of our code.

Note: most of the examples presented here are taken from the pymotw website or the official Python documentation page.

Running interactively on Cartesius

Cartesius uses slurm - a batch job scheduling system. The basic way to operate it is to create a submission script that contains the normal execution command you would normally use to run your software wrapped in some directives that define what resources are needed for the execution.

For example:

python myprogram.py

becomes a script containing:

#!/bin/sh
#SBATCH -N 1 #this means we request for 1 node
#SBATCH -p broadwell #we request a node from the Broadwell partition
#SBATCH -t 1:00:00 #we define our walltime to be 1 hour

srun python myprogram.py #here you could just do python myprogram.py, however by using srun you are using a wrapper over mpi that ensures you will run your command on all nodes. This is irrelevant for the case presented since we use only one node, however we can easily modify our script to illustrate this behaviour: change the -N parameter to 2 and in the "execution" part of it, change your command from *srun python myprogram.py* to *hostname* and then to *srun hostname*.

However, for ease, we can use interactive jobs, so the whole previous example becomes:
srun -N 1 -p broadwell -t 1:00:00 python myprogram.py

This will however lock your console and doesn't allow for you to load a different python than the system default one.

In our practical we will use:
salloc -N 1 -p normal -t 2:00:00

Then in order to see what node was allocated:
squeue -u $(whoami)

Resulting in something like:

```
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
           3199187    normal     bash   damian  R       1:18      1 tcn1131
```

Than we ssh to the reported node, for example:
ssh tcn1131

Once on the node we should do:
module load python/3.5.2-intel-u2

And now we're good to go….

Subprocess

This is a subset of the examples presented here: https://pymotw.com/3/subprocess/index.html. For more detailed information, please check the PyMOTW-3 website.

A simple example of getting a feel about what subprocess does is to run an external command without interacting with it:

```
import subprocess
completed = subprocess.run(['ls', '-lahF'])
print('returncode:', completed.returncode)
```

The command line arguments are passed as a list of strings, which avoids the need for escaping quotes or other special characters that might be interpreted by the shell. run() returns a CompletedProcess instance, with information about the process like the exit code and output.
The returncode attribute of the CompletedProcess is the exit code of the program. The caller is responsible for interpreting it to detect errors. If the check argument to run() is True, the exit code is checked and if it indicates an error happened then a CalledProcessError exception is raised.

```
import subprocess

try:
    subprocess.run(['false'], check=True)
except subprocess.CalledProcessError as err:
    print('ERROR:', err)
```

The false command always exits with a non-zero status code, which run() interprets as an error.

The standard input and output channels for the process started by run() are bound to the parent's input and output. That means the calling program cannot capture the output of the command. Pass PIPE for the stdout and stderr arguments to capture the output for later processing.

```
import subprocess

completed = subprocess.run(
    ['ls', '-lahF'],
    stdout=subprocess.PIPE,
)
print('returncode:', completed.returncode)
print('Have {} bytes in stdout:\n{}'.format(
    len(completed.stdout),
    completed.stdout.decode('utf-8'))
)
```

The function run() is a wrapper around the Popen class. Using Popen directly gives more control over how the command is run, and how its input and output streams are processed. For example, by passing different arguments for stdin, stdout, and stderr it is possible to mimic the variations of os.popen().

To run a process and read all of its output, set the stdout value to PIPE and call communicate().

```
import subprocess

print('read:')
proc = subprocess.Popen(
   ['echo', '"to stdout"'],
   stdout=subprocess.PIPE,
)
stdout_value = proc.communicate()[0].decode('utf-8')
print('stdout:', repr(stdout_value))
```

This is similar to the way the previous example works, except that the reading is managed internally by the Popen instance.

One-way communication

To set up a pipe to allow the calling program to write data to it, set stdin to PIPE.

```
import subprocess

print('write:')
proc = subprocess.Popen(
   ['cat', '-'],
   stdin=subprocess.PIPE,
)
proc.communicate('stdin: to stdin\n'.encode('utf-8'))
```

To send data to the standard input channel of the process one time, pass the data to communicate().

Two-way communication

To set up the Popen instance for reading and writing at the same time, use a combination of the previous techniques.

```
import subprocess

print('popen2:')

proc = subprocess.Popen(
```

```
    ['cat', '-'],
    stdin=subprocess.PIPE,
    stdout=subprocess.PIPE,
)
msg = 'through stdin to stdout'.encode('utf-8')
stdout_value = proc.communicate(msg)[0].decode('utf-8')
print('pass through:', repr(stdout_value))
```

Using pipes

Multiple commands can be connected into a pipeline, similar to the way the Unix shell works, by creating separate Popen instances and chaining their inputs and outputs together. The stdout attribute of one Popen instance is used as the stdin argument for the next in the pipeline, instead of the constant PIPE. The output is read from the stdout handle for the final command in the pipeline.

```
import subprocess

cat = subprocess.Popen(
    ['cat', 'mydemo.txt'],
    stdout=subprocess.PIPE,
)

grep = subprocess.Popen(
    ['grep', 'this is a text'],
    stdin=cat.stdout,
    stdout=subprocess.PIPE,
)

cut = subprocess.Popen(
    ['cut', '-f', '3', '-d:'],
    stdin=grep.stdout,
    stdout=subprocess.PIPE,
)

end_of_pipe = cut.stdout

print('Included files:')
for line in end_of_pipe:
    print(line.decode('utf-8').strip())
```

The example reproduces the command line:

```
$ cat index.rst | grep ".. literalinclude" | cut -f 3 -d:
```

Signalling Between Processes

The process management examples for the os module include a demonstration of signalling between processes using os.fork() and os.kill(). Since each Popen instance provides a pid

attribute with the process id of the child process, it is possible to do something similar with subprocess. The next example combines two scripts. This child process sets up a signal handler for the USR signal.

```
# signal_child.py
import os
import signal
import time
import sys

pid = os.getpid()
received = False

def signal_usr1(signum, frame):
    "Callback invoked when a signal is received"
    global received
    received = True
    print('CHILD {:>6}: Received USR1'.format(pid))
    sys.stdout.flush()


print('CHILD {:>6}: Setting up signal handler'.format(pid))
sys.stdout.flush()
signal.signal(signal.SIGUSR1, signal_usr1)
print('CHILD {:>6}: Pausing to wait for signal'.format(pid))
sys.stdout.flush()
time.sleep(3)

if not received:
    print('CHILD {:>6}: Never received signal'.format(pid))
```

This script runs as the parent process. It starts signal_child.py, then sends the USR1 signal.

```
# signal_parent.py
import os
import signal
import subprocess
import time
import sys

proc = subprocess.Popen(['python3', 'signal_child.py'])
print('PARENT      : Pausing before sending signal...')
sys.stdout.flush()
time.sleep(1)
print('PARENT      : Signaling child')
sys.stdout.flush()
os.kill(proc.pid, signal.SIGUSR1)
```

Signal

This is a subset of the examples presented here: https://pymotw.com/3/signal/index.html. For more detailed information, please check the PyMOTW-3 website.

Receiving Signals

As with other forms of event-based programming, signals are received by establishing a callback function, called a signal handler, that is invoked when the signal occurs. The arguments to the signal handler are the signal number and the stack frame from the point in the program that was interrupted by the signal.

```
import signal
import os
import time

def receive_signal(signum, stack):
    print('Received:', signum)

# Register signal handlers
signal.signal(signal.SIGUSR1, receive_signal)
signal.signal(signal.SIGUSR2, receive_signal)

# Print the process ID so it can be used with 'kill'
# to send this program signals.
print('My PID is:', os.getpid())

while True:
    print('Waiting...')
    time.sleep(3)
```

This example script loops indefinitely, pausing for a few seconds each time. When a signal comes in, the sleep() call is interrupted and the signal handler receive_signal prints the signal number. After the signal handler returns, the loop continues.

Send signals to the running program using os.kill() or the Unix command line program kill.

The previous output was produced by running signal_signal.py in one window, then in another window running:

```
$ kill -USR1 $pid
$ kill -USR2 $pid
$ kill -INT $pid
```

Signals and Threads

Signals and threads do not generally mix well because only the main thread of a process will receive signals. The following example sets up a signal handler, waits for the signal in one thread, and sends the signal from another.

```
import signal
import threading
import os
import time

def signal_handler(num, stack):
    print('Received signal {} in {}'.format(
        num, threading.currentThread().name))


signal.signal(signal.SIGUSR1, signal_handler)


def wait_for_signal():
    print('Waiting for signal in',
          threading.currentThread().name)
    signal.pause()
    print('Done waiting')


# Start a thread that will not receive the signal
receiver = threading.Thread(
    target=wait_for_signal,
    name='receiver',
)
receiver.start()
time.sleep(0.1)


def send_signal():
    print('Sending signal in', threading.currentThread().name)
    os.kill(os.getpid(), signal.SIGUSR1)


sender = threading.Thread(target=send_signal, name='sender')
sender.start()
sender.join()

# Wait for the thread to see the signal (not going to happen!)
print('Waiting for', receiver.name)
signal.alarm(2)
receiver.join()
```

The signal handlers were all registered in the main thread because this is a requirement of the signal module implementation for Python, regardless of underlying platform support for mixing threads and signals. Although the receiver thread calls signal.pause(), it does not

receive the signal. The signal.alarm(2) call near the end of the example prevents an infinite block, since the receiver thread will never exit.

Although alarms can be set in any thread, they are always received by the main thread.

```
import signal
import time
import threading

def signal_handler(num, stack):
    print(time.ctime(), 'Alarm in',
          threading.currentThread().name)

signal.signal(signal.SIGALRM, signal_handler)

def use_alarm():
    t_name = threading.currentThread().name
    print(time.ctime(), 'Setting alarm in', t_name)
    signal.alarm(1)
    print(time.ctime(), 'Sleeping in', t_name)
    time.sleep(3)
    print(time.ctime(), 'Done with sleep in', t_name)

# Start a thread that will not receive the signal
alarm_thread = threading.Thread(
    target=use_alarm,
    name='alarm_thread',
)
alarm_thread.start()
time.sleep(0.1)

# Wait for the thread to see the signal (not going to happen!)
print(time.ctime(), 'Waiting for', alarm_thread.name)
alarm_thread.join()

print(time.ctime(), 'Exiting normally')
```

The alarm does not abort the sleep() call in use_alarm().

Threading

The threading interface is used to manage the threads of execution. It allows a program to run multiple operations concurrently in the same process space.

The simplest way to use a Thread is to instantiate it with a target function and call start() to let it begin working.

```
import threading
```

```
def worker():
    """thread worker function"""
    print('Worker')


threads = []
for i in range(5):
    t = threading.Thread(target=worker)
    threads.append(t)
    t.start()
```

It is useful to be able to spawn a thread and pass it arguments to tell it what work to do. Any type of object can be passed as argument to the thread. This example passes a number, which the thread then prints.

```
import threading

def worker(num):
    """thread worker function"""
    print('Worker: %s' % num)

threads = []
for i in range(5):
    t = threading.Thread(target=worker, args=(i,))
    threads.append(t)
    t.start()
```

Threads can be named with:
threading.Thread(name='my_worker', target=def_worker)

In order to identify the current thread one can use:
threading.current_thread().getName()

Can you modify previous scripts in such a way that you give a unique name to each used thread and then print it?

Daemon vs. Non-Daemon Threads

Up to this point, the example programs have implicitly waited to exit until all threads have completed their work. Sometimes programs spawn a thread as a daemon that runs without blocking the main program from exiting. Using daemon threads is useful for services where there may not be an easy way to interrupt the thread, or where letting the thread die in the middle of its work does not lose or corrupt data (for example, a thread that generates "heart beats" for a service monitoring tool). To mark a thread as a daemon, pass

daemon=True when constructing it or call its set_daemon() method with True. The default is for threads to not be daemons.

```
import threading
import time

def daemon():
    print('Starting daemon')
    time.sleep(0.2)
    print('Exiting daemon')


def non_daemon():
    print ('Starting non-daemon')
    print ('Exiting non-daemon')

d = threading.Thread(name='daemon', target=daemon, daemon=True)
t = threading.Thread(name='non-daemon', target=non_daemon)

d.start()
t.start()
```

To wait until a daemon thread has completed its work, use the join() method. So, at the end of the previous example add

```
d.join()
t.join()
```

By default, join() blocks indefinitely. It is also possible to pass a float value representing the number of seconds to wait for the thread to become inactive. If the thread does not complete within the timeout period, join() returns anyway. So the two lines from above become in this case:

```
d.join(0.1)
print('d.isAlive()', d.isAlive())
t.join()
```

Enumerating All Threads

It is not necessary to retain an explicit handle to all of the daemon threads in order to ensure they have completed before exiting the main process. enumerate() returns a list of active Thread instances. The list includes the current thread, and since joining the current thread introduces a deadlock situation, it must be skipped.

```
import random
import threading
import time
```

```
def worker():
    """thread worker function"""
    pause = random.randint(1, 5) / 10
    print('sleeping %0.2f' % pause)
    time.sleep(pause)
    print('ending')

for i in range(3):
    t = threading.Thread(target=worker, daemon=True)
    t.start()

main_thread = threading.main_thread()
for t in threading.enumerate():
    if t is main_thread:
        continue
    print('joining %s' % t.getName())
    t.join()
```

Signalling Between Threads

Although the point of using multiple threads is to run separate operations concurrently, there are times when it is important to be able to synchronize the operations in two or more threads. Event objects are a simple way to communicate between threads safely. An Event manages an internal flag that callers can control with the set() and clear() methods. Other threads can use wait() to pause until the flag is set, effectively blocking progress until allowed to continue.

```
import logging
import threading
import time

def wait_for_event(e):
    """Wait for the event to be set before doing anything"""
    logging.debug('wait_for_event starting')
    event_is_set = e.wait()
    logging.debug('event set: %s', event_is_set)

def wait_for_event_timeout(e, t):
    """Wait t seconds and then timeout"""
    while not e.is_set():
        logging.debug('wait_for_event_timeout starting')
        event_is_set = e.wait(t)
        logging.debug('event set: %s', event_is_set)
        if event_is_set:
            logging.debug('processing event')
        else:
            logging.debug('doing other work')
```

```
logging.basicConfig(
    level=logging.DEBUG,
    format='(%(threadName)-10s) %(message)s',
)

e = threading.Event()
t1 = threading.Thread(
    name='block',
    target=wait_for_event,
    args=(e,),
)
t1.start()

t2 = threading.Thread(
    name='nonblock',
    target=wait_for_event_timeout,
    args=(e, 2),
)
t2.start()

logging.debug('Waiting before calling Event.set()')
time.sleep(0.3)
e.set()
logging.debug('Event is set')
```

The wait() method takes an argument representing the number of seconds to wait for the event before timing out. It returns a Boolean indicating whether or not the event is set, so the caller knows why wait() returned. The is_set() method can be used separately on the event without fear of blocking.

In this example, wait_for_event_timeout() checks the event status without blocking indefinitely. The wait_for_event() blocks on the call to wait(), which does not return until the event status changes.

Controlling Access to Resources

In addition to synchronizing the operations of threads, it is also important to be able to control access to shared resources to prevent corruption or missed data. Python's built-in data structures (lists, dictionaries, etc.) are thread-safe as a side-effect of having atomic byte-codes for manipulating them (the global interpreter lock used to protect Python's internal data structures is not released in the middle of an update). Other data structures implemented in Python, or simpler types like integers and floats, do not have that protection. To guard against simultaneous access to an object, use a Lock object.

```
import logging
import random
import threading
```

```
import time

class Counter:

    def __init__(self, start=0):
        self.lock = threading.Lock()
        self.value = start

    def increment(self):
        logging.debug('Waiting for lock')
        self.lock.acquire()
        try:
            logging.debug('Acquired lock')
            self.value = self.value + 1
        finally:
            self.lock.release()

def worker(c):
    for i in range(2):
        pause = random.random()
        logging.debug('Sleeping %0.02f', pause)
        time.sleep(pause)
        c.increment()
    logging.debug('Done')

logging.basicConfig(
    level=logging.DEBUG,
    format='(%(threadName)-10s) %(message)s',
)

counter = Counter()
for i in range(2):
    t = threading.Thread(target=worker, args=(counter,))
    t.start()

logging.debug('Waiting for worker threads')
main_thread = threading.main_thread()
for t in threading.enumerate():
    if t is not main_thread:
        t.join()
logging.debug('Counter: %d', counter.value)
```

In this example, the worker() function increments a Counter instance, which manages a Lock to prevent two threads from changing its internal state at the same time. If the Lock was not used, there is a possibility of missing a change to the value attribute.

To find out whether another thread has acquired the lock without holding up the current thread, pass False for the blocking argument to acquire(). In the next example, worker() tries to acquire the lock three separate times and counts how many attempts it has to make to do so. In the meantime, lock_holder() cycles between holding and releasing the lock, with short pauses in each state used to simulate a load.

```
import logging
import threading
import time

def lock_holder(lock):
    logging.debug('Starting')
    while True:
        lock.acquire()
        try:
            logging.debug('Holding')
            time.sleep(0.5)
        finally:
            logging.debug('Not holding')
            lock.release()
        time.sleep(0.5)

def worker(lock):
    logging.debug('Starting')
    num_tries = 0
    num_acquires = 0
    while num_acquires < 3:
        time.sleep(0.5)
        logging.debug('Trying to acquire')
        have_it = lock.acquire(0)
        try:
            num_tries += 1
            if have_it:
                logging.debug('Iteration %d: Acquired',
                              num_tries)
                num_acquires += 1
            else:
                logging.debug('Iteration %d: Not acquired',
                              num_tries)
        finally:
            if have_it:
                lock.release()
    logging.debug('Done after %d iterations', num_tries)

logging.basicConfig(
    level=logging.DEBUG,
    format='(%(threadName)-10s) %(message)s',
```

```
)

lock = threading.Lock()

holder = threading.Thread(
    target=lock_holder,
    args=(lock,),
    name='LockHolder',
    daemon=True,
)
holder.start()

worker = threading.Thread(
    target=worker,
    args=(lock,),
    name='Worker',
)
worker.start()
```

It takes worker() more than three iterations to acquire the lock three separate times.

Normal Lock objects cannot be acquired more than once, even by the same thread. This can introduce undesirable side-effects if a lock is accessed by more than one function in the same call chain.

```
import threading

lock = threading.Lock()

print('First try :', lock.acquire())
print('Second try:', lock.acquire(0))
```

In this case, the second call to acquire() is given a zero timeout to prevent it from blocking because the lock has been obtained by the first call.

In a situation where separate code from the same thread needs to “re-acquire” the lock, use an RLock instead.

```
import threading

lock = threading.RLock()

print('First try :', lock.acquire())
print('Second try:', lock.acquire(0))
```

The only change to the code from the previous example was substituting RLock for Lock.

Locks implement the context manager API and are compatible with the with statement. Using with removes the need to explicitly acquire and release the lock.

```
import threading
import logging

def worker_with(lock):
    with lock:
        logging.debug('Lock acquired via with')

logging.basicConfig(
    level=logging.DEBUG,
    format='(%(threadName)-10s) %(message)s',
)

lock = threading.Lock()
w = threading.Thread(target=worker_with, args=(lock,))
nw = threading.Thread(target=worker_no_with, args=(lock,))

w.start()
nw.start()
```

Can you write an equivalent function for worker_with() that manages the lock explicitly? What happens if you just get a thread's name without explicitly setting it?

Synchronizing Threads

In addition to using Events, another way of synchronizing threads is through using a Condition object. Because the Condition uses a Lock, it can be tied to a shared resource, allowing multiple threads to wait for the resource to be updated. In this example, the consumer() threads wait for the Condition to be set before continuing. The producer() thread is responsible for setting the condition and notifying the other threads that they can continue.

```
import logging
import threading
import time

def consumer(cond):
    """wait for the condition and use the resource"""
    logging.debug('Starting consumer thread')
    with cond:
        cond.wait()
```

```
        logging.debug('Resource is available to consumer')

def producer(cond):
    """set up the resource to be used by the consumer"""
    logging.debug('Starting producer thread')
    with cond:
        logging.debug('Making resource available')
        cond.notifyAll()


logging.basicConfig(
    level=logging.DEBUG,
    format='%(asctime)s (%(threadName)-2s) %(message)s',
)

condition = threading.Condition()
c1 = threading.Thread(name='c1', target=consumer,
                      args=(condition,))
c2 = threading.Thread(name='c2', target=consumer,
                      args=(condition,))
p = threading.Thread(name='p', target=producer,
                     args=(condition,))

c1.start()
time.sleep(0.2)
c2.start()
time.sleep(0.2)
p.start()
```

The threads use with to acquire the lock associated with the Condition. Using the acquire() and release() methods explicitly also works.

Barriers are another thread synchronization mechanism. A Barrier establishes a control point and all participating threads block until all of the participating “parties” have reached that point. It lets threads start up separately and then pause until they are all ready to proceed.

```
import threading
import time

def worker(barrier):
    print(threading.current_thread().name,
          'waiting for barrier with {} others'.format(
              barrier.n_waiting))
    worker_id = barrier.wait()
    print(threading.current_thread().name, 'after barrier',
          worker_id)
```

```
NUM_THREADS = 3

barrier = threading.Barrier(NUM_THREADS)

threads = [
   threading.Thread(
      name='worker-%s' % i,
      target=worker,
      args=(barrier,),
   )
   for i in range(NUM_THREADS)
]

for t in threads:
   print(t.name, 'starting')
   t.start()
   time.sleep(0.1)

for t in threads:
   t.join()
```

In this example, the Barrier is configured to block until three threads are waiting. When the condition is met, all of the threads are released past the control point at the same time. The return value from wait() indicates the number of the party being released, and can be used to limit some threads from taking an action like cleaning up a shared resource.

The abort() method of Barrier causes all of the waiting threads to receive a BrokenBarrierError. This allows threads to clean up if processing is stopped while they are blocked on wait().

```
import threading
import time

def worker(barrier):
   print(threading.current_thread().name,
         'waiting for barrier with {} others'.format(
            barrier.n_waiting))
   try:
      worker_id = barrier.wait()
   except threading.BrokenBarrierError:
      print(threading.current_thread().name, 'aborting')
   else:
      print(threading.current_thread().name, 'after barrier',
            worker_id)
```

```
NUM_THREADS = 3

barrier = threading.Barrier(NUM_THREADS + 1)

threads = [
   threading.Thread(
      name='worker-%s' % i,
      target=worker,
      args=(barrier,),
   )
   for i in range(NUM_THREADS)
]

for t in threads:
   print(t.name, 'starting')
   t.start()
   time.sleep(0.1)

barrier.abort()

for t in threads:
   t.join()
```

This example configures the Barrier to expect one more participating thread than is actually started so that processing in all of the threads is blocked. The abort() call raises an exception in each blocked thread.

Limiting Concurrent Access to Resources

Sometimes it is useful to allow more than one worker access to a resource at a time, while still limiting the overall number. For example, a connection pool might support a fixed number of simultaneous connections, or a network application might support a fixed number of concurrent downloads. A Semaphore is one way to manage those connections.

```
import logging
import random
import threading
import time

class ActivePool:

   def __init__(self):
      super(ActivePool, self).__init__()
      self.active = []
      self.lock = threading.Lock()

   def makeActive(self, name):
```

```
        with self.lock:
            self.active.append(name)
            logging.debug('Running: %s', self.active)

    def makeInactive(self, name):
        with self.lock:
            self.active.remove(name)
            logging.debug('Running: %s', self.active)

def worker(s, pool):
    logging.debug('Waiting to join the pool')
    with s:
        name = threading.current_thread().getName()
        pool.makeActive(name)
        time.sleep(0.1)
        pool.makeInactive(name)

logging.basicConfig(
    level=logging.DEBUG,
    format='%(asctime)s (%(threadName)-2s) %(message)s',
)

pool = ActivePool()
s = threading.Semaphore(2)
for i in range(4):
    t = threading.Thread(
        target=worker,
        name=str(i),
        args=(s, pool),
    )
    t.start()
```

In this example, the ActivePool class simply serves as a convenient way to track which threads are able to run at a given moment. A real resource pool would allocate a connection or some other value to the newly active thread, and reclaim the value when the thread is done. Here, it is just used to hold the names of the active threads to show that at most two are running concurrently.

Thread-specific Data

While some resources need to be locked so multiple threads can use them, others need to be protected so that they are hidden from threads that do not own them. The local() class creates an object capable of hiding values from view in separate threads.

```
import random
import threading
import logging
```

```
def show_value(data):
    try:
        val = data.value
    except AttributeError:
        logging.debug('No value yet')
    else:
        logging.debug('value=%s', val)

def worker(data):
    show_value(data)
    data.value = random.randint(1, 100)
    show_value(data)

logging.basicConfig(
    level=logging.DEBUG,
    format='(%(threadName)-10s) %(message)s',
)

local_data = threading.local()
show_value(local_data)
local_data.value = 1000
show_value(local_data)

for i in range(2):
    t = threading.Thread(target=worker, args=(local_data,))
    t.start()
```

The attribute local_data.value is not present for any thread until it is set in that thread.

To initialize the settings so all threads start with the same value, use a subclass and set the attributes in __init__().

Hint: __init__() is invoked on the same object (note the id() value), once in each thread to set the default values.

Queues

Threading can be complicated when threads need to share data or resources. The threading module does provide many synchronization primitives, including semaphores, condition variables, events, and locks. While these options exist, it is considered a best practice to instead concentrate on using queues. Queues are much easier to deal with, and make threaded programming considerably safer, as they effectively funnel all access to a resource to a single thread, and allow a cleaner and more readable design pattern.

Let's take a look at a very simple example of a program that will serially, or one after the other, grab a URL of a website, and print out the first 1024 bytes of the page. This is a classic example of something that could be done quicker using threads. First, let's use the urllib2 module to grab these pages one at a time, and time the code:

```
from urllib.request import urlopen
import cProfile

def do_profile(func):
    def profiled_func(*args, **kwargs):
        profile = cProfile.Profile()
        try:
            profile.enable()
            result = func(*args, **kwargs)
            profile.disable()
            return result
        finally:
            profile.print_stats()
    return profiled_func

@do_profile
def fetch_pages(hosts):
    for host in hosts:
        url = urlopen(host)
        print("Fetched : {0}".format(host))

hosts = ["https://lapp.in2p3.fr", "http://www.surfsara.nl", "http://www.intel.fr"]
fetch_pages(hosts)
```

Now, let's rewrite this code by using queues.

```
import queue as Queue
import threading
from urllib.request import urlopen
import cProfile

def do_profile(func):
    def profiled_func(*args, **kwargs):
        profile = cProfile.Profile()
        try:
            profile.enable()
            result = func(*args, **kwargs)
            profile.disable()
            return result
        finally:
            profile.print_stats()
    return profiled_func
```

```
queue = Queue.Queue()

class ThreadUrl(threading.Thread):
    """Threaded Url Grab"""
    def __init__(self, queue):
        threading.Thread.__init__(self)
        self.queue = queue

    def run(self):
        while True:
          #grabs host from queue
          host = self.queue.get()

          #grabs urls of hosts and prints first 1024 bytes of page
          url = urlopen(host)
          print("Fetched: {0}".format(host))

          #signals to queue job is done
          self.queue.task_done()

@do_profile
def fetch_pages(hosts):
    #spawn a pool of threads, and pass them queue instance
    for i in range(len(hosts)):
        t = ThreadUrl(queue)
        t.setDaemon(True)
        t.start()

        #populate queue with data
        for host in hosts:
            queue.put(host)

    #wait on the queue until everything has been processed
    queue.join()

hosts = ["https://lapp.in2p3.fr", "http://www.surfsara.nl", "http://yahoo.com"]
fetch_pages(hosts)
```

By setting daemonic threads to true, it allows the main thread, or program, to exit if only daemonic threads are alive. This creates a simple way to control the flow of the program, because you can then join on the queue, or wait until the queue is empty, before exiting.

join()
"Blocks until all items in the queue have been gotten and processed. The count of unfinished tasks goes up whenever an item is added to the queue. The count goes down

whenever a consumer thread calls task_done() to indicate that the item was retrieved and all work on it is complete. When the count of unfinished tasks drops to zero, join() unblocks.

Multiprocessing

In addition to the pymotw is the official Python3 documentation page:
https://docs.python.org/3/library/multiprocessing.html
Python's Queues and threads are limited to running on only one core due to the Global Interpreter Lock (GIL) that is a part of Python. The multiprocessing module can run on multiple cores and gets around the GIL issue.

The multiprocessing package also includes some APIs that are not in the threading module at all. For example, there is a neat Pool class that you can use to parallelize executing a function across multiple inputs.

The Process class

It is very similar to the threading module's Thread class. Let's try creating a series of processes that call the same function and see how that works:

```
import os

from multiprocessing import Process

def doubler(number):
    """
    A doubling function that can be used by a process
    """
    result = number * 2
    proc = os.getpid()
    print('{0} doubled to {1} by process id: {2}'.format(
        number, result, proc))

if __name__ == '__main__':
    numbers = [5, 10, 15, 20, 25]
    procs = []

    for index, number in enumerate(numbers):
        proc = Process(target=doubler, args=(number,))
        procs.append(proc)
        proc.start()

    for proc in procs:
        proc.join()
```

For this example, we import Process and create a doubler function. Inside the function, we double the number that was passed in. We also use Python's os module to get the current process's ID (or pid). This will tell us which process is calling the function. Then in the block of code at the bottom, we create a series of Processes and start them. The very last loop

just calls the join() method on each process, which tells Python to wait for the process to terminate. If you need to stop a process, you can call its terminate() method.

Let's take a look now at the current_process method. The current_process is basically the same thing as the threading module's current_thread:

```
import os

from multiprocessing import Process, current_process


def doubler(number):
    """
    A doubling function that can be used by a process
    """
    result = number * 2
    proc_name = current_process().name
    print('{0} doubled to {1} by: {2}'.format(
        number, result, proc_name))


if __name__ == '__main__':
    numbers = [5, 10, 15, 20, 25]
    procs = []
    proc = Process(target=doubler, args=(5,))

    for index, number in enumerate(numbers):
        proc = Process(target=doubler, args=(number,))
        procs.append(proc)
        proc.start()

    proc = Process(target=doubler, name='Test', args=(2,))
    proc.start()
    procs.append(proc)

    for proc in procs:
        proc.join()
```

We use it to grab the name of the thread that is calling our function. You will note that for the first five processes, we don't set a name. Then for the sixth, we set the process name to "Test".

Locks

The multiprocessing module supports locks in much the same way as the threading module does. All you need to do is import Lock, acquire it, do something and release it. Let's take a look:

```
from multiprocessing import Process, Lock


def printer(item, lock):
    """
    Prints out the item that was passed in
    """
    lock.acquire()
    try:
        print(item)
    finally:
        lock.release()

if __name__ == '__main__':
    lock = Lock()
    items = ['tango', 'foxtrot', 10]
    for item in items:
        p = Process(target=printer, args=(item, lock))
        p.start()
```

Logging

Logging processes is a little different than logging threads. The reason for this is that Python's logging packages doesn't use process shared locks, so it's possible for you to end up with messages from different processes getting mixed up. Let's try adding basic logging to the previous example. Here's the code:

```
import logging
import multiprocessing

from multiprocessing import Process, Lock

def printer(item, lock):
    """
    Prints out the item that was passed in
    """
    lock.acquire()
    try:
        print(item)
    finally:
        lock.release()

if __name__ == '__main__':
    lock = Lock()
    items = ['tango', 'foxtrot', 10]
    multiprocessing.log_to_stderr()
    logger = multiprocessing.get_logger()
    logger.setLevel(logging.INFO)
```

```
    for item in items:
        p = Process(target=printer, args=(item, lock))
        p.start()
```

The simplest way to log is to send it all to stderr. We can do this by calling the log_to_stderr() function. Then we call the get_logger function to get access to a logger and set its logging level to INFO. The rest of the code is the same. I will note that I'm not calling the join() method here. Instead, the parent thread (i.e. your script) will call join() implicitly when it exits.

The Pool Class

The Pool class is used to represent a pool of worker processes. It has methods which can allow you to offload tasks to the worker processes. Let's look at a really simple example:

```
from multiprocessing import Pool

def doubler(number):
    return number * 2

if __name__ == '__main__':
    numbers = [5, 10, 20]
    pool = Pool(processes=3)
    print(pool.map(doubler, numbers))
```

You can also get the result of your process in a pool by using the apply_async method:

```
from multiprocessing import Pool

def doubler(number):
    return number * 2

if __name__ == '__main__':
    pool = Pool(processes=3)
    result = pool.apply_async(doubler, [25])
    print(result.get(timeout=1))
```

Why is the input value for doubler in a list? Could it be something else other than a list?

Process communication

When it comes to communicating between processes, the multiprocessing modules has two primary methods: Queues and Pipes. The Queue implementation is actually both thread and process safe. Let's take a look at a fairly simple example that's based on the Queue code:

```
from multiprocessing import Process, Queue

sentinel = -1

def creator(data, q):
    """
    Creates data to be consumed and waits for the consumer
    to finish processing
    """
    print('Creating data and putting it on the queue')
    for item in data:

        q.put(item)


def my_consumer(q):
    """
    Consumes some data and works on it

    In this case, all it does is double the input
    """
    while True:
        data = q.get()
        print('data found to be processed: {}'.format(data))
        processed = data * 2
        print(processed)

        if data is sentinel:
            break


if __name__ == '__main__':
    q = Queue()
    data = [5, 10, 13, -1]
    process_one = Process(target=creator, args=(data, q))
    process_two = Process(target=my_consumer, args=(q,))
    process_one.start()
    process_two.start()

    q.close()
    q.join_thread()

    process_one.join()
    process_two.join()
```

Here we just need to import Queue and Process. Then we two functions, one to create data and add it to the queue and the second to consume the data and process it. Adding data to

the Queue is done by using the Queue's put() method whereas getting data from the Queue is done via the get method. The last chunk of code just creates the Queue object and a couple of Processes and then runs them. You will note that we call join() on our process objects rather than the Queue itself.

There are a lot of resources / tutorials dealing with the multiprocessing module, as historically, this has been the de facto way to parallelize Python code until the introduction of Futures and asyncio.

concurrent.futures

Manages pools of concurrent tasks. As always be sure to check the corresponding pymotw (main source for the following) and corresponding python documentation page for more detailed information

The concurrent.futures modules provides interfaces for running tasks using pools of thread or process workers. The APIs are the same, so applications can switch between threads and processes with minimal changes.

The module provides two types of classes for interacting with the pools. Executors are used for managing pools of workers, and futures are used for managing results computed by the workers. To use a pool of workers, an application creates an instance of the appropriate executor class and then submits tasks for it to run. When each task is started, a Future instance is returned. When the result of the task is needed, an application can use the Future to block until the result is available. Various APIs are provided to make it convenient to wait for tasks to complete, so that the Future objects do not need to be managed directly.

Using map() with a Basic Thread Pool

The ThreadPoolExecutor manages a set of worker threads, passing tasks to them as they become available for more work. This example uses map() to concurrently produce a set of results from an input iterable. The task uses time.sleep() to pause a different amount of time to demonstrate that, regardless of the order of execution of concurrent tasks, map() always returns the values in order based on the inputs.

```
from concurrent import futures
import threading
import time


def task(n):
    print('{}: sleeping {}'.format(
        threading.current_thread().name,
        n)
    )
    time.sleep(n / 10)
    print('{}: done with {}'.format(
        threading.current_thread().name,
```

```
        n)
    )
    return n / 10


ex = futures.ThreadPoolExecutor(max_workers=2)
print('main: starting')
results = ex.map(task, range(5, 0, -1))
print('main: unprocessed results {}'.format(results))
print('main: waiting for real results')
real_results = list(results)
print('main: results: {}'.format(real_results))
```

The return value from map() is actually a special type of iterator that knows to wait for each response as the main program iterates over it.

Scheduling Individual Tasks

In addition to using map(), it is possible to schedule an individual task with an executor using submit(), and use the Future instance returned to wait for that task's results.

```
from concurrent import futures
import threading
import time


def task(n):
    print('{}: sleeping {}'.format(
        threading.current_thread().name,
        n)
    )
    time.sleep(n / 10)
    print('{}: done with {}'.format(
        threading.current_thread().name,
        n)
    )
    return n / 10


ex = futures.ThreadPoolExecutor(max_workers=2)
print('main: starting')
f = ex.submit(task, 5)
print('main: future: {}'.format(f))
print('main: waiting for results')
result = f.result()
print('main: result: {}'.format(result))
print('main: future after result: {}'.format(f))
```

The status of the future changes after the tasks is completed and the result is made available.

Waiting for Tasks in Any Order

Invoking the result() method of a Future blocks until the task completes (either by returning a value or raising an exception), or is canceled. The results of multiple tasks can be accessed in the order the tasks were scheduled using map(). If it does not matter what order the results should be processed, use as_completed() to process them as each task finishes.

```
from concurrent import futures
import random
import time


def task(n):
    time.sleep(random.random())
    return (n, n / 10)


ex = futures.ThreadPoolExecutor(max_workers=5)
print('main: starting')

wait_for = [
    ex.submit(task, i)
    for i in range(5, 0, -1)
]

for f in futures.as_completed(wait_for):
    print('main: result: {}'.format(f.result()))
```

Because the pool has as many workers as tasks, all of the tasks can be started. They finish in a random order so the values generated by as_completed() are different each time the example runs.

To take some action when a task completed, without explicitly waiting for the result, use add_done_callback() to specify a new function to call when the Future is done. The callback should be a callable taking a single argument, the Future instance.

```
from concurrent import futures
import time


def task(n):
    print('{}: sleeping'.format(n))
    time.sleep(0.5)
    print('{}: done'.format(n))
```

```
    return n / 10


def done(fn):
    if fn.cancelled():
        print('{}: canceled'.format(fn.arg))
    elif fn.done():
        error = fn.exception()
        if error:
            print('{}: error returned: {}'.format(
                fn.arg, error))
        else:
            result = fn.result()
            print('{}: value returned: {}'.format(
                fn.arg, result))


if __name__ == '__main__':
    ex = futures.ThreadPoolExecutor(max_workers=2)
    print('main: starting')
    f = ex.submit(task, 5)
    f.arg = 5
    f.add_done_callback(done)
    result = f.result()
```

The callback is invoked regardless of the reason the Future is considered “done,” so it is necessary to check the status of the object passed in to the callback before using it in any way.

A Future can be canceled, if it has been submitted but not started, by calling its cancel() method.

```
from concurrent import futures
import time


def task(n):
    print('{}: sleeping'.format(n))
    time.sleep(0.5)
    print('{}: done'.format(n))
    return n / 10


def done(fn):
    if fn.cancelled():
        print('{}: canceled'.format(fn.arg))
    elif fn.done():
```

```
        print('{}: not canceled'.format(fn.arg))


if __name__ == '__main__':
    ex = futures.ThreadPoolExecutor(max_workers=2)
    print('main: starting')
    tasks = []

    for i in range(10, 0, -1):
        print('main: submitting {}'.format(i))
        f = ex.submit(task, i)
        f.arg = i
        f.add_done_callback(done)
        tasks.append((i, f))

    for i, t in reversed(tasks):
        if not t.cancel():
            print('main: did not cancel {}'.format(i))

    ex.shutdown()
```

cancel() returns a Boolean indicating whether or not the task was able to be canceled.

If a task raises an unhandled exception, it is saved to the Future for the task and made available through the result() or exception() methods.

```
from concurrent import futures

def task(n):
    print('{}: starting'.format(n))
    raise ValueError('the value {} is no good'.format(n))


ex = futures.ThreadPoolExecutor(max_workers=2)
print('main: starting')
f = ex.submit(task, 5)

error = f.exception()
print('main: error: {}'.format(error))

try:
    result = f.result()
except ValueError as e:
    print('main: saw error "{}" when accessing result'.format(e))
```

If result() is called after an unhandled exception is raised within a task function, the same exception is re-raised in the current context.

Executors work as context managers, running tasks concurrently and waiting for them all to complete. When the context manager exits, the shutdown() method of the executor is called.

```
from concurrent import futures


def task(n):
    print(n)


with futures.ThreadPoolExecutor(max_workers=2) as ex:
    print('main: starting')
    ex.submit(task, 1)
    ex.submit(task, 2)
    ex.submit(task, 3)
    ex.submit(task, 4)

print('main: done')
```

This mode of using the executor is useful when the thread or process resources should be cleaned up when execution leaves the current scope.

The ProcessPoolExecutor works in the same way as ThreadPoolExecutor, but uses processes instead of threads. This allows CPU-intensive operations to use a separate CPU and not be blocked by the CPython interpreter's global interpreter lock.

```
from concurrent import futures
import os


def task(n):
    return (n, os.getpid())


ex = futures.ProcessPoolExecutor(max_workers=2)
results = ex.map(task, range(5, 0, -1))
for n, pid in results:
    print('ran task {} in process {}'.format(n, pid))
```

As with the thread pool, individual worker processes are reused for multiple tasks.

If something happens to one of the worker processes to cause it to exit unexpectedly, the ProcessPoolExecutor is considered "broken" and will no longer schedule tasks.

```
from concurrent import futures
```

```
import os
import signal

with futures.ProcessPoolExecutor(max_workers=2) as ex:
    print('getting the pid for one worker')
    f1 = ex.submit(os.getpid)
    pid1 = f1.result()

    print('killing process {}'.format(pid1))
    os.kill(pid1, signal.SIGHUP)

    print('submitting another task')
    f2 = ex.submit(os.getpid)
    try:
        pid2 = f2.result()
    except futures.process.BrokenProcessPool as e:
        print('could not start new tasks: {}'.format(e))
```

The BrokenProcessPool exception is actually thrown when the results are processed, rather than when the new task is submitted.

A very useful tutorial on futures can be found here: http://masnun.com/2016/03/29/python-a-quick-introduction-to-the-concurrent-futures-module.html

Asyncio

Asyncio is the new concurrency module introduced in Python 3.4. It's designed to use coroutines and futures to simplify asynchronous code and make it almost as readable as synchronous code simply because there are no callbacks.

The main building blocks in asyncio are:

- event loops - they register and distribute various tasks.
- coroutines - they are special functions that release the control to the event loops. They form tasks.
- futures - they are objects that represent the result of a task.

Code can be structure in subtasks. These are defined as coroutines and allow scheduling (including simultaneously). Coroutines contain yield points. In these context switches are possible if other tasks are waiting (otherwise not). A context switch in asyncio represents the event loop yielding the flow of control from one coroutine to the next.

A basic example would be:

```
import asyncio

async def foo():
    print('Running in foo')
```

```
    await asyncio.sleep(0)
    print('Explicit context switch to foo again')


async def bar():
    print('Explicit context to bar')
    await asyncio.sleep(0)
    print('Implicit context switch back to bar')


ioloop = asyncio.get_event_loop()
tasks = [ioloop.create_task(foo()), ioloop.create_task(bar())]
wait_tasks = asyncio.wait(tasks)
ioloop.run_until_complete(wait_tasks)
ioloop.close()
```

First we declare a couple of simple coroutines that pretend to do non-blocking work using the sleep function in asyncio.

Coroutines can only be called from other coroutines or be wrapped in a task and then scheduled, we use create_task.

Once we have the two tasks we combine them into one that waits for both of them to complete using wait.

And finally we schedule the wait task to run using our event loop's run_until_complete.

By using await on another coroutine we declare that the coroutine may give the control back to the event loop, in this case sleep. The coroutine will yield and the event loop will switch contexts to the next task scheduled for execution: bar. Similarly, the bar coroutine uses await sleep which allows the event loop to pass control back to foo at the point where it yielded, just as normal Python generators.

Let's now simulate two blocking tasks, gr1 and gr2, say they're two requests to external services. While those are executing a third task can be doing work asynchronously, like in the following example:

```
import time
import asyncio

start = time.time()


def tic():
    return 'at %1.1f seconds' % (time.time() - start)
```

```
async def gr1():
    # Busy waits for a second, but we don't want to stick around...
    print('gr1 started work: {}'.format(tic()))
    await asyncio.sleep(2)
    print('gr1 ended work: {}'.format(tic()))


async def gr2():
    # Busy waits for a second, but we don't want to stick around...
    print('gr2 started work: {}'.format(tic()))
    await asyncio.sleep(2)
    print('gr2 Ended work: {}'.format(tic()))


async def gr3():
    print("Let's do some stuff while the coroutines are blocked, {}".format(tic()))
    await asyncio.sleep(1)
    print("Done!")


ioloop = asyncio.get_event_loop()
tasks = [
    ioloop.create_task(gr1()),
    ioloop.create_task(gr2()),
    ioloop.create_task(gr3())
]
ioloop.run_until_complete(asyncio.wait(tasks))
ioloop.close()
```

Notice how the I/O loop manages and schedules the execution allowing your single threaded code to operate concurrently. While the two blocking tasks are blocked a third one can take control of the flow.

When using concurrency, we need to be aware that the tasks finish in different order than they were scheduled:

```
import random
from time import sleep
import asyncio

def task(pid):
    """Synchronous non-deterministic task.
    """
    sleep(random.randint(0, 2) * 0.001)
    print('Task %s done' % pid)

async def task_coro(pid):
```

```
    """Coroutine non-deterministic task
    """
    await asyncio.sleep(random.randint(0, 2) * 0.001)
    print('Task %s done' % pid)


def synchronous():
    for i in range(1, 10):
        task(i)

async def asynchronous():
    tasks = [asyncio.ensure_future(task_coro(i)) for i in range(1, 10)]
    await asyncio.wait(tasks)

print('Synchronous:')
synchronous()

ioloop = asyncio.get_event_loop()
print('Asynchronous:')
ioloop.run_until_complete(asynchronous())
ioloop.close()
```

Please notice how we had to create a coroutine version of our fairly simple task. It's important to understand that asyncio does not magically make things non-blocking. At the time of writing asyncio stands alone in the standard library, the rest of modules provide only blocking functionality. You can use the concurrent.futures module to wrap a blocking task in a thread or a process and return a Future asyncio can use:

```
import random
import concurrent
from time import sleep
import asyncio

executor = concurrent.futures.ThreadPoolExecutor(8)

def task(pid):
    """Some non-deterministic task
    """
    sleep(random.randint(0, 2) * 0.001)
    print('Task %s done' % pid)

@asyncio.coroutine
def task_coro(pid):
    """Some non-deterministic task
    """
    yield from asyncio.sleep(random.randint(0, 2) * 0.001)
    print('Task %s done' % pid)
```

```
def synchronous():
    for i in range(1, 10):
        task(i)

@asyncio.coroutine
def asynchronous_threads():
    # asyncio.ensure_future is named asyncio.async in Python < 3.4.4
    tasks = [
        asyncio.ensure_future(ioloop.run_in_executor(executor, task, i))
        for i in range(1, 10)
    ]
    yield from asyncio.wait(tasks)

@asyncio.coroutine
def asynchronous_coro():
    tasks = [asyncio.ensure_future(task_coro(i)) for i in range(1, 10)]
    yield from asyncio.wait(tasks)

print('Synchronous:')
synchronous()

ioloop = asyncio.get_event_loop()

print('Asynchronous threads:')
ioloop.run_until_complete(asynchronous_threads())

print('Asynchronous using coroutines:')
ioloop.run_until_complete(asynchronous_coro())

ioloop.close()
```

Coroutines can be scheduled to run or retrieve their results in different ways. Let's move onto Futures and a callback when the task is done.

```
import asyncio

@asyncio.coroutine
def my_coroutine(future, task_name, seconds_to_sleep=3):
    print('{0} sleeping for: {1} seconds'.format(task_name, seconds_to_sleep))
    yield from asyncio.sleep(seconds_to_sleep)
    future.set_result('{0} is finished'.format(task_name))


def got_result(future):
    print(future.result())
```

```
loop = asyncio.get_event_loop()
future1 = asyncio.Future()
future2 = asyncio.Future()

tasks = [
    my_coroutine(future1, 'task1', 3),
    my_coroutine(future2, 'task2', 1)]

future1.add_done_callback(got_result)
future2.add_done_callback(got_result)

loop.run_until_complete(asyncio.wait(tasks))
loop.close()
```

This is pretty neat in the sense you could easily have this go execute some I/O bound tasks like fetching data over the local network or Internet. Once you've collected the data you can then set the result with the Future's set_result.

Let's continue with a typical example people like to show which is an application that can fetch the contents of several websites at once. I'm going to use aiohttp which can be installed easily:

$ pip install aiohttp
The code will look like this:

```
import asyncio
import aiohttp

@asyncio.coroutine
def fetch_page(url):
    response = yield from aiohttp.request('GET', url)
    assert response.status == 200
    content = yield from response.read()
    print('URL: {0}:  Content: {1}'.format(url, content))


loop = asyncio.get_event_loop()
tasks = [
    fetch_page('http://google.com'),
    fetch_page('http://cnn.com'),
    fetch_page('http://twitter.com')]
loop.run_until_complete(asyncio.wait(tasks))
loop.close()

for task in tasks:
    print(task)
```

Looking at the results it might look like this still happened synchronously since the results (in the example above) came back in the order we dispatched them. So, let's go ahead and prove that this actually happened asynchronously by injecting a delay on the call to cnn.com . We can modify the code like so:

```
import asyncio
import aiohttp

@asyncio.coroutine
def fetch_page(url, pause=False):
    if pause:
        yield from asyncio.sleep(2)

    response = yield from aiohttp.request('GET', url)
    assert response.status == 200
    content = yield from response.read()
    print('URL: {0}:  Content: {1}'.format(url, content))


loop = asyncio.get_event_loop()
tasks = [
    fetch_page('http://google.com'),
    fetch_page('http://cnn.com', True),
    fetch_page('http://twitter.com')]
loop.run_until_complete(asyncio.wait(tasks))
loop.close()

for task in tasks:
    print(task)
```

The asyncio is a rather large addition to the Python programming language and it's a very important one. If you want to go more in-depth, the official Python documentation page - https://docs.python.org/3/library/asyncio.html - contains more advanced examples (for example using async.queue). A good tutorial on asyncio can be found here: https://www.blog.pythonlibrary.org/2016/07/26/python-3-an-intro-to-asyncio/