Getting started

The **threading** module was first introduced in Python 1.5.2 as an enhancement of the low-level **thread** module. The threading module makes working with threads much easier and allows the program to run multiple operations at once.

Note that the threads in Python work best with I/O operations, such as downloading resources from the Internet or reading files and directories on your computer. If you need to do something that will be CPU intensive, then you will want to look at Python's **multiprocessing** module instead. The reason for this is that Python has the Global Interpreter Lock (GIL) that basically makes all threads run inside of one master thread. Because of this, when you go to run multiple CPU intensive operations with threads, you may find that it actually runs slower. So we will be focusing on what threads do best: I/O operations!

Intro to threads

A thread let's you run a piece of long running code as if it were a separate program. It's kind of like calling **subprocess** except that you are calling a function or class instead of a separate program. I always find it helpful to look at a concrete example. Let's take a look at something that's really simple:

```
import threading


def doubler(number):
    """
    A function that can be used by a thread
    """
    print(threading.currentThread().getName() + '\n')
    print(number * 2)
    print()


if __name__ == '__main__':
    for i in range(5):
        my_thread = threading.Thread(target=doubler, args=(i,))
        my_thread.start()
```

Here we import the threading module and create a regular function called **doubler**. Our function takes a value and doubles it. It also prints out the name of the thread that is calling the function and prints a blank line at the end. Then in the last block of code, we create five threads and start each one in turn. You will note that when we instantiate a thread, we set its **target** to our doubler function and we also pass an argument to the function. The reason the **args** parameter looks a bit odd is that we need to pass a sequence to the doubler function and it only takes one argument, so we need to put a comma on the end to actually create a sequence of one.

Note that if you'd like to wait for a thread to terminate, you would need to call its **join()** method.

When you run this code, you should get the following output:

```
Thread-1

0

Thread-2

2

Thread-3

4

Thread-4

6

Thread-5

8
```

Of course, you normally wouldn't want to print your output to stdout. This can end up being a really jumbled mess when you do. Instead, you should use Python's **logging** module. It's thread-safe and does an excellent job. Let's modify the example above to use the logging module and name our threads while we'll at it:

```
import logging
import threading

def get_logger():
    logger = logging.getLogger("threading_example")
    logger.setLevel(logging.DEBUG)

    fh = logging.FileHandler("threading.log")
    fmt = '%(asctime)s - %(threadName)s - %(levelname)s - %(message)s'
    formatter = logging.Formatter(fmt)
    fh.setFormatter(formatter)

    logger.addHandler(fh)
    return logger


def doubler(number, logger):
    """
    A function that can be used by a thread
    """
    logger.debug('doubler function executing')
    result = number * 2
    logger.debug('doubler function ended with: {}'.format(
        result))


if __name__ == '__main__':
    logger = get_logger()
    thread_names = ['Mike', 'George', 'Wanda', 'Dingbat', 'Nina']
    for i in range(5):
        my_thread = threading.Thread(
            target=doubler, name=thread_names[i], args=(i,logger))
        my_thread.start()
```

The big change in this code is the addition of the **get_logger** function. This piece of code will create a logger that's set to the debug level. It will save the log to the current working directory (i.e. where the script is run from) and then we set up the format for each line logged. The format includes the time stamp, the thread name, the logging level and the message logged.

In the doubler function, we change our **print** statements to logging statements. You will note that we are passing the logger into the doubler function when we create the thread. The reason we do this is that if you instantiated the logging object in each thread, you would end up with multiple logging singletons and your log would have a lot of duplicate lines in it.

Lastly, we name our threads by creating a list of names and then setting each thread to a specific name using the **name** parameter. When you run this code, you should get a log file with the following contents:

```
2016-07-24 20:39:50,055 - Mike - DEBUG - doubler function executing
2016-07-24 20:39:50,055 - Mike - DEBUG - doubler function ended with: 0
2016-07-24 20:39:50,055 - George - DEBUG - doubler function executing
2016-07-24 20:39:50,056 - George - DEBUG - doubler function ended with: 2
2016-07-24 20:39:50,056 - Wanda - DEBUG - doubler function executing
2016-07-24 20:39:50,056 - Wanda - DEBUG - doubler function ended with: 4
2016-07-24 20:39:50,056 - Dingbat - DEBUG - doubler function executing
2016-07-24 20:39:50,057 - Dingbat - DEBUG - doubler function ended with: 6
2016-07-24 20:39:50,057 - Nina - DEBUG - doubler function executing
2016-07-24 20:39:50,057 - Nina - DEBUG - doubler function ended with: 8
```

That output is pretty self-explanatory, so let’s move on. I want to cover one more topic in this section. Namely, subclassing **threading.Thread**. Let’s take this last example and instead of calling Thread directly, we’ll create our own custom subclass. Here is the updated code:

```
import logging
import threading

class MyThread(threading.Thread):

    def __init__(self, number, logger):
        threading.Thread.__init__(self)
        self.number = number
        self.logger = logger

    def run(self):
        """
        Run the thread
        """
        logger.debug('Calling doubler')
        doubler(self.number, self.logger)


def get_logger():
    logger = logging.getLogger("threading_example")
    logger.setLevel(logging.DEBUG)

    fh = logging.FileHandler("threading_class.log")
    fmt = '%(asctime)s - %(threadName)s - %(levelname)s - %(message)s'
    formatter = logging.Formatter(fmt)
    fh.setFormatter(formatter)

    logger.addHandler(fh)
    return logger


def doubler(number, logger):
    """
    A function that can be used by a thread
    """
    logger.debug('doubler function executing')
    result = number * 2
    logger.debug('doubler function ended with: {}'.format(
        result))
```

```
if __name__ == '__main__':
    logger = get_logger()
    thread_names = ['Mike', 'George', 'Wanda', 'Dingbat', 'Nina']

    for i in range(5):
        thread = MyThread(i, logger)
        thread.setName(thread_names[i])
        thread.start()
```

In this example, we just subclassed **threading.Thread**. We pass in the number that we want to double and the logging object as before. But this time, we set the name of the thread differently by calling **setName** on the thread object. We still need to call **start** on each thread, but you will notice that we didn't need to define that in our subclass. When you call **start**, it will run your thread by calling the **run** method. In our class, we call the doubler function to do our processing. The output is pretty much the same except that I added an extra line of output. Go ahead and run it to see what you get.