The threading module

(Optional). This is a higher-level interface for threading. It's modeled after the Java thread facilities. Like the lower-level **thread** module, it's only available if your interpreter was built with thread support.

To create a new thread, subclass the **Thread** class and define the run method. To run such threads, create one or more instances of that class, and call the **start** method. Each instance's **run** method will execute in its own thread.

Example: Using the threading module

```
# File: threading-example-1.py

import threading
import time, random

class Counter:
    def __init__(self):
        self.lock = threading.Lock()
        self.value = 0

    def increment(self):
        self.lock.acquire() # critical section
        self.value = value = self.value + 1
        self.lock.release()
        return value

counter = Counter()

class Worker(threading.Thread):

    def run(self):
        for i in range(10):
            # pretend we're doing something that takes 10-100 ms
            value = counter.increment() # increment global counter
            time.sleep(random.randint(10, 100) / 1000.0)
            print self.getName(), "-- task", i, "finished", value

#
# try it

for i in range(10):
    Worker().start() # start a worker


$ python threading-example-1.py
Thread-1 -- task 0 finished 1
Thread-3 -- task 0 finished 3
Thread-7 -- task 0 finished 8
Thread-1 -- task 1 finished 7
Thread-4 -- task 0 Thread-5 -- task 0 finished 4
finished 5
Thread-8 -- task 0 Thread-6 -- task 0 finished 9
finished 6
...
Thread-6 -- task 9 finished 98
Thread-4 -- task 9 finished 99
Thread-9 -- task 9 finished 100
```

This example also uses **Lock** objects to create a critical section inside the global counter object. If you remove the calls to **acquire** and **release**, it's pretty likely that the counter won't reach 100.