Multithreading - Sample programs:

Index

Threading program:

```
import threading
import time

exitFlag = 0

class myThread (threading.Thread):
    def __init__(self, threadID, name, counter):
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name
        self.counter = counter
    def run(self):
        print "Starting " + self.name
        print_time(self.name, self.counter, 5)
        print "Exiting " + self.name

def print_time(threadName, delay, counter):
    while counter:
        if exitFlag:
            thread.exit()
        time.sleep(delay)
        print "%s: %s" % (threadName, time.ctime(time.time()))
        counter -= 1

# Create new threads
thread1 = myThread(1, "Thread-1", 1)
thread2 = myThread(2, "Thread-2", 2)

# Start new Threads
thread1.start()
thread2.start()

print "Exiting Main Thread"
```

Sample Output:

```
Starting Thread-1
Starting Thread-2
Exiting Main Thread
Thread-1: Thu Mar 21 09:10:03 2013
Thread-1: Thu Mar 21 09:10:04 2013
Thread-2: Thu Mar 21 09:10:04 2013
Thread-1: Thu Mar 21 09:10:05 2013
Thread-1: Thu Mar 21 09:10:06 2013
Thread-2: Thu Mar 21 09:10:06 2013
Thread-1: Thu Mar 21 09:10:07 2013
Exiting Thread-1
Thread-2: Thu Mar 21 09:10:08 2013
Thread-2: Thu Mar 21 09:10:10 2013
Thread-2: Thu Mar 21 09:10:12 2013
Exiting Thread-2
```

Daemon vs. Non-Daemon Threads

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, call its setDaemon() method with a boolean argument. The default is for threads to not be daemons, so passing True turns the daemon mode on.

```
import threading
import time
import logging

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

def daemon():
    logging.debug('Starting')
    time.sleep(2)
    logging.debug('Exiting')

d = threading.Thread(name='daemon', target=daemon)
d.setDaemon(True)

def non_daemon():
    logging.debug('Starting')
    logging.debug('Exiting')

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

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

Notice that the output does not include the "Exiting" message from the daemon thread, since all of the non-daemon threads (including the main thread) exit before the daemon thread wakes up from its two second sleep.

```
$ python threading_daemon.py
```

Sample Output:

```
(daemon    ) Starting
(non-daemon) Starting
(non-daemon) Exiting
```

To wait until a daemon thread has completed its work, use the join() method.

```
import threading
import time
import logging

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

def daemon():
    logging.debug('Starting')
    time.sleep(2)
    logging.debug('Exiting')

d = threading.Thread(name='daemon', target=daemon)
d.setDaemon(True)

def non_daemon():
    logging.debug('Starting')
    logging.debug('Exiting')

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

d.start()
t.start()

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

Waiting for the daemon thread to exit using join() means it has a chance to produce its "Exiting" message.

```
$ python threading_daemon_join.py
```

Sample Output:

```
(daemon    ) Starting
(non-daemon) Starting
(non-daemon) Exiting
(daemon    ) Exiting
```

By default, join() blocks indefinitely. It is also possible to pass a timeout argument (a float 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.

```
import threading
import time
import logging

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

def daemon():
    logging.debug('Starting')
    time.sleep(2)
    logging.debug('Exiting')

d = threading.Thread(name='daemon', target=daemon)
d.setDaemon(True)

def non_daemon():
    logging.debug('Starting')
    logging.debug('Exiting')

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

d.start()
t.start()

d.join(1)
print 'd.isAlive()', d.isAlive()
t.join()
```

Since the timeout passed is less than the amount of time the daemon thread sleeps, the thread is still “alive” after join() returns.

```
$ python threading_daemon_join_timeout.py
```

Sample Output:

```
(daemon    ) Starting
(non-daemon) Starting
(non-daemon) Exiting
d.isAlive() True
```

Locking:

The threading module provided with Python includes a simple-to-implement locking mechanism that allows you to synchronize threads. A new lock is created by calling the *Lock()* method, which returns the new lock.

The *acquire(blocking)* method of the new lock object is used to force threads to run synchronously. The optional *blocking* parameter enables you to control whether the thread waits to acquire the lock.

If *blocking* is set to 0, the thread returns immediately with a 0 value if the lock cannot be acquired and with a 1 if the lock was acquired. If blocking is set to 1, the thread blocks and wait for the lock to be released.

The *release()* method of the new lock object is used to release the lock when it is no longer required.

```
import threading
import time

class myThread (threading.Thread):
    def __init__(self, threadID, name, counter):
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name
        self.counter = counter
    def run(self):
        print "Starting " + self.name
        # Get lock to synchronize threads
        threadLock.acquire()
        print_time(self.name, self.counter, 3)
        # Free lock to release next thread
        threadLock.release()

def print_time(threadName, delay, counter):
    while counter:
        time.sleep(delay)
        print "%s: %s" % (threadName, time.ctime(time.time()))
        counter -= 1

threadLock = threading.Lock()
threads = []

# Create new threads
thread1 = myThread(1, "Thread-1", 1)
thread2 = myThread(2, "Thread-2", 2)

# Start new Threads
thread1.start()
thread2.start()

# Add threads to thread list
threads.append(thread1)
threads.append(thread2)

# Wait for all threads to complete
```

```
for t in threads:
   t.join()
print "Exiting Main Thread"
```

Sample Output:

```
Starting Thread-1
Starting Thread-2
Thread-1: Thu Mar 21 09:11:28 2013
Thread-1: Thu Mar 21 09:11:29 2013
Thread-1: Thu Mar 21 09:11:30 2013
Thread-2: Thu Mar 21 09:11:32 2013
Thread-2: Thu Mar 21 09:11:34 2013
Thread-2: Thu Mar 21 09:11:36 2013
Exiting Main Thread
```

Lock and Release Lock

In this chapter, we'll learn how to control access to shared resources. The control is necessary to prevent corruption of data. In other words, to guard against simultaneous access to an object, we need to use a **Lock object**.

A primitive **lock** is a synchronization primitive that is not owned by a particular thread when locked. In Python, it is currently the lowest level synchronization primitive available, implemented directly by the **_thread** extension module.
- https://docs.python.org/3/library/threading.html.

A primitive lock is in one of two states, "locked" or "unlocked". It is created in the unlocked state. It has two basic methods, **acquire()** and **release()**. When the state is unlocked, **acquire()** changes the state to locked and returns immediately. When the state is locked, **acquire()** blocks until a call to **release()** in another thread changes it to unlocked, then the **acquire()** call resets it to locked and returns. The **release()** method should only be called in the locked state; it changes the state to unlocked and returns immediately. If an attempt is made to release an unlocked lock, a RuntimeError will be raised.

Here is our example code using the **Lock** object. In the code the **worker()** function increments a **Counter** instance, which manages a Lock to prevent two threads from changing its internal state at the same time.

```
import threading
import time
import logging
import random

logging.basicConfig(level=logging.DEBUG,
                    format='(%(threadName)-9s) %(message)s',)

class Counter(object):
   def __init__(self, start = 0):
      self.lock = threading.Lock()
      self.value = start
   def increment(self):
      logging.debug('Waiting for a lock')
```

```
        self.lock.acquire()
        try:
            logging.debug('Acquired a lock')
            self.value = self.value + 1
        finally:
            logging.debug('Released a lock')
            self.lock.release()

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

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

Output:

```
(Thread-1 ) Sleeping 0.04
(MainThread) Waiting for worker threads
(Thread-2 ) Sleeping 0.11
(Thread-1 ) Waiting for a lock
(Thread-1 ) Acquired a lock
(Thread-1 ) Released a lock
(Thread-1 ) Sleeping 0.30
(Thread-2 ) Waiting for a lock
(Thread-2 ) Acquired a lock
(Thread-2 ) Released a lock
(Thread-2 ) Sleeping 0.27
(Thread-1 ) Waiting for a lock
(Thread-1 ) Acquired a lock
(Thread-1 ) Released a lock
(Thread-1 ) Done
(Thread-2 ) Waiting for a lock
(Thread-2 ) Acquired a lock
(Thread-2 ) Released a lock
(Thread-2 ) Done
(MainThread) Counter: 4
```

Another example

In this example, **worker()** tries to acquire the lock three separate times, and counts how many attempts it has to make to do so. In the mean time, **locker()** cycles between holding and releasing the lock, with short sleep in each state used to simulate load.

```
import threading
import time
import logging

logging.basicConfig(level=logging.DEBUG,
                    format='(%(threadName)-9s) %(message)s',)

def locker(lock):
    logging.debug('Starting')
    while True:
        lock.acquire()
        try:
            logging.debug('Locking')
            time.sleep(1.0)
        finally:
            logging.debug('Not locking')
            lock.release()
        time.sleep(1.0)
    return

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')
        acquired = lock.acquire(0)
        try:
            num_tries += 1
            if acquired:
                logging.debug('Try #%d : Acquired',  num_tries)
                num_acquires += 1
            else:
                logging.debug('Try #%d : Not acquired', num_tries)
        finally:
            if acquired:
                lock.release()
    logging.debug('Done after %d tries', num_tries)

if __name__ == '__main__':
    lock = threading.Lock()

    locker = threading.Thread(target=locker, args=(lock,), name='Locker')
    locker.setDaemon(True)
    locker.start()

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

Output:

```
(Locker   ) Starting
(Locker   ) Locking
(Worker   ) Starting
(Worker   ) Trying to acquire
(Worker   ) Try #1 : Not acquired
(Locker   ) Not locking
(Worker   ) Trying to acquire
(Worker   ) Try #2 : Acquired
(Worker   ) Trying to acquire
(Worker   ) Try #3 : Acquired
(Locker   ) Locking
(Worker   ) Trying to acquire
(Worker   ) Try #4 : Not acquired
(Worker   ) Trying to acquire
(Worker   ) Try #5 : Not acquired
(Locker   ) Not locking
(Worker   ) Trying to acquire
(Worker   ) Try #6 : Acquired
(Worker   ) Done after 6 tries
```

RLock

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)
print "print this if not blocked..."
```

Output:

```
First try : True
Second try: False
print this if not blocked...
```

As we can see from the code, since both functions are using the same global lock, and one calls the other, the second acquisition fails and would have blocked using the default arguments to **acquire(blocking=True, timeout=-1)** as shown in the following code and output:

```
import threading

lock = threading.Lock()

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

print "print this if not blocked..."
```

This code is blocking:

```
First try : True
Second try:
```

So, as in the example above, in a situation where separate code from the same thread needs to "re-acquire" the lock, we need to use an **threading.RLock** instead of a simple **threading.Lock()**:

```
import threading

lock = threading.RLock()

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

Output shows that we're able to "re-acquire" the lock:

```
First try : True
Second try: 1
```