Part 5: The Threading Module

Part 5: The Threading Module

Starting threads doesn't take much:

```
import sys
import threading
import time

def func():
    for i in range(5):
        print("hello from thread %s" % threading.current_thread().name)
        time.sleep(1)

threads = []
for i in range(3):
    thread = threading.Thread(target=func, args=())
    thread.start()
    threads.append(thread)
```

- The process will exit when the last non-daemon thread exits.
- A thread can be specified as a daemon thread by setting its daemon attribute: `thread.daemon = True`
- daemon threads get cut off at program exit, without any opportunity for cleanup. But you don't have to track and manage them. Useful for things like garbage collection, network keepalives, ..
- You can block and wait for a thread to exit with thread.join()

Subclassing Thread

You can add threading capability to your own classes.

Subclass Thread and implement the run method.

```
import threading

class MyThread(threading.Thread):

    def run(self):
        print("hello from %s" % threading.current_thread().name)

thread = MyThread()
thread.start()
```

Race Conditions

In the last example we saw threads competing for access to stdout.

Worse, if competing threads try to update the same value, we might get an unexpected race condition.

Race conditions occur when multiple statements need to execute atomically, but get interrupted midway.

See race_condition.py in your repository.

No race condition

Thread 1	Thread 2		Integer value
			0

			0
read value		←	0
increase value			0
write back		→	1
	read value	←	1
	increase value		1
	write back	→	2

Race Condition!

Thread 1	**Thread 2**		**Integer value**
			0
read value		←	0
	read value	←	0
increase value			0
	increase value		0
write back		→	1
	write back	→	1

http://en.wikipedia.org/wiki/Race_condition

Deadlocks

Synchronization and Critical Sections are used to control race conditions

But they introduce other potential problems...

like: http://en.wikipedia.org/wiki/Deadlock

“A deadlock is a situation in which two or more competing actions are each waiting for the other to finish, and thus neither ever does.”

When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone

See also *Livelock*:

Two people meet in a narrow corridor, and each tries to be polite by moving aside to let the other pass, but they end up swaying from side to side without making any progress because they both repeatedly move the same way at the same time.

Locks

Lock objects allow threads to control access to a resource until they're done with it.

This is known as mutual exclusion, often called "mutex".

A Lock has two states: locked and unlocked.

If multiple threads have access to the same Lock, they can police themselves by calling its `.acquire()` and `.release()` methods

If a Lock is locked, .acquire will block until it becomes unlocked.

These threads will wait in line politely for access to the statements in f()

Mutex locks (`threading.Lock`)

- Probably the most common.
- Only one thread can modify shared data at any given time.
- Thread determines when unlocked.
- Must put lock/unlock around critical code in ALL threads.
- Difficult to manage.

Easiest with context manager:

```
x = 0
x_lock = threading.Lock()

# Example critical section
with x_lock:
    # statements using x
```

Only one lock per thread! (or risk mysterious deadlocks).

Or use RLock for code-based locking (locking function/method execution rather than data access).

```
import threading
import time

lock = threading.Lock()

def f():
    lock.acquire()
    print("%s got lock" % threading.current_thread().name)
    time.sleep(1)
    lock.release()

threading.Thread(target=f).start()
threading.Thread(target=f).start()
threading.Thread(target=f).start()
```

Nonblocking Locking

`.acquire()` will return True if it successfully acquires a lock

Its first argument is a boolean which specifies whether a lock should block or not. The default is `True`

```
import threading
```

```
lock = threading.Lock()
lock.acquire()
if not lock.acquire(False):
    print("couldn't get lock")
lock.release()
if lock.acquire(False):
    print("got lock")
```

`threading.RLock` - Reentrant Lock

Useful for recursive algorithms, a thread-specific count of the locks is maintained

A reentrant lock can be acquired multiple times by the same thread

`Lock.release()` must be called the same number of times as `Lock.acquire()` by that thread

`threading.Semaphore`

Like an `RLock`, but in reverse

A Semaphore is given an initial counter value, defaulting to 1

Each call to `acquire()` decrements the counter, `release()` increments it

If `acquire()` is called on a Semaphore with a counter of 0, it will block until the Semaphore counter is greater than 0.

Useful for controlling the maximum number of threads allowed to access a resource simultaneously

Semaphore

Events (`threading.Event`)

- Threads can wait for particular event
- Setting an event unblocks all waiting threads

Common use: barriers, notification

Condition (`threading.Condition`)

- Combination of locking/signaling
- lock protects code that establishes a “condition” (e.g., data available)