

Multithreaded Programming In Python

How, When, and Why (Not)

by Wesley J. Chun

wescpy@gmail.com

<http://cyberwebconsulting.com>

Fall 2009

Roadmap

- Intro to threading
- Threading and Python
- Global Interpreter Lock
- Managing Threads
- Thread Objects and Methods
- Threading Alternatives
- Examples from *Core Python Programming* (Chapter 18)

Threading

- Why not do unrelated tasks in parallel?
- Asynchronous & discrete tasks are candidates
- Processes vs. Threads
 - Processes require more (OS) overhead
 - Threads share the same memory space
- Threads cost less but need existing tools
 - Synchronization primitives (execution)
 - Locking (data)

Thread Support in Python

- Supported on Win, Linux, Irix, Solaris 2
 - Also any POSIX "pthread" implementation
- There is a lower-level `thread` module
 - Renamed to "`_thread`" in Python 3
 - Avoid using it
 - No suite of synchronization primitives
 - Only provides simple locking
- Only use the higher-level `threading` module

Python Thread Gotchas

- Threads must exit themselves
 - Cannot go "kill a thread"
 - Need to create some global "exit" flag
 - Make all (appropriate) threads watch this flag
- Multiple threads is nice BUT
 - Python is a single-threaded process
 - Has a global interpreter lock (GIL)
 - Means only one thread exec'd by VM

More about the GIL

- GIL necessary because Python not thread-safe
 - Could be race conditions on reference counts
 - Threads must acquire the GIL before they can run
- The GIL prevents true parallelism
 - Releasing the GIL == more parallelism
 - Threads share the same memory space
 - `time.sleep` : a cheap way to yield control
- I/O-bound applications win over CPU-bound
 - I/O system calls release the GIL
 - C extensions can also release the GIL

Creating and Running Threads

- Recipe is fairly straightforward:
 - Create function(s) to run in thread(s)
 - Will be called by `run` method (or make your own)
 - Setup the arguments to pass to the function
 - Create thread(s)
 - "Call" `threading.Thread` to instantiate (but not start)
 - Start thread(s)
 - Call `start` method (which starts thread & calls `run`)
 - (optionally) Wait for thread(s) to finish
 - Call `join` method (which blocks caller until thread done)

Example

```
import threading

def bar(text):
    print text

t = threading.Thread(target=bar,
    args=('foo',))
t.start()
t.join()
```

Thread Naming

- Threads have names
- Default names are boring, i.e., "Thread-123"
- You can set (then get) names of threads with methods:
 - `thread.setName`
 - `thread.getName`
- Better yet, stop thinking like Java and use `thread.name`

Are you alive?

- Threads are "alive" when `start` returns after launching thread
- Can monitor "status" with method
 - `thread.isAlive` or `thread.is_alive`
- Can get list of all alive threads with `threading.enumerate`
 - Also includes "daemon threads"
- Threads that are *not* alive
 - Those that have terminated
 - Those that have not yet been started

Daemon Threads

- Threads considered to be "service" not core functionality
- A Python program exits if all non-daemon threads have completed
- Any daemon threads will be killed off
- You can set (then check) a thread as daemon with
`thread.setDaemon`
`thread.isDaemon`
- Better yet, stop thinking like Java and use
`thread.daemon`

Longer Threading Example (mtsleap3.py)

```
import threading
from time import sleep, ctime

loops = ( 4, 2 )
def loop(nloop, nsec):
    print 'start loop', nloop, 'at:', ctime()
    sleep(nsec)
    print 'loop', nloop, 'done at:', ctime()

print 'starting threads...'
threads = []
nloops = range(len(loops))

for i in nloops:
    t = threading.Thread(target=loop, args=(i, loops[i]))
    threads.append(t)
for i in nloops: threads[i].start() # start threads
for i in nloops:
    threads[i].join() # wait 4 threads 2 finish
print 'all DONE at:', ctime()
```

Threading Example Output

```
$ mtsleep3.py
starting threads...
start loop 0 at: Sun Aug 13 18:16:38 2006
start loop 1 at: Sun Aug 13 18:16:38 2006
loop 1 done at: Sun Aug 13 18:16:40 2006
loop 0 done at: Sun Aug 13 18:16:42 2006
all DONE at: Sun Aug 13 18:16:42 2006
```

- [Next examples use same imports & declarations]

More on Creating Threads

- When creating threads, you have 3 alternatives:
 - Create `Thread` instance, passing in function
 - We did this for first example
 - Create `Thread` instance, passing in callable class instance
 - Subclass `Thread` and create subclass instance

Improving our Threading Code

- We can improve on first example by adding more OO
 - Classes give you more power than just functions
- The second example introduces a thread class
 - We pass our instance into Thread
 - Note that instances must now be callable
- Final example frees us from instantiating Thread
 - Code is cleaner and do not need a callable instance

Thread Class Example (mtsleap4.py)

```
class ThreadFunc:
    def __init__(self, func, args, name=''):
        self.name = name
        self.func = func
        self.args = args

    def __call__(self):
        self.func(*self.args)

print 'starting threads...'
threads = []
nloops = range(len(loops))
for i in nloops:
    t = threading.Thread(target=ThreadFunc(loop, (i, loops[i]),
        loop.__name__))
    threads.append(t)
for i in nloops: threads[i].start()
for i in nloops: threads[i].join()
print 'all DONE at:', ctime()
```

Thread Subclass Example (mtsleap5.py)

```
class MyThread(threading.Thread):
    def __init__(self, func, args, name=''):
        threading.Thread.__init__(self)
        self.name = name
        self.func = func
        self.args = args

    def run(self):
        self.func(*self.args)

print 'starting threads...'
threads = []
nloops = range(len(loops))
for i in nloops:
    t = MyThread(loop, (i, loops[i]), loop.__name__)
    threads.append(t)
for i in nloops: threads[i].start()
for i in nloops: threads[i].join()
print 'all DONE at:', ctime()
```

Thread Class Methods

- `start` - commence thread execution
- `run` - thread executes this (the thread's "work")
- `join` - wait until the thread terminates
- `getName` - returns the thread's name
- `setName` - assigns the thread's name
- `isAlive` - Boolean indicating if thread is running
- `isDaemon` - Boolean of whether thread is daemonic
- `setDaemon` - set the thread's daemon flag

Other threading Module Objects

- `Lock` - Primitive lock object
- `RLock` - Re-entrant lock (can be reacquired by lock owner)
- `Condition` - Condition variable: temporary lock release until a condition has been met
- `Semaphore` - Limited resource synchronization based on a counter
- `BoundedSemaphore` - Restricted semaphores whose counter cannot exceed orig value
- `Event` - Flag-based synchronization: threads block until flag is `True`
- `Timer` - Like `Thread` but start after an elapsed time

Queue Module and Class

- Thread-safe synchronized data structure
- Available via the Queue module
 - Renamed to queue in Python 3
- `Queue.Queue` : FIFO queue structure
 - Takes a `maxsize` int
 - Non-positive `maxsize` == unsized queue
- New in Python 2.6
 - `Queue.LifoQueue` : LIFO queue structure
 - `Queue.PriorityQueue` : queue structure sorted by weight
- `collections.deque` : thread-safe high-performing double-ended queue

Executing Other Code (most non-concurrent)

`os.system()`

- Executes an external cmd... any output shows up too!

`os.popen()`

- Like `os.system` but allows you to read output
- "Opens" a cmd like a file with standard input access

`os.fork()`, `os.exec*()`

- Create and run a new process

`os.popen[2-4]()`

- Gives ability to read/write from/to cmds

■ `subprocess` Module

- Intended to replace `os.system`, `os.popen*`

Alternative Standard Library Services

- `multiprocessing` module
 - Avoid the GIL and take advantage of multi-core CPUs
 - Shares same API as `threading`
 - Has an underlying forking mechanism
 - Added in 2.6; also see PEP-371
- `asyncore` and `asynchat`
 - Manage multiple I/O streams w/o threads
 - Uses `select` module to determine "active" connections

Alternative 3rd-party Packages

- Stackless <http://stackless.com>
 - Breaks C stack limitation allowing for many "microthreads"
- Twisted <http://twistedmatrix.com>
 - Asynchronous/event-driven networking system
- Greenlets <http://undefined.org/python/#greenlet>
 - Think coroutines; similar to Stackless except...
 - You have more control over scheduling
 - It is a module/package instead of VM reimplementation
- Parallel Python <http://parallelpython.com>
 - Concurrent execution on SMP and clusters
- More at <http://wiki.python.org/moin/ParallelProcessing>

Questions?

FINIS