

Python Threads

Aahz

`aahz@pobox.com`

`http://starship.python.net/crew/aahz/`

Powered by PythonPoint

`http://www.reportlab.com/`

Meta Tutorial

- I'm hearing-impaired
Please write questions if at all possible
- Take notes - or not
- Pop Quiz
- Slideshow on web

Contents

- Goal: Use Threads!
- Thread Overview
- Python's Thread Library
- Two Applications
 - Web Spider
 - GUI Background Thread

Part 1

- What are threads?
- GIL
- Python threads

Generic Threads

- Similar to processes
- Shared memory
- Light-weight
- Difficult to set up
 - Especially cross-platform

Why Use Threads?

- Efficiency/speed
- Responsiveness
- Algorithmic simplicity

Python Threads

- Class-based
 - Use `threading`, not `thread`
- Cross-platform
- Thread Library

Python 1.5.2 vs. 2.0

- Compile `--with-thread`
Except on MS Windows and some Linux distributions
- Multi-CPU bug
Creating/destroying large numbers of threads

GIL

- Global Interpreter Lock (GIL)

- Full Documentation:

`http://www.python.org/doc/current/api/threads.html`

- Only one Python thread can run
- GIL is your friend (really!)

GIL in Action

- Which is faster?

One Thread

```
total = 1
for i in range(10000):
    total += 1
total = 1
for i in range(10000):
    total += 1
```

Two Threads

```
total = 1
for i in range(10000):
    total += 1
```

```
total = 1
for i in range(10000):
    total += 1
```

Dealing with GIL

- `sys.setcheckinterval()`
(default 10)
- C extensions can release GIL
- Blocking I/O releases GIL
So does `time.sleep(!= 0)`
- Multiple Processes

Performance Tip

- `python -O`

Also set `PYTHONOPTIMIZE`

15% performance boost

Removes bytecodes (`SET_LINENO`)

Fewer context switches!

GIL and C Extensions

- Look for macros:
`Py_BEGIN_ALLOW_THREADS`
`Py_END_ALLOW_THREADS`
- Some common extensions:
mxODBC - yes
NumPy - no

Share External Objects

- Files, GUI, DB connections

Share External Objects

- Files, GUI, DB connections

Don't

Share External Objects

- Files, GUI, DB connections

Don't

- Partial exception: `print`
- Still need to share?
Use worker thread

Create Python Threads

- Subclass `threading.Thread`
- Override `__init__()` and `run()`
- Do *not* override `start()`
- In `__init__()`, call `Thread.__init__()`

Using Python Threads

- Instantiate thread object

```
t = MyThread()
```

- Start the thread

```
t.start()
```

- Methods/attrs from outside thread

```
t.put('foo')
```

```
if t.done:
```

Non-threaded Example

```
class Retriever:
    def __init__(self, URL):
        self.URL = URL
        self.page = self.getPage()

retriever = Retriever('http://www.foo.com/')
URLs = retriever.getLinks()
```

Threaded Example

```
from threading import Thread

class Retriever(Thread):
    def __init__(self, URL):
        Thread.__init__(self)
        self.URL = URL
    def run(self):
        self.page = self.getPage()

retriever = Retriever('http://www.foo.com/')
retriever.start()
while retriever.isAlive():
    time.sleep(1)
URLs = retriever.getLinks()
```

Multiple Threads

```
seeds = ['http://www.foo.com/',
         'http://www.bar.com/',
         'http://www.baz.com/']
threadList = []
URLs = []

for seed in Seed:
    retriever = Retriever(seed)
    retriever.start()
    threadList.append(retriever)

for retriever in threadList:
    # join() is more efficient than sleep()
    retriever.join()
    URLs += retriever.getLinks()
```

`import` Editorial

- How to import
from threading import Thread, Semaphore
or
import threading
- Don't use
from threading import *

Thread Methods

- Module functions:
 - `activeCount ()` (not useful)
 - `enumerate ()` (not useful)
- Thread object methods:
 - `start ()`
 - `join ()` (somewhat useful)
 - `isAlive ()` (not useful)
 - `isDaemon ()`
 - `setDaemon ()`

Unthreaded Spider

- `SingleThreadSpider.py`
- `Compare Tools/webchecker/`

Brute Thread Spider

- `BruteThreadSpider.py`
- Few changes from `SingleThreadSpider.py`
- Spawn one thread per retrieval
- Inefficient polling in main loop

Part 2

- Recap
- Thread Theory
- Python Thread Library

Recap Part 1

- GIL
- Creating threads
- Brute force threads

Thread Order

- Non-determinate

Thread 1

```
print "a, ",  
print "b, ",  
print "c, ",
```

Thread 2

```
print "1, ",  
print "2, ",  
print "3, ",
```

- Sample output

```
1, a, b, 2, c, 3,  
a, b, c, 1, 2, 3,  
1, 2, 3, a, b, c,  
a, b, 1, 2, 3, c,
```

Thread Communication

- **Critical Section Lock**
 - Protects shared memory
 - Only one thread accesses chunk of code
aka "mutex", or "atomic operation"
- **Wait/Notify**
 - Synchronizes actions between threads
 - Threads wait for each other to finish a task
 - More efficient than polling

Thread Library

- `Lock ()`
- `RLock ()`
- `Semaphore ()`
- `Condition ()`
- `Event ()`
- `Queue.Queue ()`

Lock()

- Basic building block
- Methods
 - `acquire(blocking)`
 - `release()`

Critical Section Lock

Thread 1

```
mutex.acquire()  
if myList:  
    work = myList.pop()  
mutex.release()  
...  
...  
...  
...
```

Thread 2

```
...  
...  
...  
...  
mutex.acquire()  
if len(myList)<10:  
    myList.append(work)  
mutex.release()
```

GIL and Shared Vars

- **Safe: one bytecode**
Single operations against Python basic types (e.g. appending to a list)
- **Unsafe**
Multiple operations against Python variables (e.g. checking the length of a list before appending) or any operation that involves a callback to a class (e.g. the `__getattr__` hook)

GIL example

- Mutex only one thread

Thread 1

```
myList.append(work)  
...  
...  
...
```

Thread 2

```
mutex.acquire()  
if myList:  
    work = myList.pop()  
mutex.release()
```

dis this

- disassemble source to byte codes
- Thread-unsafe statement
 - If a single Python statement uses the same shared variable across multiple byte codes, or if there are multiple mutually-dependent shared variables, that statement is not thread-safe

Misusing Lock()

- Lock () steps on itself

```
mutex = Lock()  
mutex.acquire()  
...  
mutex.acquire()    # OOPS!
```

Synch Two Threads

```
class Synchronize:
    def __init__(self):
        self.lock = Lock()
    def wait(self):
        self.lock.acquire()
        self.lock.acquire()
        self.lock.release()
    def notify(self):
        self.lock.release()
```

Thread 1

```
self.synch.wait()
...
...
self.synch.notify()
```

Thread 2

```
...
self.synch.notify()
self.synch.wait()
...
```

RLock()

- Mutex only
Other threads cannot release RLock()
- Recursive
- Methods
`acquire(blocking)`
`release()`

Using RLock()

```
mutex = RLock()  
mutex.acquire()  
...  
mutex.acquire()    # Safe  
...  
mutex.release()  
mutex.release()
```

Thread 1

```
mutex.acquire()  
self.update()  
mutex.release()  
...  
...  
...
```

Thread 2

```
...  
...  
...  
mutex.acquire()  
self.update()  
mutex.release()
```

Semaphore()

- Restricts number of running threads
In Python, primarily useful for simulations
(but consider using microthreads)

- Methods

`Semaphore(value)`

`acquire(blocking)`

`release()`

Condition()

- **Methods**

`Condition(lock)`

`acquire(blocking)`

`release()`

`wait(timeout)`

`notify()`

`notifyAll()`

Using Condition()

- Must use lock

```
cond = Condition()  
cond.acquire()  
cond.wait()           # or notify()/notifyAll()  
cond.release()
```

- Avoid *timeout*
Creates polling loop, so inefficient

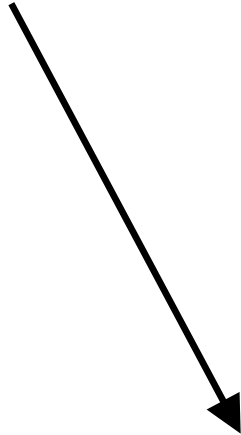
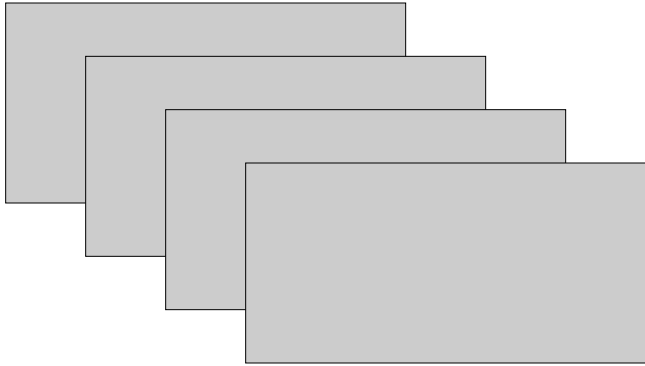
Event()

- Thin wrapper for `Condition()`
 - Don't have to mess with lock
 - Only uses `notifyAll()`, so can be inefficient
- Methods
 - `set()`
 - `clear()`
 - `isSet()`
 - `wait(timeout)`

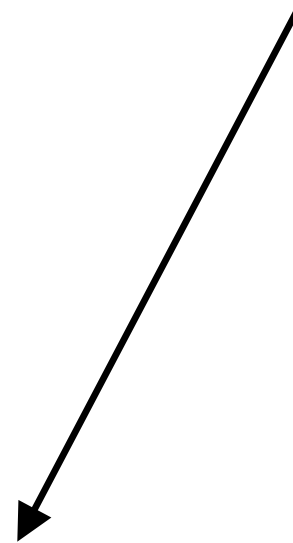
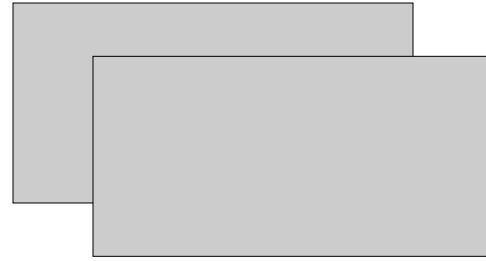
TMTOWTDI

- Perl:
There's More Than One Way To Do It
- Python:
There should be one - and preferably only one - obvious way to do it
- Threads more like Perl

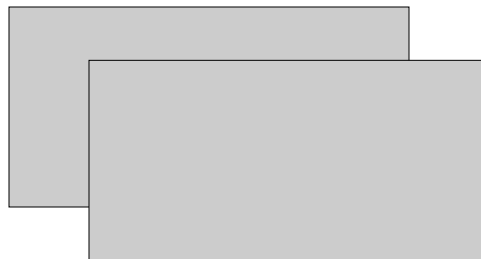
Body factory



Wheel factory



Assembly



Factory Objects 1

Body

```
body.list  
body.rlock  
body.event  
assembly.event
```

Assembly

```
body.list  
body.rlock  
body.event  
wheels.list  
wheels.rlock  
wheels.event  
assembly.rlock  
assembly.event
```

Wheels

```
wheels.list  
wheels.rlock  
wheels.event  
assembly.event
```

Queue()

- Does not use `threading`
Can be used with `thread`
- Designed for subclassing
Can implement stack, priority queue, etc.
- Simple!
Handles *both* data protection and synchronization

Queue() Objects

- **Methods**

`Queue(maxsize)`

`put(item, block)`

`get(block)`

`qsize()`

`empty()`

`full()`

- **Raises exception when non-blocking**

Using Queue()

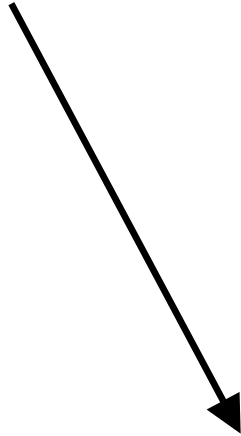
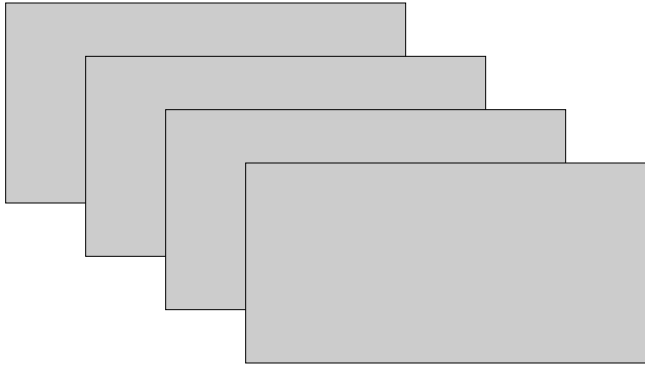
Thread 1

```
output = self.doWork()  
queue.put(output)  
...  
...  
...  
self.input = queue.get()
```

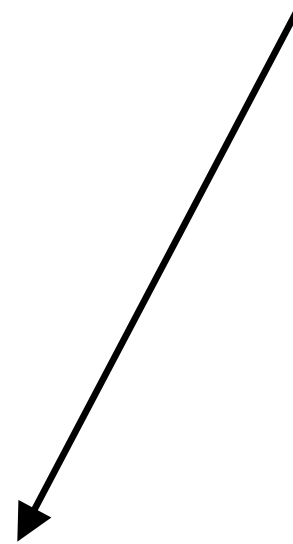
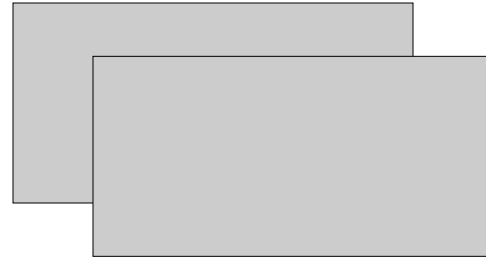
Thread 2

```
...  
...  
self.input = queue.get()  
output = self.doWork()  
queue.put(output)  
...
```

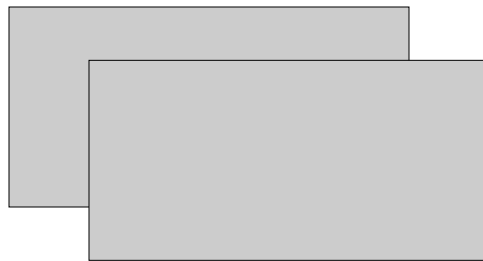
Body factory



Wheel factory



Assembly



Factory Objects 2

Body

body.queue

Assembly

body.queue

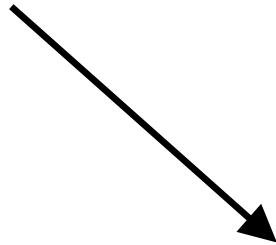
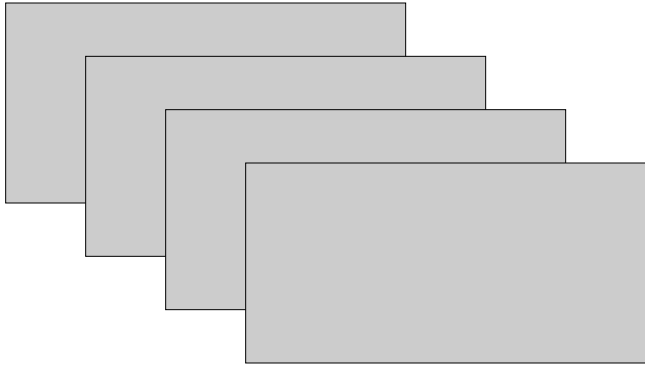
wheels.queue

assembly.rlock

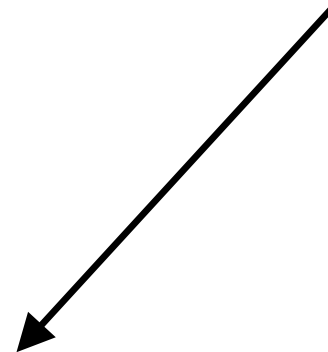
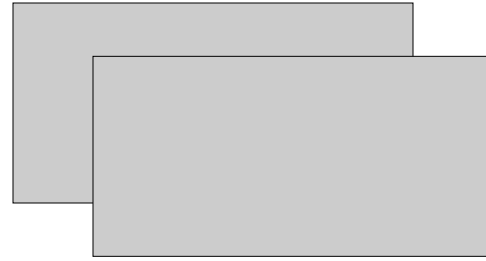
Wheels

wheels.queue

Body factory



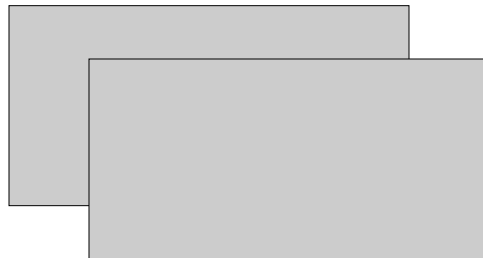
Wheel factory



Packager



Assembly



Factory Objects 3

Body

`body.queue`

Wheels

`wheels.queue`

Packager

```
while 1:  
    body = self.body.queue.get()  
    wheels = self.wheels.queue.get()  
    self.assembly.put( (body,wheels) )
```

Assembly

`assembly.queue`

Part 3

- Recap
- Using Queues
 - spider
 - GUI (Tkinter)

Recap Part 2

- Data protection and synchronization
- Python Thread Library
- Queues are good

Spider w/Queue

- `ThreadPoolSpider.py`
- Two queues
 - Pass work to thread pool
 - Get links back from thread pool
- Queue for both data and events

Tkinter Intro

This space intentionally left blank

GUI building blocks

- **Widgets**
Windows, buttons, checkboxes, text entry, listboxes
- **Events**
Widget activation, keypress, mouse movement, mouse click, timers

Widgets

- Geometry manager
- Register callbacks

Events

- Event loop
- Trigger callbacks

Tkinter resources

- Web

`http://www.python.org/topics/tkinter/doc.html`

- Books

Python and Tkinter Programming, John E. Grayson

Fibonacci

- `Fibonacci.py`
- UI freezes during calc
- Frequent screen updates slow calc

Threaded Fibonacci

- `FibThreaded.py`
- Tkinter needs to poll
 Use `after` event
- Single-element queue
 Use in non-blocking mode to minimize updates
- Must use "Quit" button

Pop Quiz 1

How are threads and processes similar and different?

What is the GIL?

In what ways does the GIL make thread programming easier and harder?

How do you create a thread in Python?

What should not be shared between threads?

What are "brute force" threads?

Pop Quiz 2

Explain what each of the following is used for:

Lock()

RLock()

Semaphore()

Condition()

Event()

Queue.Queue()

Why are queues great?