Python Threads

Aahz

aahz@pobox.com

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

Powered by PythonPoint

http://www.reportlab.com/

title:

Meta Tutorial

- I'm hearing-impaired

  Please write questions if at all possible

- Pop Quiz
- Slides and scripts on web

Contents

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

Part 1: Thread Intro

- What are threads?
- GIL
- Python threads
- Brute force threads

title: Part 1

Generic Threads

- Similar to processes
- Shared memory
- Light-weight
- Difficult to set up

  Especially cross-platform

title: Generic Threads

Why Use Threads?

- Efficiency/speed

  multiple CPUs, parallelize blocking I/O

- Responsiveness

  e.g. background thread for GUI

- Algorithmic simplicity

  simulations, data passing

  (mostly skipped in this tutorial)

Python Threads

- Class-based

  Use `threading`, not `thread`

- Cross-platform, OS-level
- Thread Library

Python 1.5.2

- `configure --with-thread`
  - Except on MS Windows and some Linux distributions
- Multi-CPU bug
  - Creating/destroying large numbers of threads
- Upgrade to 2.x

GIL

- Global Interpreter Lock (GIL)
- Full Documentation:

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

- Only one Python thread can run
  - Even with multiple CPUs
- GIL is your friend (really!)

title: GIL

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

title: GIL in action

Dealing with GIL

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

title: Dealing with GIL

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__()`

Use Python Threads

- Instantiate thread object

```
    t = MyThread()
```

- Start the thread

```
    t.start()
```

- Methods/attribs from outside thread

```
    t.put('foo')
    if t.done:
```

Non-threaded Example

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

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

title: Multiple Threads

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 Force Threads

- Quick-convert to multiple threads
- Need worker class
  - Just inherit from `threading.Thread`
- One instance per work unit

Brute Thread Spider

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

Recap Part 1

- GIL
- Creating threads
- Brute force threads

Part 2

- Thread Theory
- Python Thread Library

title: Part 2

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,
```

title: Thread Order

Thread Communication

- Data protection
- Synchronization

title: Thread Communication

Data Protection

- Keeps shared memory safe
- Restricted code access

  Only one thread accesses block of code
- "critical section lock"

  aka "mutex", "atomic operation"
- Similar to DBMS locking

Synchronization

- Synchronize action between threads
- Passing data

  Threads wait for each other to finish tasks
- More efficient than polling

  aka "wait/notify", "rendezvous"

Thread Library

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

title: Thread Library

Lock()

- Basic building block

  Handles either protection or synchronization

- Methods

  `acquire(`*`blocking`*`)`

  `release()`

Critical Section Lock

```
    Thread 1                Thread 2
mutex.acquire()         ...
if myList:              ...
    work = myList.pop() ...
mutex.release()         ...
...                     mutex.acquire()
...                     if len(myList)<10:
...                         myList.append(work)
...                     mutex.release()
```

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()
...
```

title: Synching threads

RLock()

- Mutex only

  Other threads cannot release RLock()

- Recursive

- Methods

  ```
  acquire(blocking)
  release()
  ```

title: RLock()

Using RLock()

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

```
    Thread 1                    Thread 2
mutex.acquire()             ...
self.update()               ...
mutex.release()             ...
...                         mutex.acquire()
...                         self.update()
...                         mutex.release()
```

title: Using RLock()

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()

title: Condition()

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`*`)`

title: Event()

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

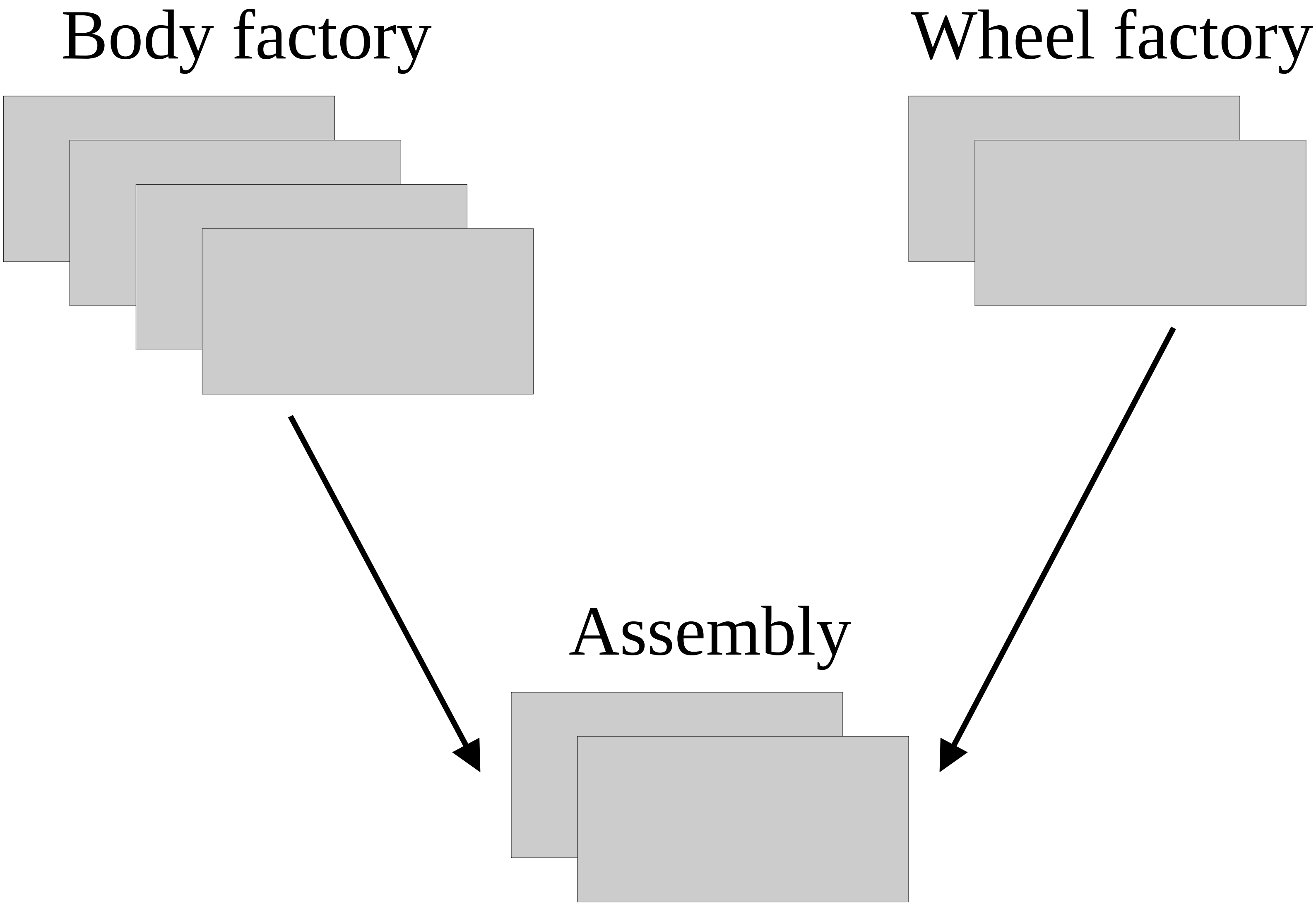
- Threads more like Perl

Producer/Consumer

- Example: factory

    One part of the factory *produces* part of a widget; another part of the factory *consumes* widget parts to make complete widgets. Trick is to keep it all in balance.

title:



Body factory

Wheel factory

Assembly

title: Factory 1

Factory Objects 1

Body

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

Wheels

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

Assembly

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

title: Factory Objects 1

Queue()

- Built on top of `thread`
  - Use with both `threading` and `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 non-blocking

title: Queue() Objects

Using Queue()

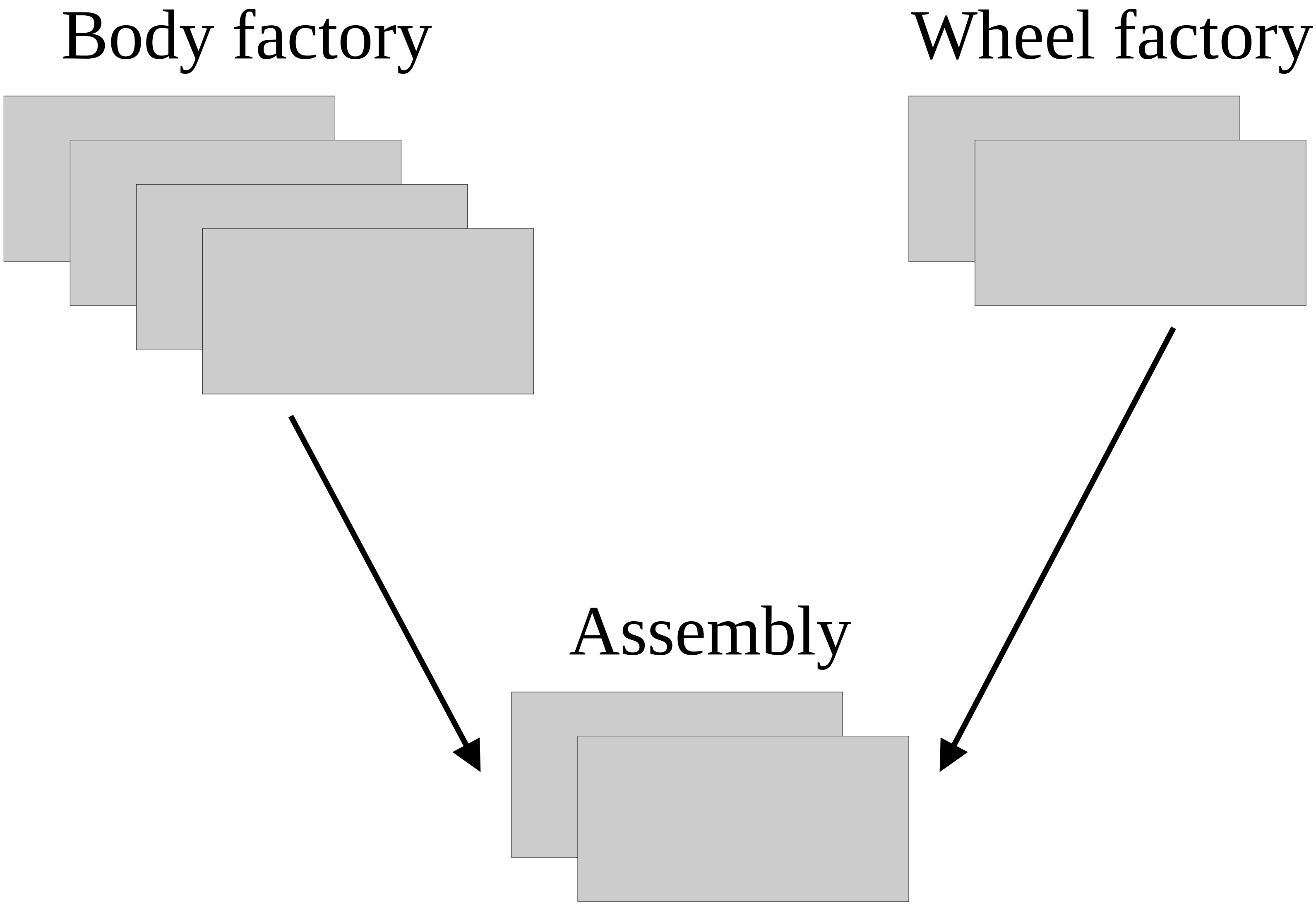
Thread 1

```
out = self.doWork()
queue2.put(output)
...
...
...
self.in = queue1.get()
```

Thread 2

```
...
...
self.in = queue2.get()
out = self.doWork()
queue1.put(output)
...
```

title: Using Queue()



Body factory

Wheel factory

Assembly

title: Factory 2

Factory Objects 2

Body

```
body.queue
```

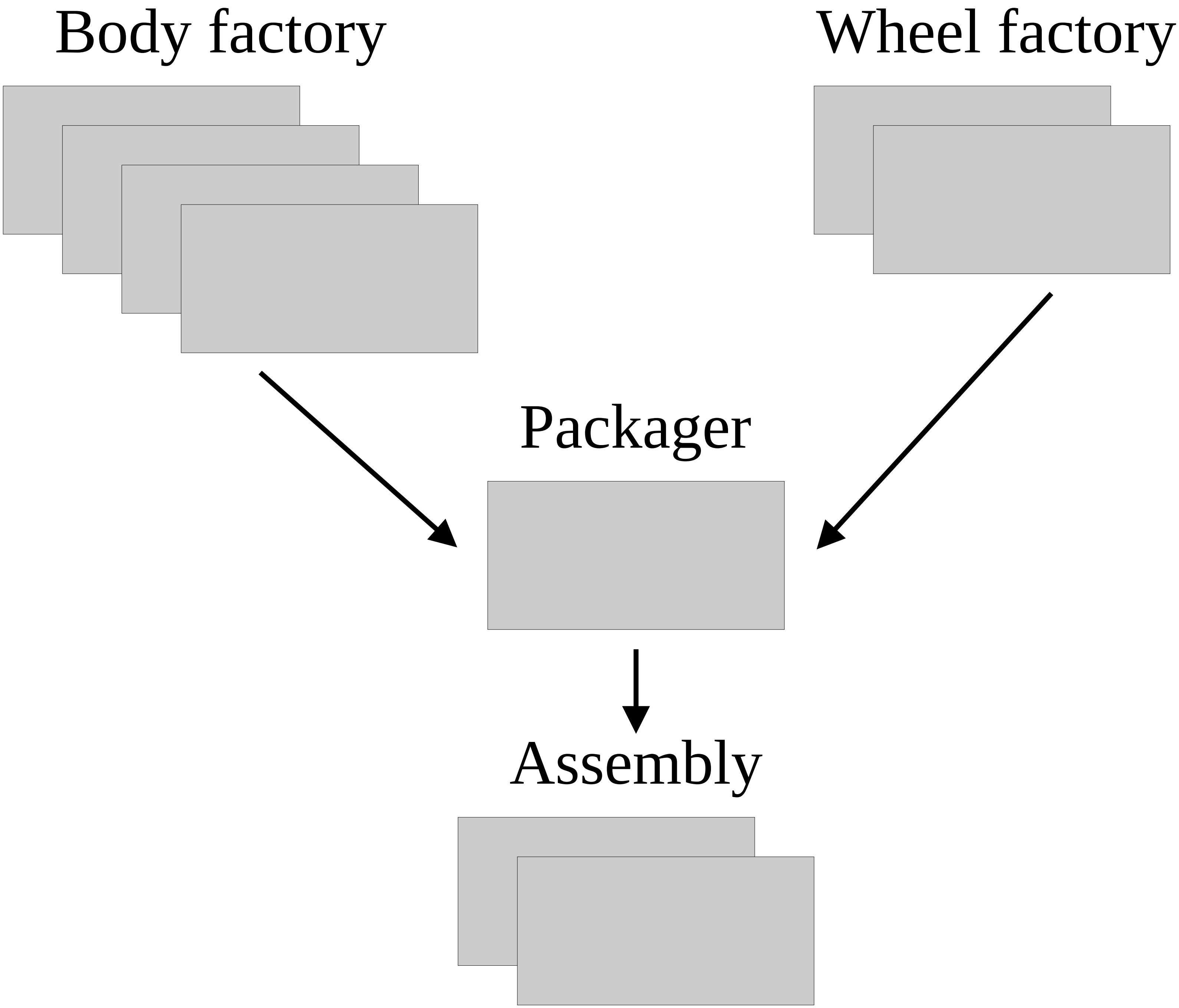
Wheels

```
wheels.queue
```

Assembly

```
body.queue
wheels.queue
assembly.rlock
```

title: Factory Objects 2



Body factory

Wheel factory

Packager

Assembly

title: Factory 3

Factory Objects 3

Body

```
body.queue
```

Wheels

```
wheels.queue
```

Packager

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

Assembly

```
assembly.queue
```

Recap Part 2

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

Part 3: Two Apps

- Using Queues
  - spider (thread pool)
  - GUI (Tkinter) (background thread)

title: Part 3

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

title: Widgets

Events

- Event loop
- Trigger callbacks

Tkinter resources

- Web

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

- Books

  Python and Tkinter Programming, John E. Grayson

title: Tkinter resources

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

FibThreaded Bugs and Exercises

- Fix deadlock on quit
- Fix display of illegal values
- Refactor for generic calc object

Compare Spider/Fib

- Shared structures vs. callbacks

Recap Part 3

title: Recap Part 3

Part 4: Miscellaneous

- Grab bag of useful info

title: Part 4

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)

Locks vs GIL

- Each lock is unique, a real OS-level lock; GIL is separate

GIL example

- Mutex only reading threads

Threads 1,4

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

Threads 2,3,5

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

- *Not* safe with UserList

title: GIL example

`dis` this

- `dis`assemble 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

Performance Tip

- `python -O`

    Also set `PYTHONOPTIMIZE`

    15% performance boost

    Removes bytecodes (`SET_LINENO`)

    Fewer context switches!

    Also removes `assert`

title: Performance Tip

`import` Editorial

- How to import

  ```
  from threading import Thread, Semaphore
  ```

  or

  ```
  import threading
  ```

- Don't use

  ```
  from threading import *
  ```

title: import Editorial

GIL and C Extensions

- Look for macros:

  `Py_BEGIN_ALLOW_THREADS`

  `Py_END_ALLOW_THREADS`

- Some common extensions:

  mxODBC - yes

  NumPy - no

- I/O exception: library problems

  e.g. `socket.gethostbyname()`

Stackless/Microthreads

- *Not* OS-level threads
- Mix: cooperative and preemptive
- Useful for thousands of threads

  e.g. simulations
- More info:

  ```
  http://www.tismer.com/research/stackless/
  http://world.std.com/~wware/uthread.html
  ```

Killing Threads

title: Killing Threads

Debugging Threads

- gdb

Thread Scheduling

- always on same cpu?
- specify CPU?

title: Thread Scheduling

Handling Exceptions

- `try`/`finally`

  Use to make sure locks get released

- `try`/`except`

  Close down all threads in outer block

  Be careful to pass `SystemExit` and `KeyboardInterrupt`

try/finally

title: try/finally

try/except

title: try/except

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?

title: Pop Quiz 1

Pop Quiz 2

What are "brute force" threads?

Explain what each of the following is used for:

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

Why are queues great?

Pop Quiz 3

How do you handle exceptions?

title: Pop Quiz 3