# Introduction to Threads

David Beazley

http://www.dabeaz.com

Note: This is a supplemental subject component to Dave's Python training classes. Details at:

http://www.dabeaz.com/python.html

Last Update : March 22, 2009

# Background

- Python is often used in applications where you want to the interpreter to be working on more than one task at once
- Example: An internet server handling hundreds of client connections

# Background

- There is also interest in making Python run faster with multiple CPUs

*"Can I make Python run 4 times faster on my quad-core desktop?"*

*"Can I make Python run 100 times faster on our mondo enterprise server?"*

- A delicate issue surrounded by tremendous peril

# Overview

- In this section, we'll look at some different aspects of Python thread programming
- This is mainly just an introduction
- The devil is in the details (left as an "exercise")

# Disclaimer

- Parallel programming is a huge topic
- This is not a tutorial on all of the possible ways you might go about doing it
- Really just a small taste of it

# Concept: Threads

- An independent task running inside a process
- Shares resources with the process (memory, files, network connections, etc.)
- Has own flow of execution (stack, PC)

# Thread Basics

```
% python program.py
        ↓
     statement
     statement
        ...
        ↓
```

"main thread"

Program launch. Python loads a program and starts executing statements

# Thread Basics

```
% python program.py
        ↓
    statement
    statement
       ...
        ↓
create thread(foo)  ·········>  def foo():
```

Creation of a thread.
Launches a function.

# Thread Basics

```
% python program.py
         ↓
     statement
     statement
        ...
         ↓
  create thread(foo)  ········→  def foo():
         ↓                            ↓
     statement                    statement
     statement                    statement
        ...                          ...
         ↓                            ↓
```

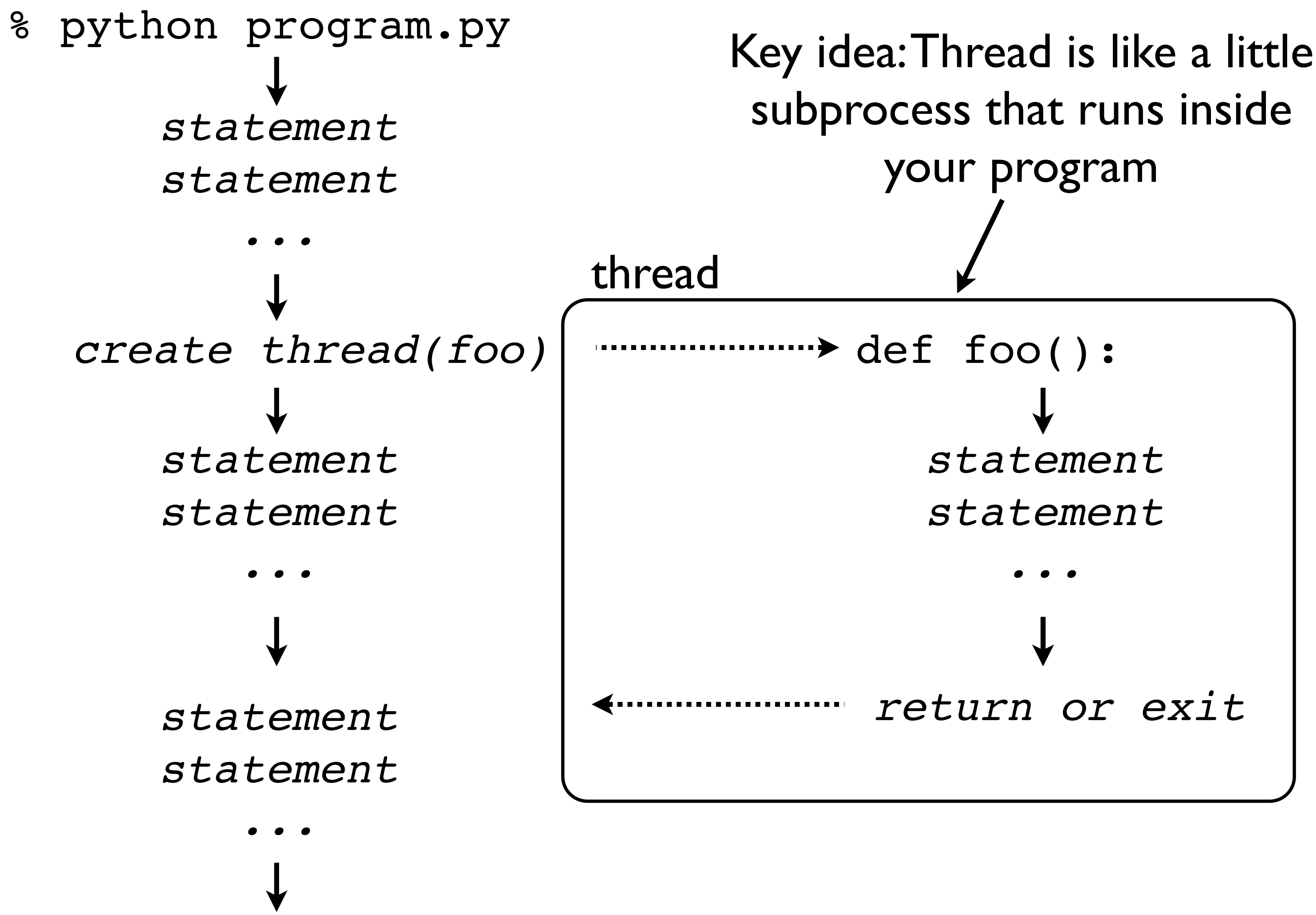
Parallel execution of statements

# Thread Basics

```
% python program.py
        ↓
     statement
     statement
        ...
        ↓
  create thread(foo)  ··········>  def foo():
        ↓                              ↓
     statement                      statement
     statement                      statement
        ...                            ...
        ↓                              ↓
     statement        <··········  return or exit
     statement
        ...
        ↓
```

thread terminates on return or exit

# Thread Basics



# threading module

- Threads are defined by a class

```
import time
import threading

class CountdownThread(threading.Thread):
    def __init__(self,count):
        threading.Thread.__init__(self)
        self.count = count
    def run(self):
        while self.count > 0:
            print "Counting down", self.count
            self.count -= 1
            time.sleep(5)
        return
```

- Inherit from Thread and redefine run()

# threading module

- To launch, create objects and use start()

```
t1 = CountdownThread(10)  # Create the thread object
t1.start()                # Launch the thread

t2 = CountdownThread(20)  # Create another thread
t2.start()                # Launch
```

- Threads execute until the run() method stops

# Functions as threads

- Alternative method of launching threads

```
def countdown(count):
    while count > 0:
        print "Counting down", count
        count -= 1
        time.sleep(5)

t1 = threading.Thread(target=countdown,args=(10,))
t1.start()
```

- Runs a function. Don't need to define a class

# Joining a Thread

- Once you start a thread, it runs independently
- Use t.join() to wait for a thread to exit

```
t.start()           # Launch a thread
...
# Do other work
...
# Wait for thread to finish
t.join()            # Waits for thread t to exit
```

- Only works from *other* threads
- A thread can't join itself

# Thread Methods

- How to check if a thread is still alive

```
if t.isAlive():
     # Still Alive
```

- Getting the thread name (a string)

```
name = t.getName()
```

- Changing the thread name

```
t.setName("threadname")
```

# Thread Execution

- Python stays alive until *all* threads exit
- This may or may not be what you want
- Common confusion: main thread exits, but Python keeps running (some other thread is still alive)

# Daemonic Threads

- Creating a daemon thread (detached thread)

```
t.setDaemon(True)
```

- Daemon threads run forever
- Can't be joined and is destroyed automatically when the interpreter exits
- Typically used to set up background tasks

# Thread Synchronization

- Different threads may share common data
- Extreme care is required
- One thread must not modify data while another thread is reading it
- Otherwise, will get a "race condition"

# Race Condition

- Consider a shared object

```
x = 0
```

- And two threads

```
Thread-1                    Thread-2
--------                    --------
...                         ...
x = x + 1                   x = x - 1
...                         ...
```

- Possible that the value will be corrupted
- If one thread modifies the value just after the other has read it.

# Race Condition

- The two threads

```
Thread-1                      Thread-2
--------                      --------
...                           ...
x = x + 1                     x = x - 1
...                           ...
```

- Low level interpreter execution

```
Thread-1                                  Thread-2
--------                                  --------

LOAD_GLOBAL  1 (x)
LOAD_CONST   2 (1)
                     thread switch -->
                                          LOAD_GLOBAL  1 (x)
                                          LOAD_CONST   2 (1)
                                          BINARY_SUB
                                          STORE_GLOBAL 1 (x)
BINARY_ADD           <-- thread switch
STORE_GLOBAL 1 (x)
```

# Race Condition

- Low level interpreter code

```
Thread-1                                Thread-2
--------                                --------

LOAD_GLOBAL  1 (x)
LOAD_CONST   2 (1)
                    thread switch
                                        LOAD_GLOBAL  1 (x)
                                        LOAD_CONST   2 (1)
                                        BINARY_SUB
                                        STORE_GLOBAL 1 (x)
                    thread switch
BINARY_ADD
STORE_GLOBAL 1 (x)
```

These operations get performed with a "stale" value of x. The computation in Thread-2 is lost.

# Race Condition

- Is this a real concern or some kind of theoretical computer science problem?

```
>>> x = 0
>>> def foo():
...     global x
...     for i in xrange(100000000): x += 1
...
>>> def bar():
...     global x
...     for i in xrange(100000000): x -= 1
...
>>> t1 = threading.Thread(target=foo)
>>> t2 = threading.Thread(target=bar)
>>> t1.start(); t2.start()
>>> t1.join(); t2.join()
>>> x
-834018
>>>
```

???

Yes, it's a real problem!

# Mutex Locks

- Mutual exclusion locks

```
m = threading.Lock()        # Create a lock
m.acquire()                 # Acquire the lock
m.release()                 # Release the lock
```

- Only one thread may hold the lock
- If another thread tries to acquire the lock, it blocks until the lock is released
- Use a lock to make sure only one thread updates shared data at once

# Use of Mutex Locks

- Commonly used to enclose critical sections

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

Thread-1                          Thread-2
--------                          --------
...                               ...
x_lock.acquire()                  x_lock.acquire()

x = x + 1                         x = x - 1

x_lock.release()                  x_lock.release()
...                               ...
```

Critical Section

- Only one thread can execute in critical section at a time (lock gives exclusive access)

# Other Locking Primitives

- Reentrant Mutex Lock

```
m = threading.RLock()       # Create a lock
m.acquire()                 # Acquire the lock
m.release()                 # Release the lock
```

- Can be acquired multiple times by same thread
- Semaphores

```
m = threading.Semaphore(n) # Create a semaphore
m.acquire()                 # Acquire the lock
m.release()                 # Release the lock
```

- Lock based on a counter
- Won't cover in detail here

# Events

- Use to communicate between threads

```
e = threading.Event()
e.isSet()          # Return True if event set
e.set()            # Set event
e.clear()          # Clear event
e.wait()           # Wait for event
```

- Common use

```
Thread 1                              Thread 2
--------                              --------
...                                   ...
# Wait for an event                   # Trigger an event
e.wait()  <------------------------   e.set()
                      notify
...
# Respond to event
```

# Thread Programming

- Programming with threads is hell
  - Complex algorithm design
  - Must identify all shared data structures
  - Add locks to critical sections
  - Cross fingers and pray that it works
- Typically you would spend several weeks of a graduate operating systems course covering all of the gory details of this

# Many Problems

- Excessive locking (poor performance)
- Deadlock
- Mismanagement of locks
- Debugging
- Frankly, it's almost never a good idea...

# Cost of Threads

- Threads sometimes considered for applications where there is massive concurrency (e.g., server with thousands of clients)
- However, threads are fairly expensive
- Often don't improve performance (extra thread-switching and locking)
- May incur considerable memory overhead (each thread has its own C stack, etc.)

# The Bad News

- Even if you can get your multithreaded program to work, it might not be faster
- In fact, it will probably run slower!
- The C Python interpreter itself is single-threaded and protected by a global interpreter lock (GIL)
- Python only utilizes one CPU--even on multi-CPU systems!

# Is There a Fix?

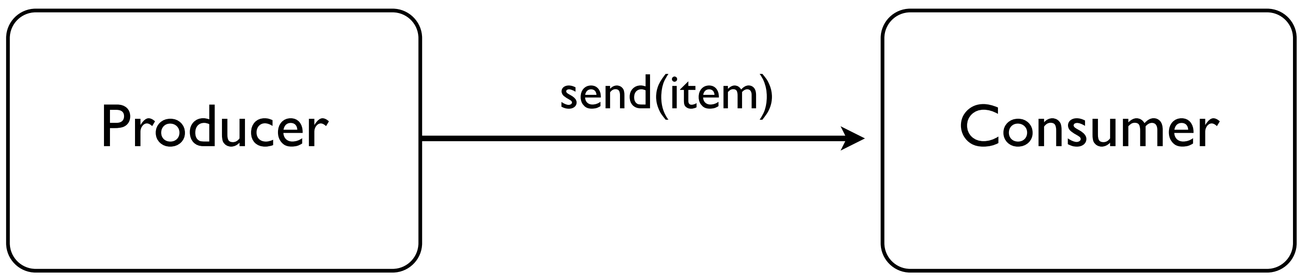
- No fix for the GIL is planned
- A big part of the problem concerns reference counting--which is an especially poor memory management strategy for multithreading
- May get true concurrency using Jython or IronPython which are built on JVM/.Net
- C/C++ extensions can also release the GIL

# A Thread Alternative

- Use message passing
- Multiple independent Python processes (possibly running on different machines) that perform their own processing, but which communicate by sending/receiving messages
- This approach is widely used in supercomputing for massive parallelization (1000s of processors)
- It can also work well for multiple CPU cores if you know what you're doing

# Threads and Messages

- If possible, try to organize multithreaded programs so that they are based on messaging




- Producer/consumer model.

# Consumers/Producers

- A thread should either be a producer or consumer of a data stream
- Producer : Produce a stream of data which other objects will receive
- Consumer : Consumes a sequence of data sent to it.

# Producer Thread

- Producers send data to subscribers...

```
class ProducerThread(threading.Thread):
    def __init__(self):
        threading.Thread.__init__(self)
        self.consumers = set()
    def register(self,cons):
        self.consumers.add(cons)
    def unregister(self,cons):
        self.consumers.remove(cons)
    def run(self):
        while True:
              ...
              # produce item
              for cons in self.consumers:
                  cons.send(item)
```

send data to consumers

# Consumers

- Always structure consumers as an object to which you send messages

```
class Consumer(object):
    # Send an item to the consumer
    def send(self,item):
        print "Got item"
        ...
    # No more items
    def close(self):
        print "I'm done."
```

- send() is what producers use to communicate with the consumer

# Consumer Example

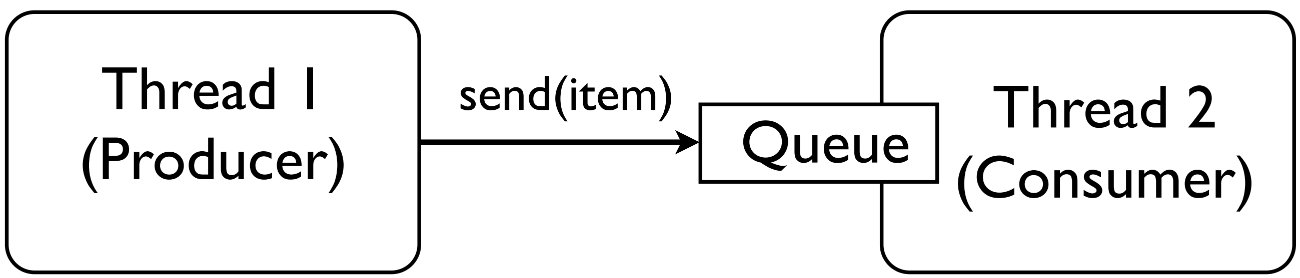
- Here is a simple example

```
class Countdown(object):
    def send(self,item):
        print "T-minus", item
    def close(self):
        print "Kaboom!"

>>> c = Countdown()
>>> c.send(10)
T-minus 10
>>> c.send(9)
T-minus 9
>>> c.close()
Kaboom!
>>>
```

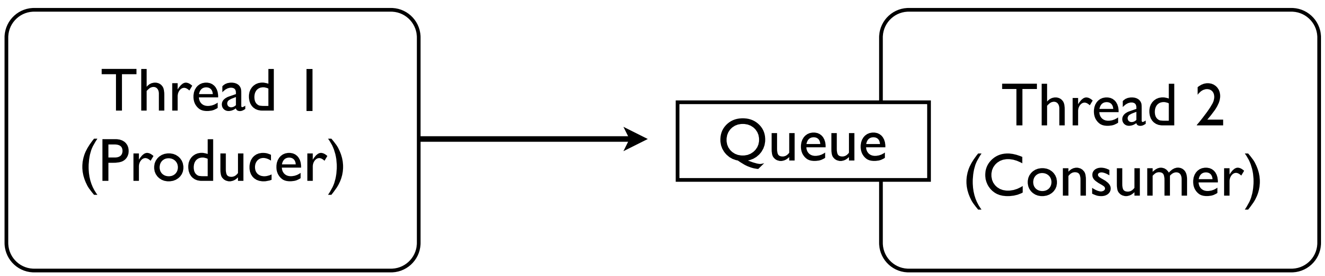
# Threads and Queues

- Producers and consumers can easily run in separate threads if you hook them together with a message queue



# Queue Module

- Provides a thread-safe queue object
- Designed for "Producer-Consumer" problems




- One thread produces data that is to be consumed by another thread

# Queue Module

- Creating a Queue

```
import Queue
q = Queue.Queue([maxsize])
```

- Putting items into a queue

```
q.put(item)
```

- Removing items from the queue

```
item = q.get()
```

- Both operations are thread-safe (no need for you to add locks)

# Consumer Thread

- Create a thread wrapper and use a Queue to receive and dispatch incoming messages

```
class ConsumerThread(threading.Thread):
    def __init__(self, consumer):
        threading.Thread.__init__(self)
        self.setDaemon(True)
        self.__consumer = consumer
        self.__in_q = Queue.Queue()
    def send(self,item):
        self.__in_q.put(item)
    def run(self):
        while True:
            item = self.__in_q.get()
            self.__consumer.send(item)
```

- Note: This wraps any non-threaded consumer

# Consumer Example

- Here is a simple example

```
class Countdown(object):
    def send(self,item):
        print "T-minus", item
    def close(self):
        print "Kaboom!"

>>> c = ConsumerThread(Countdown())
>>> c.start()
>>> c.send(10)
T-minus 10
>>> c.send(9)
T-minus 9
>>>
```

- Note: We're using our original non-threaded consumer as a target

# Consumer Shutdown

- Implementing close() on a thread

```
class ConsumerExit(object): pass         # A sentinel
class ConsumerThread(threading.Thread):
    ...
    def run(self):
        while True:
             item = self.__in_q.get()
             if item is ConsumerExit:
                  self.__consumer.close()
                  return
             else:
                  self.__consumer.send(item)
    def close(self):
        self.send(ConsumerExit)
```

- Note: ConsumerExit used as object that's placed on the queue to signal shutdown

# Coroutines

- The design of the consumer in the previous section was intentional
- Python has another programming language feature that is closely related to this style of programming
- Coroutines
- A form of cooperative multitasking

# Generators (Reprise)

- Recall that Python has generator functions

```
def countdown(n):
    print "Counting down"
    while n >= 0:
        yield n
        n -= 1
```

- This generates a sequence of values to be consumed by a for-loop

```
>>> c = countdown(5)
>>> for i in c:
...     print i,
Counting down
5 4 3 2 1
>>>
```

# Coroutines

- You can put yield in an expression instead

```
def countdown():
    print "Receiving countdown"
    while True:
        n = (yield)     # Receive a value
        print "T-minus", n
```

- This flips a generator around and makes it something that you send values to

```
>>> c = countdown()
>>> c.next()    # Alert! Advances to the first (yield)
>>> c.send(10)
T-minus 10
>>> c.send(9)
T-minus 9
>>>
```

# Control-flow

- send() sends a value into the (yield)
- The coroutine runs until it hits the next (yield) or it returns
- At that point, send() returns

```
...                              def coroutine():
statements                           ...
...                                  item = (yield)
c.send(item)                         ...
...                                  statements
statements                           ...
...                                  nextitem = (yield)
```

# Coroutine Setup

- One hacky bit...
- With a co-routine, you must always first call .next() to launch it properly
- This gets the co-routine to advance to the first (yield) expression

```
c = countdown()
c.next()
```

```
def countdown():
    print "Receiving countdown"
    while True:
        n = (yield)
        print "T-minus", n
```

- Now it's primed for receiving values...

# Coroutine Shutdown

- Co-routines can be shutdown with .close()
- Produces a GeneratorExit exception

```
def countdown():
    print "Receiving countdown"
    try:
        while True:
            n = (yield)     # Receive a value
            print "T-minus", n
    except GeneratorExit:
            print "Kaboom!"
```

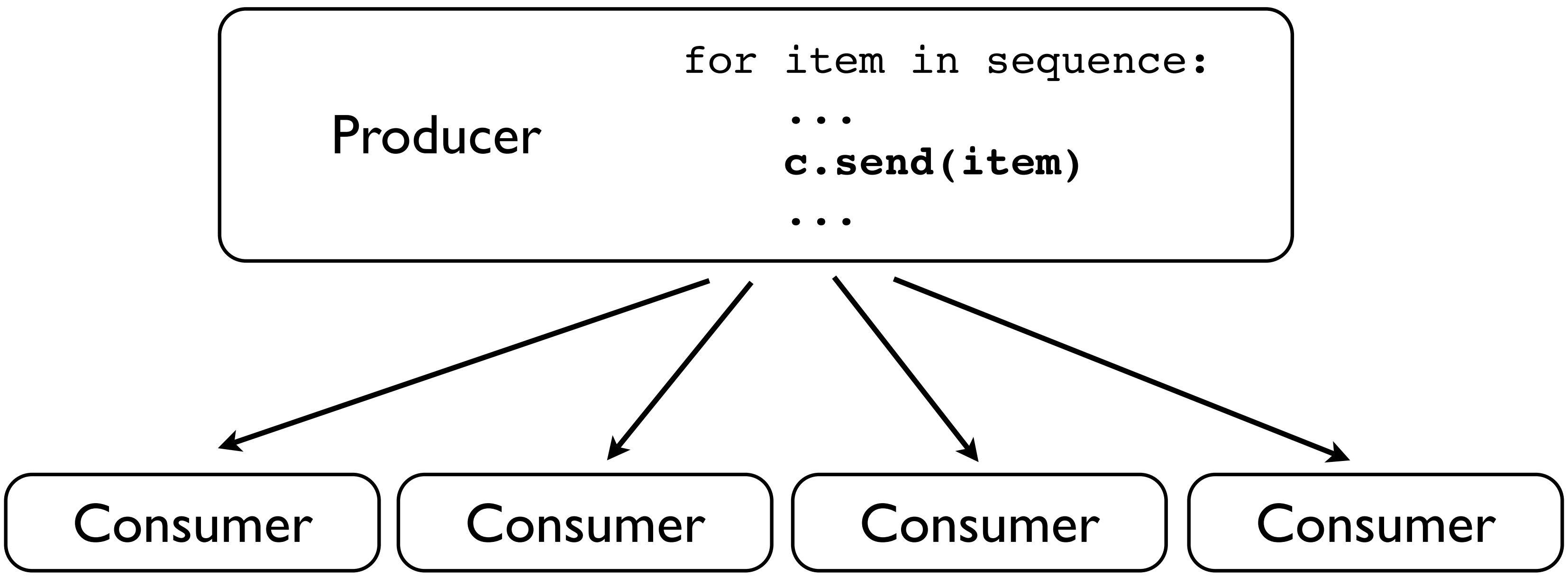
# Coroutine Shutdown

- Example

```
>>> c = countdown()
>>> c.next()    # Alert! Advances to the first (yield)
>>> c.send(10)
T-minus 10
>>> c.send(9)
T-minus 9
>>> c.close()
Kaboom!
>>>
```

# Dispatching

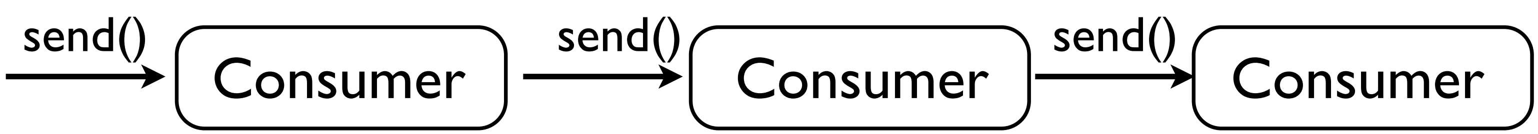
- Coroutines/threads often used to dispatch data to many consumers




- Consumers could be threads <u>or</u> coroutines

# Chaining

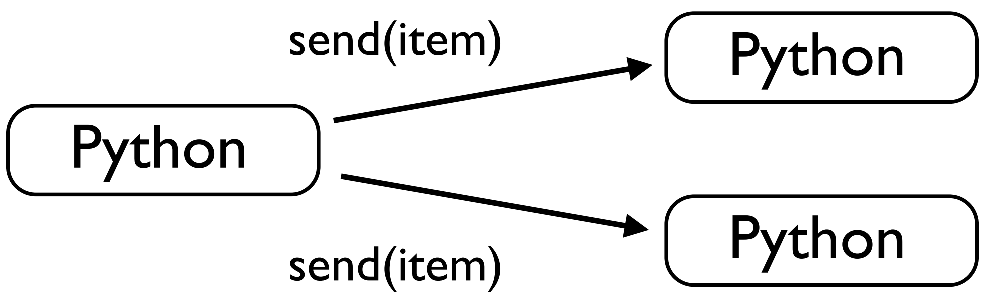
- Can chain consumers together as both consumers and producers of data




- Another way to set up processing pipelines

# Coprocesses

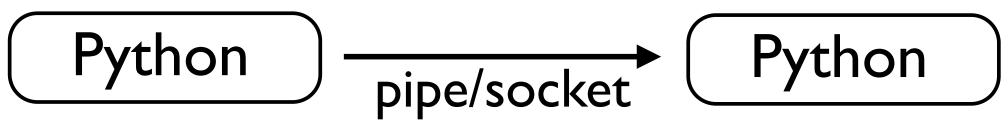
- Threads with message queues and coroutines lend themselves to one other concurrent programming technique
- Message-passing to coprocesses




- Independent Python processes (possibly running on different machines)

# Coprocesses

- Can set up a communication channel between two instances of the interpreter
- Use pipes, FIFOs, sockets, etc.



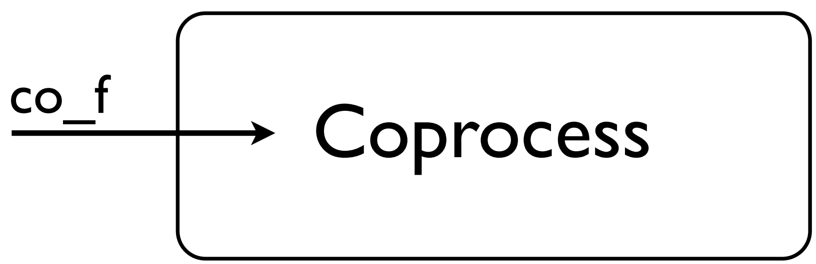

- At this time, there is no entirely "standard" interface for doing this, but you can roll your own if you have to

# Coprocess Object

- Create an object that wraps a file

```
class CoprocessBase(object):
    def __init__(self,co_f):
        self.co_f  = co_f
```

- This gives us an object with an input channel



# Coprocess Send

- Send an object to a coprocess

```
import cPickle as pickle
class CoprocessSender(CoprocessBase):
    def send(self,item):
        pickle.dump(item,self.co_f)
        self.co_f.flush()
    def close(self):
        self.co_f.close()
```

- Just use pickle to package up the payload.

# Coprocess Receiver

- Receive and dispatch items sent to a co-process

```
class Coprocess(CoprocessBase):
    def __init__(self,co_f,consumer):
        CoprocessBase.__init__(self)
        self.__consumer = consumer
    def run(self):
        while True:
            try:
               item = pickle.load(self.co_f)
               self.__consumer.send(item)
            except EOFError:
               self.__consumer.close()
```

- Again, this is a wrapper around a consumer

# Coprocess Example

- A simple example (assuming a pipe to stdin)

```
# countdown.py
import coprocess
import sys

class Countdown(object):
    def send(self,item):
        print "T-minus", item
    def close(self):
        print "Kaboom!"

c = coprocess.Coprocess(sys.stdin,Countdown())
c.run()
```

- Yes, this is the same consumer as before

# Coprocess Example

- Launching the coprocess

```
>>> import subprocess
>>> import coprocess
>>> p = subprocess.Popen(["python","countdown.py"],
...                      stdin=subprocess.PIPE)
>>> c = coprocess.CoprocessSender(p.stdin)
>>> c.send(5)
T-minus 5
>>> c.send(4)
T-minus 4
>>> c.close()
Kaboom!
>>>
```

- Note: coprocess output might show up elsewhere depending on the environment

# Commentary

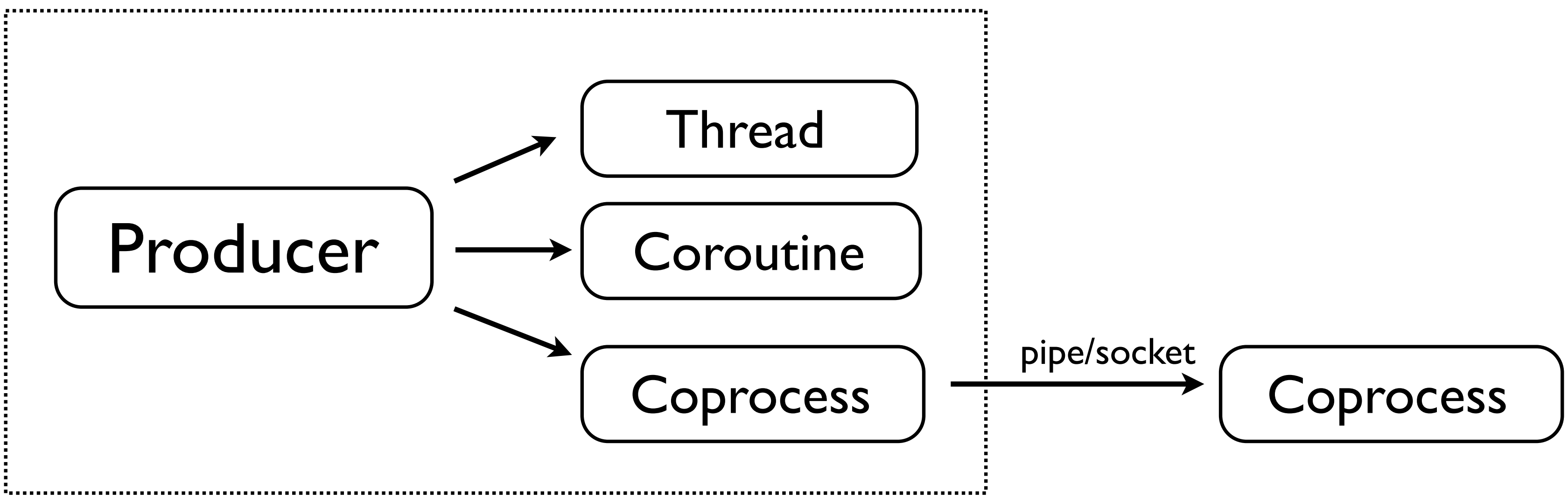
- This coprocess implementation <u>will work</u> across many different kinds of I/O channels
  - Pipes
  - FIFOs
  - Network sockets (s.makefile())
- This approach <u>will</u> result in concurrency across multiple CPUs (operating system can schedule independent processes on different processors)

# Limitations

- Security. Since we used pickle in the implementation, you would not use this where any end-point was untrusted
- Performance. Might want to use cPickle or a different messaging protocol.
- Two-way communication. No provision for the co-process to send data back to the sender. Possible, but very tricky.
- Debugging. Yow!

# Big Picture

- With care, the same consumer object can run as a thread, a coroutine, or a coprocess
- Various consumers all implement the same programming interface (send,close)



# Final Words

- Concurrent programming is not easy
- Personal preference : Use programming abstractions that are simple and easy to incorporate into different execution models
- Message-passing is one such example