Secrets of the Multiprocessing Module

DAVID BEAZLEY

David Beazley is an open source developer and author of the *Python Essential Reference* (4th Edition, Addison-Wesley, 2009). He is also known as the creator of Swig (http://www.swig.org) and Python Lex-Yacc (http://www.dabeaz.com/ply.html). Beazley is based in Chicago, where he also teaches a variety of Python courses.
dave@dabeaz.com

One of the most significant additions to Python's standard library in recent years is the inclusion of the multiprocessing library. First introduced in Python 2.6, multiprocessing is often pitched as an alternative to programming with threads. For example, you can launch separate Python interpreters in a subprocess, interact with them using pipes and queues, and write programs that work around issues such as Python's Global Interpreter Lock, which limits the execution of Python threads to a single CPU core.

Although multiprocessing has been around for many years, I needed some time to wrap my brain around how to use it effectively. Surprisingly, I have found my own use differs from those often provided in examples and tutorials. In fact, some of my favorite features of this library tend not to be covered at all.

In this column, I decided to dig into some lesser-known aspects of using the multiprocessing module.

Multiprocessing Basics

To introduce the multiprocessing library, briefly discussing thread programming in Python is helpful. Here is a sample of how to launch a simple thread using the threading library:

```
import time
import threading

def countdown(n):
   while n > 0:
      print "T-minus", n
      n -= 1
      time.sleep(5)
   print "Blastoff!"

t = threading.Thread(target=countdown, args=(10,))
t.start()
# Go do other processing
...
# Wait for the thread to exit
t.join()
```

Granted, this is not a particularly interesting thread example. Threads often want to do things, such as communicate with each other. For this, the Queue library provides a thread-safe queuing object that can be used to implement various forms of producer/consumer problems. For example, a more enterprise-ready countdown program might look like this:

```
import threading
import Queue
import time

def producer(n, q):
   while n > 0:
      q.put(n)
      time.sleep(5)
      n -= 1
   q.put(None)

def consumer(q):
   while True:
      # Get item
      item = q.get()
      if item is None:
         break
      print "T-minus", item
   print "Blastoff!"

if __name__ == '__main__':
   # Launch threads
   q = Queue.Queue()
   prod_thread = threading.Thread(target=producer, args=(10, q))
   prod_thread.start()

   cons_thread = threading.Thread(target=consumer, args=(q,))
   cons_thread.start()
   cons_thread.join()
```

But aren't I supposed to be discussing multiprocessing? Yes, but the above example serves as a basic introduction.

One of the core features of multiprocessing is that it clones the programming interface of threads. For instance, if you wanted to make the above program run with two separate Python processes instead of using threads, you would write code like this:

```
import multiprocessing
import time

def producer(n, q):
   while n > 0:
      q.put(n)
      time.sleep(5)
      n -= 1
   q.put(None)
```

```
def consumer(q):
    while True:
        # Get item
        item = q.get()
        if item is None:
            break
        print "T-minus", item
    print "Blastoff!"

if __name__ == '__main__':
    q = multiprocessing.Queue()
    prod_process = multiprocessing.Process(target=producer, args=(10, q))
    prod_process.start()

    cons_process = multiprocessing.Process(target=consumer, args=(q,))
    cons_process.start()
    cons_process.join()
```

A Process object represents a forked, independently running copy of the Python interpreter. If you view your system's process viewer while the above program is running, you'll see that three copies of Python are running. As for the shared queue, that's simply a layer over interprocess communication where data is serialized using the pickle library.

Although this example is simple, multiprocessing provides a whole assortment of other low-level primitives, such as pipes, locks, semaphores, events, condition variables, and so forth, all modeled after similar constructs in the threading library. Multiprocessing even provides some constructs for implementing shared-memory data structures.

No! No! No!

From the previous example, you might get the impression that multiprocessing is a drop-in replacement for thread programming. That is, you just replace all of your thread code with multiprocessing calls and magically your code is now running in multiple interpreters using multiple CPUs; this is a common fallacy. In fact, in all of the years I've used multiprocessing, I don't think I have ever used it in the manner I just presented.

The first problem is that one of the most common uses of threads is to write I/O handling code in servers. For example, here is a multithreaded TCP server using a thread-pool:

```
from socket import socket, AF_INET, SOCK_STREAM
from Queue import Queue
from threading import Thread

def echo_server(address, nworkers=16):
    sock = socket(AF_INET, SOCK_STREAM)
    sock.bind(address)
    sock.listen(5)

    # Launch workers
    q = Queue(nworkers)
```

```
    for n in range(nworkers):
        t = Thread(target=echo_client, args=(q,))
        t.daemon = True
        t.start()

    # Accept connections and feed to workers
    while True:
        client_sock, addr = sock.accept()
        print "Got connection from", addr
        q.put(client_sock)

def echo_client(work_q):
    while True:
        client_sock = work_q.get()
        while True:
            msg = client_sock.recv(8192)
            if not msg:
                break
            client_sock.sendall(msg)
        print "Connection closed"

if __name__ == '__main__':
    echo_server(("",15000))
```

If you try to change this code to use multiprocessing, the code doesn't work at all because it tries to serialize and pass an open socket across a queue. Because sockets can't be serialized, this effort fails, so the idea that multiprocessing is a drop-in replacement for threads just doesn't hold water.

The second problem with the multiprocessing example is that I don't want to write a lot of low-level code. In my experience, when you mess around with Process and Queue objects, you eventually make a badly implemented version of a process-worker pool, which is a feature that multiprocessing already provides.

MapReduce Parallel Processing with Pools

Instead of viewing multiprocessing as a replacement for threads, view it as a library for performing simple parallel computing, especially parallel computing that falls into the MapReduce style of processing.

Suppose you have a directory of gzip-compressed Apache Web server logs:

```
logs/
    20120701.log.gz
    20120702.log.gz
    20120703.log.gz
    20120704.log.gz
    20120705.log.gz
    20120706.log.gz
    ...
```

And each log file contains lines such as:

```
124.115.6.12 - - [10/Jul/2012:00:18:50 -0500] "GET /robots.txt HTTP/1.1" 200 71
```

```
210.212.209.67 - - [10/Jul/2012:00:18:51 -0500] "GET /ply/ HTTP/1.0" 200
11875
210.212.209.67 - - [10/Jul/2012:00:18:51 -0500] "GET /favicon.ico HTTP/1.0"
404 369
61.135.216.105 - - [10/Jul/2012:00:20:04 -0500] "GET /blog/atom.xml HTTP/1.1"
304 -
...
```

This simple script takes the data and identifies all hosts that have accessed the robots.txt file:

```
# findrobots.py

import gzip
import glob

def find_robots(filename):
    '''
    Find all of the hosts that access robots.txt in a single log file
    '''
    robots = set()
    with gzip.open(filename) as f:
        for line in f:
            fields = line.split()
            if fields[6] == '/robots.txt':
                robots.add(fields[0])
    return robots

def find_all_robots(logdir):
    '''
    Find all hosts across an entire sequence of files
    '''
    files = glob.glob(logdir+"/*.log.gz")
    all_robots = set()
    for robots in map(find_robots, files):
        all_robots.update(robots)
    return all_robots

if __name__ == '__main__':
    robots = find_all_robots("logs")
    for ipaddr in robots:
        print(ipaddr)
```

The above program is written in the style of MapReduce. The function find_robots() is mapped across a collection of filenames, and the results are combined into a single result—the all_robots set in the find_all_robots() function.

Suppose you want to modify this program to use multiple CPUs. To do so, simply replace the map() operation with a similar operation carried out on a process pool from the multiprocessing library. Here is a slightly modified version of the code:

```
# findrobots.py

import gzip
```

```
import glob
import multiprocessing

# Process pool (created below)
pool = None

def find_robots(filename):
    '''
    Find all of the hosts that access robots.txt in a single log file
    '''
    robots = set()
    with gzip.open(filename) as f:
        for line in f:
            fields = line.split()
            if fields[6] == '/robots.txt':
                robots.add(fields[0])
    return robots

def find_all_robots(logdir):
    '''
    Find all hosts across and entire sequence of files
    '''
    files = glob.glob(logdir+"/*.log.gz")
    all_robots = set()
    for robots in pool.map(find_robots, files):
        all_robots.update(robots)
    return all_robots

if __name__ == '__main__':
    pool = multiprocessing.Pool()
    robots = find_all_robots("logs")
    for ipaddr in robots:
        print(ipaddr)
```

If you make these changes, the script produces the same result, but runs about four times faster on my machine, which has four CPU cores. The actual performance will vary according to the number of CPUs available on your machine.

Using a Pool as a Thread Coprocessor

Another handy aspect of multiprocessing pools is their use when combined with thread programming. A well-known limitation of Python thread programming is that you can't take advantage of multiple CPUs because of the Global Interpreter Lock (GIL); however, you can often use a pool as a kind of coprocessor for computationally intensive tasks.

Consider this slight variation of our network server that does something a bit more useful than echoing data—in this case, computing Fibonacci numbers:

```
from socket import socket, AF_INET, SOCK_STREAM
from Queue import Queue
from threading import Thread
from multiprocessing import Pool
```

```
pool = None   # (Created below)

# A horribly inefficient implementation of Fibonacci numbers
def fib(n):
   if n < 3:
     return 1
   else:
     return fib(n-1) + fib(n-2)

def fib_client(work_q):
   while True:
      client_sock = work_q.get()
      while True:
         msg = client_sock.recv(32)
         if not msg:
            break
         # Run fib() in a separate process
         n = pool.apply(fib, (int(msg),))
         client_sock.sendall(str(n))
      print "Connection closed"

def fib_server(address, nworkers=16):
   sock = socket(AF_INET, SOCK_STREAM)
   sock.bind(address)
   sock.listen(5)

   # Launch workers
   q = Queue(nworkers)
   for n in range(nworkers):
      t = Thread(target=fib_client, args=(q,))
      t.daemon = True
      t.start()

   # Accept connections and feed to workers
   while True:
      client_sock, addr = sock.accept()
      print "Got connection from", addr
      q.put(client_sock)

if __name__ == '__main__':
   pool = Pool()
   fib_server(("",15000))
```

If you run this server, you'll find that it performs a neat little trick. For each client that needs to compute fib(n), the operation is handed off to a pool worker using pool.apply(). While the work takes place, the calling thread goes to sleep and waits for the result to come back. If multiple client threads make requests, the work is handed off to different workers and you'll find that your server is processing in parallel. Under heavy load, the server will take full advantage of every available CPU. The fabled GIL is not an issue here because all of the threads spend most of their time sleeping.

Note that this technique of using a pool as a coprocessor also works well in applications involving asynchronous I/O (i.e., code based on select-loops or event handlers), but because of space constraints, you'll just have to take my word for it.

Multiprocessing as a Messaging Library

Perhaps the most underrated feature of multiprocessing is its use as a messaging library from which you can build simple distributed systems. This functionality is almost never mentioned, but you can find it in the multiprocessing.connection submodule.

Setting up a connection between independent processes is easy. The following is an example of a simple echo-server:

```
# server.py
from multiprocessing.connection import Listener
from threading import Thread

def handle_client(c):
   while True:
      msg = c.recv()
      c.send(msg)

def echo_server(address, authkey):
   server_c = Listener(address, authkey=authkey)
   while True:
      client_c = server_c.accept()
      t = Thread(target=handle_client, args=(client_c,))
      t.daemon = True
      t.start()

if __name__ == '__main__':
    echo_server(("",16000), "peekaboo")
```

Here is an example of how you would connect to the server and send/receive messages:

```
>>> from multiprocessing.connection import Client
>>> c = Client(("localhost",16000), authkey="peekaboo")
>>> c.send("Hello")
>>> c.recv()
'Hello'
>>> c.send([1,2,3,4])
>>> c.recv()
[1, 2, 3, 4]
>>> c.send({'name':'Dave','email':'dave@dabeaz.com'})
>>> c.recv()
{'name': 'Dave', 'email': 'dave@dabeaz.com'}
>>>
```

As you can see, this is not just a simple echo-server as with sockets. You can actually send almost any Python object—including strings, lists, and dictionaries—back and forth between interpreters. Thus, this connection becomes an easy way to pass data structures around. In fact, any data compatible with the pickle module

should work. Further, there is even authentication of endpoints involving the authkey parameter, which is used to seed a cryptographic HMAC-based authentication scheme.

Although the messaging features of multiprocessing don't match those found in a library such as ZeroMQ (0MQ), you can use the messaging to perform much of the same functionality, if you're willing to write a bit of code. For example, here is a server that implements a Remote Procedure Call (RPC):

```
# rpcserver.py
from multiprocessing.connection import Listener, Client
from threading import Thread

class RPCServer(object):
   def __init__(self, address, authkey):
      self._functions = { }
      self._server_c = Listener(address, authkey=authkey)

   def register_function(self, func):
      self._functions[func.__name__] = func

   def serve_forever(self):
      while True:
         client_c = self._server_c.accept()
         t = Thread(target=self.handle_client, args=(client_c,))
         t.daemon = True
         t.start()

   def handle_client(self, client_c):
      while True:
         func_name, args, kwargs = client_c.recv()
         try:
            r = self._functions[func_name](*args,**kwargs)
            client_c.send(r)
         except Exception as e:
            client_c.send(e)

class RPCProxy(object):
   def __init__(self, address, authkey):
      self._conn = Client(address, authkey=authkey)
   def __getattr__(self, name):
      def do_rpc(*args, **kwargs):
         self._conn.send((name, args, kwargs))
         result = self._conn.recv()
         if isinstance(result, Exception):
            raise result
         return result
      return do_rpc

# Sample usage
if __name__ == '__main__':
   # Remote functions
```

```
def add(x,y):
   return x+y
def sub(x,y):
   return x-y

# Create and run the server
serv = RPCServer(("localhost",17000),authkey="peekaboo")
serv.register_function(add)
serv.register_function(sub)
serv.serve_forever()
```

To access this server as a client, in another Python invocation, you would simply do this:

```
>>> from rserver import RPCProxy
>>> c = RPCProxy(("localhost",17000), authkey="peekaboo")
>>> c.add(2,3)
5
>>> c.sub(2,3)
-1
>>> c.sub([1,2],4)
Traceback (most recent call last):
  File "", line 1, in
  File "rpcserver.py", line 37, in do_rpc
    raise result
TypeError: unsupported operand type(s) for -: 'list' and 'int'
>>>
```

Final Words

The multiprocessing module is a tool worth keeping in your back pocket. If you are performing MapReduce-style data analysis, you can use process pools for simple parallel computing. If you are writing programs with threads, pools can be used like a coprocessor to offload CPU-intensive tasks. Finally, the messaging features of multiprocessing can be used to pass data around between independent Python interpreters and build simple distributed systems.

References

[1] Multiprocessing official documentation: http://docs.python.org/library/multiprocessing.html.