Python Gets an Event Loop (Again)

DAVID BEAZLEY

David Beazley is an open source developer and author of the *Python Essential Reference* (4th Edition, Addison-Wesley, 2009) and *Python Cookbook* (3rd Edition, O'Reilly Media, 2013). 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

March 2014 saw the release of Python 3.4. One of its most notable additions is the inclusion of the new `asyncio` module to the standard library [1]. The `asyncio` module is the result of about 18 months of effort, largely spearheaded by the creator of Python, Guido van Rossum, who introduced it during his keynote talk at the PyCon 2013 conference. However, ideas concerning asynchronous I/O have been floating around the Python world for much longer than that. In this article, I'll give a bit of historical perspective as well as some examples of using the new library. Be aware that this topic is pretty bleeding edge—you'll probably need to do a bit more reading and research to fill in some of the details.

Some Basics: Networking and Threads

If you have ever needed to write a simple network server, Python has long provided modules for socket programming, processes, and threads. For example, if you wanted to write a simple TCP/IP echo server capable of handling multiple client connections, an easy way to do it is to write some code like this:

```
# echoserver.py

from socket import socket, AF_INET, SOCK_STREAM
import threading

def echo_client(sock):
  while True:
     data = sock.recv(8192)
     if not data:
        break
     sock.sendall(data)
  print('Client closed')
  sock.close()

def echo_server(address):
  sock = socket(AF_INET, SOCK_STREAM)
  sock.bind(address)
  sock.listen(5)
  while True:
     client_sock, addr = sock.accept()
     print('Connection from', addr)
     t = threading.Thread(target=echo_client,
                          args=(client_sock,))
     t.start()
```

```
if __name__ == '__main__':
    echo_server(('', 25000))
```

Although simple, this style of programming is the foundation for Python's `socketserver` module (called `SocketServer` in Python 2). `socketserver`, in turn, is the basis of Python's other built-in server libraries for HTTP, XML-RPC, and similar.

Asynchronous I/O

Even though programming with threads is a well-known and relatively simple approach, it is not always appropriate in all cases. For example, if a server needs to manage a very large number of open connections, running a program with 10,000 threads may not be practical or efficient. For such cases, an alternative solution involves creating an asynchronous or event-driven server built around low-level system calls such as `select()` or `poll()`. The underlying approach is based on an underlying event-loop that constantly polls all of the open sockets and triggers event-handlers (i.e., callbacks) on objects to respond as appropriate.

Python has long had a module, `asyncore`, for supporting asynchronous I/O. Here is an example of the same echo server implemented using it:

```
# echoasyncore.py
from socket import AF_INET, SOCK_STREAM
import asyncore

class EchoClient(asyncore.dispatcher):
    def __init__(self, sock):
        asyncore.dispatcher.__init__(self, sock)
        self._outbuffer = b''
        self._readable = True

    def readable(self):
        return self._readable

    def handle_read(self):
        data = self.recv(8192)
        self._outbuffer += data

    def handle_close(self):
        self._readable = False
        if not self._outbuffer:
            print('Client closed connection')
            self.close()

    def writable(self):
        return bool(self._outbuffer)

    def handle_write(self):
        nsent = self.send(self._outbuffer)
        self._outbuffer = self._outbuffer[nsent:]
        if not (self._outbuffer or self._readable):
            self.handle_close()

class EchoServer(asyncore.dispatcher):
    def __init__(self, address):
        asyncore.dispatcher.__init__(self)
        self.create_socket(AF_INET, SOCK_STREAM)
        self.bind(address)
        self.listen(5)

    def readable(self):
        return True

    def handle_accept(self):
        client, addr = self.accept()
        print('Connection from', addr)
        EchoClient(client)

EchoServer(('', 25000))
asyncore.loop()
```

In this code, the various objects `EchoServer` and `EchoClient` are really just wrappers around a traditional network socket. All of the important logic is found in callback methods such as `handle_accept()`, `handle_read()`, `handle_write()`, and so forth. Finally, instead of running a thread or process, the server runs a centralized event-loop initiated by the final call to `asyncore.loop()`.

Wilted Async?

Although `asyncore` has been part of the standard library since Python 1.5.2, it's always been a bit of an abandoned child. Programming with it directly is difficult—involving layers upon layers of callbacks. Moreover, the standard library doesn't provide any other support to make `asyncore` support higher-level protocols (e.g., HTTP) or to interoperate with other parts of Python (e.g., threads, queues, subprocesses, pipes, signals, etc.). Thus, if you've never actually encountered any code that uses `asyncore` in the wild, you're not alone. Almost nobody uses it—it's just too painful and low-level to be a practical solution for most programmers.

Instead, you'll more commonly find asynchronous I/O supported through third-party frameworks such as Twisted, Tornado, or Gevent. Each of these frameworks tends to be a large world unto itself. That is, they each provide their own event loop, and they provide asynchronous compatible versions of common library functions. Although it is possible to perform a certain amount of adaptation to make these different libraries work together, it's all a bit messy.

Enter asyncio

The `asyncio` library introduced in Python 3.4 represents a modern attempt to bring asynchronous I/O back into the standard library and to provide a common core upon which additional async-oriented libraries can be built. `asyncio` also aims to standardize the implementation of the event-loop so that it can be adapted to support existing frameworks such as Twisted or Tornado.

The definitive description of asyncio can be found in PEP-3156 [2]. Rather than rehash the contents of the admittedly dense PEP, I'll provide a simple example to show what it looks like to program with asyncio. Here is a new implementation of the echo server:

```
# echoasync.py

from socket import socket, AF_INET, SOCK_STREAM
import asyncio

loop = asyncio.get_event_loop()

@asyncio.coroutine
def echo_client(sock):
  while True:
     data = yield from loop.sock_recv(sock, 8192)
     if not data:
        break
     yield from loop.sock_sendall(sock, data)
  print('Client closed')
  sock.close()

@asyncio.coroutine
def echo_server(address):
  sock = socket(AF_INET, SOCK_STREAM)
  sock.bind(address)
  sock.listen(5)
  sock.setblocking(False)
  while True:
     client_sock, addr = yield from loop.sock_accept(sock)
     print('Connection from', addr)
     asyncio.async(echo_client(client_sock))

if __name__ == '__main__':
  loop.run_until_complete(echo_server(('', 25000)))
```

Carefully compare this code to the first example involving threads. You will find that the code is virtually identical except for the mysterious @asyncio.coroutine decorator and use of the yield from statements. As for those, what you're seeing is a programming style based on coroutines—in essence, a form of cooperative user-level concurrency.

A full discussion of coroutines is beyond the scope of this article; however, the general idea is that each coroutine represents a kind of user-level "task" that can be executed concurrently. The yield from statement indicates an operation that might potentially block or involve waiting. At these points, the coroutine can be suspended and then resumed at a later point by the underlying event loop. To be honest, it's all a bit magical under the covers. I previously presented a PyCon tutorial on coroutines [3]. However, the yield from statement is an even more modern development that is only available in Python 3.3 and newer, described in PEP-380 [4]. For now, just accept the fact that the yield from is required and that you've probably never seen it used in any previous Python code.

Getting Away from Low-Level Sockets

As shown, the sample echo server is directly manipulating a low-level socket. However, it's possible to write a server that abstracts the underlying protocol away. Here is a slightly modified example that uses a higher-level transport interface:

```
import asyncio
loop = asyncio.get_event_loop()

@asyncio.coroutine def
echo_client(reader, writer):
  while True:
     data = yield from reader.readline()
     if not data:
        break
     writer.write(data)
  print('Client closed')

if __name__ == '__main__':
  fut = asyncio.start_server(echo_client, '', 25000)
  loop.run_until_complete(fut)
  loop.run_forever()
```

As shown, this runs as a TCP/IP echo server. However, you can change it to a UNIX domain server if you simply change the last part as follows:

```
if __name__ == '__main__':
  fut = asyncio.start_unix_server(echo_client, '/tmp/spam')
  loop.run_until_complete(fut)
  loop.run_forever()
```

In both cases, the underlying protocol is abstracted away. The echo_client() function simply receives reader and writer objects on which to read and write data—it doesn't need to worry about the exact protocol being used to transport the bytes.

More Than Sockets

A notable feature of asyncio is that it's much more than a simple wrapper around sockets. For example, here's a modified client that feeds its data to a subprocess running the UNIX wc command and collects the output afterwards:

```
from asyncio import subprocess

@asyncio.coroutine
def echo_client(reader, writer):
  proc = yield from asyncio.create_subprocess_exec('wc',
                                stdin=subprocess.PIPE,
                               stdout=subprocess.PIPE)
  while True:
     data = yield from reader.readline()
```

```
            if not data:
                break
            proc.stdin.write(data)
            writer.write(data)
        proc.stdin.close()
        stats = yield from proc.stdout.read()
        yield from proc.wait()
        print("Client closed:", stats.decode('ascii'))
```

Here is an example of a task that simply sleeps and wakes up periodically on a timer:

```
@asyncio.coroutine
def counter():
    n = 0
    while True:
        print("Counting:", n)
        yield from asyncio.sleep(5)
        n += 1

if __name__ == '__main__':
    asyncio.async(counter())
    loop.run_forever()
```

There is even support for specialized tasks such as attaching a signal handler to the event loop. For example:

```
import signal

def handle_sigint():
    print('Quitting')
    loop.stop()

loop.add_signal_handler(signal.SIGINT, handle_sigint)
```

Last, but not least, you can delegate non-asynchronous work to threads or processes. For example, if you had a burning need for a task to print out Fibonacci numbers using a horribly inefficient implementation and you didn't want the computation to block the event loop, you could write code like this:

```
import asyncio
from concurrent.futures import ThreadPoolExecutor

loop = asyncio.get_event_loop()

def fib(n):
    if n <= 2:
        return 1
    else:
        return fib(n-1) + fib(n-2)

@asyncio.coroutine
def fibonacci():
    n = 1
    while True:
        r = yield from loop.run_in_executor(pool, fib, n)
        print("Fib(%d): %d" % (n, r))
        yield from asyncio.sleep(1)
        n += 1

if __name__ == '__main__':
    pool = ThreadPoolExecutor(8)
    asyncio.async(fibonacci())
    loop.run_forever()
```

In this example, the `loop.run_in_executor()` arranges to run a user-supplied function (`fib`) in a separate thread. The first argument supplies a thread-pool or process-pool as created by the `concurrent.futures` module.

Where to Go from Here?

There is much more to `asyncio` than presented here. However, I hope the few examples here have given you a small taste of what it looks like. For more information, you might consult the official documentation [1]; if you're like me, however, you'll find the documentation a bit dense and lacking in examples. Thus, you're probably going to have to fiddle around with it as an experiment. Searching the Web for "asyncio examples" can yield some additional information and insight for the brave. In the references section, I've listed a couple of presentations and sites that have more examples [5, 6].

As for the future, it will be interesting to see whether `asyncio` is adopted as a library for writing future asynchronous libraries and applications. As with most things Python 3, only time will tell.

If you're still using Python 2.7, the Trollius project [7] is a backport of the asyncio library to earlier versions of Python. The programming interface isn't entirely the same because of the lack of support for the "yield from" statement, but the overall architecture and usage are almost identical.

References

[1] asyncio (official documentation): http://docs.python.org/dev/library/asyncio.html.

[2] PEP 3156: http://python.org/dev/peps/pep-3156/.

[3] David Beazley, "A Curious Course on Coroutines and Concurrency": http://www.dabeaz.com/coroutines.

[4] PEP 380: http://python.org/dev/peps/pep-0380/.

[5] Saul Ibarra Corretge, "A Deep Dive Into PEP-3156 and the New asyncio Module": http://www.slideshare.net/saghul/asyncio.

[6] Feihong Hsu, "Asynchronous I/O in Python 3": http://www.slideshare.net/megafeihong/tulip-24190096.

[7] Trollius project: https://pypi.python.org/pypi/trollius.