Awaiting for Godot

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

Python 3.5 was released to the world on September 13, 2015. Included in this release was a substantial upgrade to asynchronous I/O support in the `asyncio` module, including brand new syntax in the Python language itself [1]. In this article, we dive into modern `asyncio` and take it for a test drive on code involving low-level socket programming, high-level sockets, HTTP clients, and HTTP servers. Prepare yourself for the unexpected—this is not the Python you know. Just to be clear, you'll need Python 3.5 or newer to try the examples.

But First, Some Beckett

To be honest, I've never considered myself to be much of a theater nut. In fact, my wife often teases me about how easily I fall asleep at shows. However, much to her chagrin, I find myself to be a huge fan of Samuel Beckett plays. For reasons that will become clear shortly, the title of this article is a reference to Beckett's most famous work, "Waiting for Godot," a play in which nothing actually seems to happen! I like such plays—they're easier to follow.

Beckett plays are not your ordinary kind of theatrical affair. For example, a few years back, I found myself riveted during a production of *Krapp's Last Tape*, a play where not a single word is spoken for the first 25 minutes. Instead the main character slowly paces around stage eating a banana—deep in thought. What is this? What is going on here? Or as a more extreme example, there is Beckett's *Play*—a production in which three motionless heads, situated atop funeral urns, talk frenetically amongst themselves at such a rapid pace, you can't make any sense of what is actually happening or what it is about. The play suddenly ends, the stage lights go out, and you're left wondering, "WHAT was THAT?" No, wait, the lights come back on and they simply repeat the whole play word-for-word start-to-finish again. What? As I said, Beckett plays defy convention.

This brings me to the topic of this article—asynchronous I/O and Python's new `asyncio` module. Like a Beckett play, `asyncio` defies normal Python conventions. At times, I don't know what exactly I'm looking at, and I'm not even sure if I like it. However, it's never boring. If anything, understanding `asyncio` will push your Python knowledge to the very edge of what's possible. First added in Python 3.4, `asyncio`'s goal was to reboot Python's support for asynchronous I/O and to take advantage of programming techniques involving advanced use of generator functions. However, in Python 3.5, it has taken flight with new language syntax and constructs. To the uninitiated, the code looks foreign and crazy. Is this even Python? What is this? What is going on?! As a whole, the Python community tends toward the conservative (just consider the number of people still using Python 2). Yet Python 3.5 might represent the most radical departure from the ordinary that I can recall in my nearly 20 years of Python coding. Yes, it's that different.

Although `asyncio` has been brewing in Python for a few years, I've never really quite figured out how to tackle it as a topic. Should I dive into its history and talk about the internals that make it work? Or should I just throw people into the deep end of the pool and see what happens? This article takes the latter approach. Assuming you have never done any prior async

programming in Python and know nothing about past efforts, what does it look like in Python 3.5? This is what we're going to find out. Part of the exercise is to see if using `asyncio` can be as easy and useful as more traditional parts of Python.

Some Background

Before beginning, it's useful to know why you might consider asynchronous I/O. In many modern network applications, it is common to have to a huge number of connected clients. These clients typically don't involve tons of high-bandwidth I/O—it's just that there are a lot of them (imagine a single server process with 10,000–20,000 open clients). Classic techniques for managing concurrency such as threads and processes have known limitations working with such a large number of clients. Asynchronous I/O takes a different approach. Typically, a single process runs an event-loop that polls for activity on all of the clients and handles it in some way, such as triggering event callback functions. However, callbacks have their own problem—leading to what's commonly referred to as a kind of "callback hell" of spaghetti code.

The `asyncio` module takes a different approach involving coroutines. Coroutines look a lot like normal synchronous code but are driven by an event loop under the covers. This tends to lead to code that is much easier to write and reason about. However, coroutines are not the same as ordinary functions or procedures. Thus, Python's `asyncio` module has a rather different flavor than the rest of the standard library.

To explore `asyncio`, we will look at four common problems involving network programming: programming directly with sockets, creating high-level socket servers, interacting with HTTP services, and creating a simple HTTP server. You'll probably want to hang on for the ride. Let's begin.

Low-Level Socket Programming

One of my first uses of Python involved writing simple network services directly using sockets [2]. Having previously written such code in C, it was an enlightening experience to see how easy it was in Python. For example, here is a simple multithreaded echo-server:

```
# A simple echo server using sockets and threads

from socket import *
from threading import Thread

def echo_server(address):
    sock = socket(AF_INET, SOCK_STREAM)
    sock.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
    sock.bind(address)
    sock.listen(5)
    while True:
        client, addr = sock.accept()
        print('Connection from:', addr)
        Thread(target=echo_client, args=(client,),
            daemon=True).start()

def echo_client(client):
    while True:
        data = client.recv(1000)
        if not data:
            break
        client.sendall(data)
    print('Connection closed')
    client.close()

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

It's simple, yet perhaps a bit dicey with the potential for unlimited thread creation as the number of clients increases.

As a first test of `asyncio`, let's see if we can write the same code without much fuss. Here is a direct translation of the code to an asynchronous version:

```
# A simple server using asyncio and sockets

from socket import *
import asyncio

async def echo_server(loop, address):
    sock = socket(AF_INET, SOCK_STREAM)
    sock.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
    sock.bind(address)
    sock.listen(5)
    sock.setblocking(False)   # Critical
    while True:
        client, addr = await loop.sock_accept(sock)
        print('Connection from:', addr)
        task = loop.create_task(echo_client(loop, client))

async def echo_client(loop, client):
    while True:
        data = await loop.sock_recv(client, 1000)
        if not data:
            break
        await loop.sock_sendall(client, data)
    print('Connection closed')
    client.close()

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

At first glance, this code will shatter your mind. `async def` and `await`? What is this business? It's new Python syntax related to asynchronous programming. The `async def` statement declares a function as a coroutine that must be managed by an event loop. The `await` statement is used to execute a coroutine and return its result. The two statements are meant to work together.

As noted, the execution of the code requires the use of an underlying event loop. The `asyncio.get_event_loop()` and `loop.run_until_complete()` calls at the end of the program show how you obtain a reference to the loop and initiate execution of the program.

Other operations in the code have changed into methods involving the event loop. For example, the `loop.sock_recv()`, `loop.sock_accept()`, and `loop.sock_sendall()` carry out the standard socket operations. The `loop.create_task()` method launches a new task much like the creation of a thread.

If you run the code, you'll find that it works just as it did before. Concurrent connections also work fine even though no threads or subprocesses are involved. Great!

High-Level Sockets

For most, directly programming with sockets is not the most ideal way to write network applications. Python has long provided a higher-level interface in the `socketserver` module (named `SocketServer` in legacy Python) [3]. Here is another implementation of an echo server using the streams interface of `socketserver`:

```
# Echo server using socketserver

from socketserver import (
    StreamRequestHandler,
    ThreadingTCPServer
)

class EchoHandler(StreamRequestHandler):
    def handle(self):
        print('Connection from:', self.client_address)
        for line in self.rfile:
            self.wfile.write(line)
        print('Connection closed')

if __name__ == '__main__':
    serv = ThreadingTCPServer(('', 25000), EchoHandler)
    serv.serve_forever()
```

In this code, the `EchoHandler` class has `rfile` and `wfile` attributes, which operate like files for reading and writing network data. A simple `for`-loop can be used to read line-by-line as shown. This loop will terminate when the connection is closed.

As a second test of `asyncio`, can we write a similar high-level program? Here is the answer:

```
# Echo server using asyncio and streams

import asyncio

async def handle_echo(reader, writer):
    print('Connection from:', writer.get_extra_info('peername'))
    async for line in reader:
        writer.write(line)
        await writer.drain()
    print('Connection closed')
    writer.close()

if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    coro = asyncio.start_server(handle_echo, '', 25000, loop=loop)
    server = loop.run_until_complete(coro)
    loop.run_forever()
```

Although the code is packaged in a slightly different way (no need for a class definition), it is comparably short and similar in logic. You'll find that it handles concurrent client connections even though no threads are being used.

Again, `async def` is being used to declare the handler as coroutine. The `await writer.drain()` statement waits for the written data to be successfully sent. A new mystery arises in the `async for` statement. What in the world is that? In short, it's a for-loop whose iteration requires coordination with the event loop (e.g., the event loop has to suspend the code and resume it when data actually arrives). If you don't use `async for`, the code would change slightly to the following:

```
async def handle_echo(reader, writer):
    print('Connection from:', writer.get_extra_info('peername'))
    while True:
        line = await reader.readline()
        if not line:
            break
        writer.write(line)
        await writer.drain()
    print('Connection closed')
    writer.close()
```

That's not nearly as elegant. Besides, dropping some Python code with an `async for` loop in it on your coworkers will be a good thing for them. Do it.

Making HTTP Requests

One of my favorite libraries for interacting with the Web is the `requests` library [4]. Using it is easy:

```
>>> import requests
>>> r = requests.get('http://www.python.org')
>>> print(r.text)
... see returned result ...
```

It's almost too easy. Of course, a really good request needs some threads. And threads need semaphores. And why not a queue for good measure? Behold:

```
# get.py
#
# Fetch data from a collection of URLs using requests and threads

import requests
from queue import Queue
from threading import Thread, Semaphore

max_active = Semaphore(4)

def url_worker(url, result_queue):
    with max_active:
        r = requests.get(url)
        result_queue.put((url, r.status_code, r.text))

def get_urls(urls):
    result_queue = Queue()

    # Launch workers
    workers = []
    for url in urls:
        worker = Thread(target=url_worker, args=(url,
             result_queue))
        worker.start()
        workers.append(worker)

    # Wait from the workers to finish and collect results
    results = { }
    for worker in workers:
        worker.join()
        url, status, text = result_queue.get()
        results[url] = (status, text)

    return results

if __name__ == '__main__':
    urls = [
        'https://docs.python.org/3/library/asyncio.html',
        'https://docs.python.org/3/library/select.html',
        'https://docs.python.org/3/library/threading.html',
        'https://docs.python.org/3/library/selectors.html',
        'https://docs.python.org/3/library/queue.html',
        'https://docs.python.org/3/library/socket.html',
        'https://docs.python.org/3/library/socketserver.html',
        ]
    result = get_urls(urls)
```

In this program, the `get_urls()` function takes a list of URLs and spawns a collection of worker threads to fetch data concurrently. The workers are throttled using a semaphore. Results are returned via a queue. The final result is a dict mapping URLs to a status code and text from the response. It's a program that only a parent process could love. Take that async!

A similar program can be written for asyncio using the third-party aiohttp library [5]. Using that, here's an async version:

```
# aget.py
#
# Fetch data from a collection of URLs using
# aiohttp and asyncio

import aiohttp
import asyncio

max_active = asyncio.Semaphore(4)

async def url_worker(url, result_queue):
    async with max_active:
        r = await aiohttp.get(url)
        await result_queue.put((url, r.status,
             await r.text()))

async def _get_urls_task(urls, loop):
    result_queue = asyncio.Queue()

    # Launch workers
    workers = []
    for url in urls:
        worker = loop.create_task(url_worker(url,
             result_queue))
        workers.append(worker)

    # Wait for the workers to finish
    results = { }
    for worker in workers:
        await asyncio.wait_for(worker, timeout=None)
        url, status, text = await result_queue.get()
        results[url] = (status, text)

    return results

def get_urls(urls):
    loop = asyncio.get_event_loop()
    result = loop.run_until_complete(_get_urls_task(urls, loop))
    return result
```

```
if __name__ == '__main__':
    urls = [
        'https://docs.python.org/3/library/asyncio.html',
        'https://docs.python.org/3/library/select.html',
        'https://docs.python.org/3/library/threading.html',
        'https://docs.python.org/3/library/selectors.html',
        'https://docs.python.org/3/library/queue.html',
        'https://docs.python.org/3/library/socket.html',
        'https://docs.python.org/3/library/socketserver.html',
        ]

    result = get_urls(urls)
```

Does it work? You bet, but there are a variety of details. First, the `aiohttp` module provides its own `get()` method for making an HTTP request. It's similar to the `requests` library except that you need to use `r = await aiohttp.get()` when making the request and `await r.text()` to obtain the downloaded data.

Next, the `asyncio` module provides its own version of synchronization primitives and queues. So the code uses `asyncio.Semaphore` and `asyncio.Queue`. The `async with max_active` statement in `url_worker` performs an asynchronous acquisition of the associated semaphore. All queueing operations are prefaced by the `await` keyword to indicate their asynchronous nature. The `asyncio.wait_for()` call waits for a task to finish. In many ways, it's not too dissimilar from threads. One subtle aspect concerns the separate `_get_urls_task()` and `get_urls()` functions. With `asyncio` nothing happens unless someone drives the underlying event loop. Thus, the `get_urls()` function works by setting up the calculation and driving the loop until the result comes back.

Stepping back for a moment, the only purpose of the queue in the threading example is to deal with the fact that there is no clean way to communicate results back from worker threads. It turns out you can eliminate it from the asynchronous code entirely. Here's a slightly modified version that is a bit cleaner:

```
import aiohttp
import asyncio

max_active = asyncio.Semaphore(4)

async def url_worker(url):
    async with max_active:
        r = await aiohttp.get(url)
        return url, r.status, await r.text()

async def _get_urls_task(urls, loop):
    # Launch workers
    workers = []
    for url in urls:
        worker = loop.create_task(url_worker(url))
        workers.append(worker)

    # Wait for the workers to finish
    results = { }
    for worker in workers:
        url, status, text = await asyncio.wait_for(worker,
                timeout=None)
        results[url] = (status, text)
        print(url, status)

    return results
```

`asyncio` provides a fairly complete set of primitives normally associated with thread programming. These include locks, semaphores, events, queues, and more. Although these objects do not provide 100% compatibility with their threading counterparts, it seems that many programs written for threads could probably be adapted to `asyncio` in a straightforward way.

Writing a Web Service

As a final test, let's write a simple Web service. In this example, the third-party `bottle` library is used to implement an endpoint that computes Fibonacci numbers, returning the result as JSON [6]:

```
# Simple web service using bottle

import bottle

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

@bottle.route('/fib/<n>;')
def serv_fib(n):
    result = fib(int(n))
    return { 'result': result }

if __name__ == '__main__':
    bottle.run(host='', port=25000, debug=True)
```

Connect to the service using a URL such as `http://localhost:25000/fib/6` on your machine. Replace `n` by an integer number such as 6. You should get a response such as this:

```
{"result": 8}
```

Can this code be replaced by an asynchronous version? `aiohttp` to the rescue!

```
# Simple web service using aiohttp

from aiohttp import web
import asyncio
import json
```

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

async def fib_handler(request):
    n = int(request.match_info['n'])
    result = fib(n)
    resp = { 'result': result }
    return web.Response(text=json.dumps(resp),
            content_type='application/json')

def run(address):
    loop = asyncio.get_event_loop()
    app = web.Application(loop=loop)
    app.router.add_route('GET', '/fib/{n}', fib_handler)
    srv = loop.create_server(app.make_handler(), *address)
    loop.run_until_complete(srv)
    loop.run_forever()

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

Alas, it's not quite as compact as the `bottle` version, but it performs the same function.

One downside of this implementation is that the implementation of `fib()` is especially bad for large integers. Try giving a number such as `45` as input and watch how the whole server locks up (you can't make requests from other clients). This is the nature of async—long-running calculations will block the entire event loop. However, you can move the work out to a thread pool by using the `loop.run_in_executor()` function as follows:

```
from concurrent.futures import ThreadPoolExecutor
_pool = ThreadPoolExecutor()

async def fib_handler(request):
    n = int(request.match_info['n'])
    if n < 15:
        result = fib(n)
    else:
        loop = asyncio.get_event_loop()
        result = await loop.run_in_executor(_pool, fib, n)
    resp = { 'result': result }
    return web.Response(text=json.dumps(resp),
            content_type='application/json')
```

If you try this modified version with a large integer, you'll find that the computation no longer blocks everything—you'll be able to make other requests while the calculation is churning away. Excellent. A similar technique would need to be used for any code that might potentially block the event loop.

Thoughts

In this article, I've presented a few examples of using `asyncio` with variations of the modern `async` syntax. At first glance, the code is probably a bit jarring—it looks unlike any Python code you might have written before. Yet, at the same time, the code retains the clarity of synchronous code written to use threads or processes. That is certainly a big plus.

A major concern with `asyncio` is that it tends to be an "all in" prospect for software development. If you're going to use it, you need to make sure that all of your libraries and code support it. Particular attention needs to be given to blocking operations involving files, network connections, subprocesses, and other I/O related tasks. A big part of the new "async" and "await" syntax is making the distinction between synchronous and asynchronous code absolutely clear. As a general rule, any existing Python code will probably run in an asynchronous environment, but unless "await" is being used, there's no guarantee that it won't block the event loop by accident. Whenever possible, you'll want to use libraries that have been written with `asyncio` in mind.

On that subject, just what libraries are available? At this time, there seems to be a growing set of modules for interacting with a variety of network services. As noted, `aiohttp` provides async support for HTTP, both as a client and a server. The library provides additional support for WebSockets and other modern HTTP features. A search for "asyncio" on the Python Package Index reveals a wide variety of libraries for interacting with services such as Redis, ZeroMQ, MondoDB, XML-RPC, IRC, Postgres, and others. Naturally, many of these are new and experimental. However, it seems that Python's asynchronous future is poised to be rather interesting.

Finally, a few words of caution. In writing this article, I debated how much information to provide on how `asyncio` actually works under the covers. If anything, you really don't want to know how it works—if you go wandering into the source code, your mind will completely shatter into a million pieces. Let's just say that it involves a lot of very clever programming with generators. I wrote about some of the general concepts in a previous *;login:* article although `asyncio` takes it to a whole new level of complexity [7]. Also, if you're going to use `asyncio` you must heed every word of advice you can find in the documentation. If you decide to play it fast and loose, you're likely going to spend hours debugging and cursing. For example, I wasted the better part of a half-day trying to figure out why I couldn't get my simple socket example to work. As it turns out I forgot to put the socket in non-blocking mode—a detail found in the documentation, but which I overlooked at the time. The bottom line: you need to read the documentation carefully.

More Information

The documentation for `asyncio` should be your first starting point. PEP 3156 is a must read for background information and rationale for `asyncio` [8]. PEP 492 contains information on the new `async` and `await` syntax [9]. There have been numerous talks given about `asyncio` over the last year. There are too many to list individually, but a search of pyvideo will not disappoint [10].

References

[1] asyncio module: https://docs.python.org/3/library/asyncio.html.

[2] socket module: https://docs.python.org/3/library/socket.html.

[3] socketserver module: https://docs.python.org/3/library/socketserver.html.

[4] requests module: http://www.python-requests.org/en/latest/.

[5] aiohttp module: http://aiohttp.readthedocs.org.

[6] bottle module: http://bottlepy.org.

[7] David Beazley, "A Tale of Two Concurrencies (Part 2)," *;login:*, vol. 40, no. 4, August 2015: https://www.usenix.org/publications/login/aug15/beazley.

[8] PEP 3156: https://www.python.org/dev/peps/pep-3156/.

[9] PEP 492: https://www.python.org/dev/peps/pep-0492/.

[10] pyvideo: http://pyvideo.org.