Crossing the Asynchronous Divide

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

The addition of improved support for asynchronous I/O in Python 3 is one of the most significant changes to the Python language since its inception. However, it also balkanizes the language and libraries into synchronous and asynchronous factions—neither of which particularly like to interact with the other. Needless to say, this presents an interesting challenge for developers writing a program involving I/O. In this article, I explore the problem of working in an environment of competing I/O models and whether or not they can be bridged in some way. As a warning, just about everything in this article is quite possibly a bad idea. Think of it as a thought experiment.

Pick a Color

I recently read an interesting blog post "What Color Is Your Function?" by Bob Nystrom [1]. I'm going to paraphrase briefly, but imagine a programming language where every function or method had to be assigned one of two colors, blue or red. Moreover, imagine that the functions were governed by some rules:

- The way in which you call a function differs according to its color.
- A red function can only be called by another red function.
- A blue function can never call a red function.
- A red function can call a blue function, but unknown bad things might happen.
- Calling a red function is much more difficult than calling a blue function.

Surely such an environment would lead to madness. What is the deal with those difficult red functions? In fact, after a bit of coding, you'd probably want to ditch all of the red code and its weird rules. Yes, you would, except for a few other details:

- Some library you're using has been written by someone who loves red functions.
- Red functions offer some advantages (i.e., concurrency, less memory required, more scalability, better performance, etc.).

Sigh. So, those red functions really are annoying. However, you're still going to have to deal with them and their weird rules in some manner.

Although this idea of coloring functions might seem like an invention of evil whimsy, it accurately reflects the emerging reality of asynchronous I/O in Python. Starting in Python 3.5, it is possible to define asynchronous functions using the `async` keyword [2]. For example:

```
async def greeting(name):
    print('Hello', name)
```

If you define such a function, it can be called from other asynchronous functions using the `await` statement.

```
async def spam():
    await greeting('Guido')
```

However, don't dare call an asynchronous function from normal Python code or from the interactive prompt—it doesn't work:

```
>>> await spam()
  File "<stdin>", line 1
    await spam()
          ^
SyntaxError: invalid syntax
>>>
```

You might think that you could make it do something if you take away the `await` statement. However, if you do that, you won't get an error—instead nothing happens at all.

```
>>> spam()
<coroutine object spam at 0x101a262b0>
>>>
```

To get an asynchronous function to run, you have to run it inside a separate execution context such as an event-loop from the `asyncio` library [3].

```
>>> import asyncio
>>> loop = asyncio.get_event_loop()
>>> loop.run_until_complete(spam())
Hello Guido
>>>
```

The `async` functions are clearly the red functions. They have weird rules and don't play nicely with other Python code. If you're going to use them, there will be consequences. Pick a side. You're either with us or against us.

An Example

To better illustrate the divide and to put a more practical face on the problem, suppose you were writing a network application that involved some code for sending JSON-encoded objects. Maybe your code involved a function such as this:

```
import json

def send_json(sock, obj):
    data = json.dumps(obj)
    sock.sendall(data.encode('utf-8'))
```

As written here, the function has been written in a synchronous manner. You could use it in a program that uses normal functions, threads, processes, and other Python features. For example, this function waits for a connection and sends back a JSON object in response:

```
def stest():
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.bind(('',25000))
    s.listen(1)
    c,a = s.accept()
    request = {
        'msg': 'Hello World',
        'data': 'x'
        }
    send_json(c, request)
    c.close()
    s.close()
```

To test this code, run `stest()` and connect using `nc` or `telnet`. You should see a JSON object sent back.

Now, suppose you wrote an asynchronous version of the `stest()` function:

```
import asyncio

async def atest(loop):
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.bind(('',25000))
    s.listen(1)
    s.setblocking(False)
    c,a = await loop.sock_accept(s)
    request = {
        'msg': 'Hello World',
        'data': 'x'
        }
    send_json(c, request)       # Dicey! Danger!
    c.close()
    s.close()

loop = asyncio.get_event_loop()
loop.run_until_complete(atest(loop))
```

In this code, you will notice that the `send_json()` function is being called directly. This is allowed by the rules (red functions can call blue functions). If you test the code, you'll find that it even appears to "work." Well, all except for the hidden time bomb lurking in the `send_json()` function.

Time bomb you say? Try changing the `request` to some large object like this:

```
request = {
    'msg': 'Hello World',
    'data': 'x'*10000000
}
```

Now, run the test again. You'll suddenly find that the program blows up in your face:

```
>>> loop.run_until_complete(atest(loop))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "python3.5/asyncio/base_events.py",
       line 342, in run_until_complete return future.result()
  File "python3.5/asyncio/futures.py", line
       274, in
       result raise self._exception
  File "python3.5/asyncio/tasks.py", line 239,
       in _step
       result = coro.send(value)
  File "j.py", line 32, in atest
    send_object(c, request)
  File "j.py", line 7, in send_object
    sock.sendall(data.encode('utf-8'))
BlockingIOError: [Errno 35] Resource temporarily unavailable
>>>
```

Silly you—that's what you get for filling up all of the I/O buffers in a function that was never safe to use in an asynchronous context. I hope you enjoy your 3:15 a.m. phone call about the whole compute cluster mysteriously going offline.

Of course, this problem can be fixed by writing a separate asynchronous implementation of the `send_json()` function. For example:

```
async def send_json(sock, obj):
    loop = asyncio.get_event_loop()
    data = json.dumps(obj)
    await loop.sock_sendall(sock, data.encode('utf-8'))
```

In your asynchronous code, you would then use this function by executing the following statement:

```
await send_json(c, request)
```

It's almost too simple—except for the fact that it's actually horrible.

Interlude: The Horror, the Horror

In the previous example, you can see how the code is forced to pick an I/O model. Interoperability between the two models isn't really possible. If you are writing a general-purpose library, you might consider supporting both I/O models by simply providing two different implementations of your code. However, that's also a pretty ugly situation to handle. Changes to one implementation would probably require changes to the other. Working with the two factions of your library is going to be a constant headache.

If you were working on a larger library or framework, you would likely find that your code base splits along the synchronous/asynchronous divide whenever I/O is involved. It would probably result in a gigantic mess. You might just abandon one of the sides altogether.

Even if you can manage to hold the whole ball of mud together in your mind, a library mixing synchronous and asynchronous code together is fraught with other problems. For example, users might forget to use the special `await` syntax in asynchronous calls. Synchronous calls executed from asynchronous functions may or may not work—with a variety of unpredictable consequences (e.g., blocking the event loop). Debugging would likely be fun.

Thought Experiment: Can You Know Your Color?

Needless to say, working in a mixed synchronous/asynchronous world has certain difficulties. However, what if functions could somehow determine the nature of the context in which they were called? Specifically, what if a function could somehow know whether it was called asynchronously?

As it turns out, this can be determined with a clever bit of devious frame hacking. Try defining this function:

```
import sys

def print_context():
    if sys._getframe(1).f_code.co_flags & 0x80:
        print('Asynchronous')
    else:
        print('Synchronous')
```

Just from the fact that the function uses a hardwired mysterious hex constant (0x80), you know that it's going to be good. Actually, you might want to ignore that part. However, try it out with this example:

```
>>> def foo():
...     print_context()
...
>>> foo()
Synchronous
>>> async def bar():
...     print_context()
...
>>> import asyncio
>>> loop = asyncio.get_event_loop()
>>> loop.run_until_complete(bar())
Asynchronous
>>>
```

This is interesting. The function `print_context()` is a normal Python function, yet it can determine the nature of the environment from which it was called. Naturally, this raises further questions about what might be possible with such information.

For example, could you use this in various metaprogramming features such as decorators? If so, maybe you can change some of the rules. Maybe you don't have to play by the rules.

Walled Gardens

Suppose you wanted to more strongly isolate the world of synchronous and asynchronous functions to prevent errors and undefined behavior. Here is a decorator that more strictly enforces the underlying I/O model on the calling context:

```
from functools import wraps
import sys
import inspect

def strictio(func):
   # Determine if func is an async coroutine
   is_async = inspect.iscoroutinefunction(func)
   @wraps(func)
   def wrapper(*args, **kwargs):
      called_async = sys._getframe(1).f_code.co_flags & 0x80
      if is_async:
         assert called_async, "Can't call async function here"
      else:
         assert not called_async, "Can't call sync function here"
      return func(*args, **kwargs)
   return wrapper
```

To use this decorator, simply apply it to either kind of function:

```
@strictio
def foo():
   print('Synchronous')

@strictio
async def bar():
   print('Asynchronous')
```

Attempts to call these functions from the wrong context now result in an immediate assertion error. For example:

```
>>> bar()
Traceback (most recent call last):
...
AssertionError: Can't call async function here
>>>

>>> async def test():
       foo()

>>> import asyncio
>>> loop = asyncio.get_event_loop()
>>> loop.run_until_complete(test())
Traceback (most recent call last):
...
AssertionError: can't call sync function here
>>>
```

As you can see, this is an even stronger version of the partisan rules—crossing the asynchronous divide is simply not allowed. If you applied this to the earlier `send_json()` function, you might have been able to prevent a hidden time bomb from showing up in your code. So that's probably a good thing.

Adaptive I/O

Rather than strictly separating the two worlds, another approach might be to adapt the execution of a function to the current I/O environment. For example, consider this decorator:

```
import sys
import inspect
import asyncio
from functools import partial

def adaptiveio(func):
   is_async = inspect.iscoroutinefunction(func)
   @wraps(func)
   def wrapper(*args, **kwargs):
      called_async = sys._getframe(1).f_code.co_flags & 0x80
      if is_async and not called_async:
         # Run an async function in a synchronous context
         loop = asyncio.new_event_loop()
         return loop.run_until_complete(func(*args,
             **kwargs))
      elif not is_async and called_async:
         # Run a sync function in an asynchronous context
         loop = asyncio.get_event_loop()
         return loop.run_in_executor(None, partial(func,
             *args, **kwargs))
      else:
         return func(*args, **kwargs)
   return wrapper
```

Unlike the previous example, this decorator adapts a function to the calling context if there is a mismatch. If called from an asynchronous context, a synchronous function is executed in a separate thread using `loop.run_in_executor()`. An asynchronous function called synchronously is executed using an event loop. Let's try it:

```
@adaptiveio
def foo():
   print('Synchronous')

@adaptiveio
async def bar():
   print('Asynchronous')
```

Now, make some calls:

```
>>> foo()
Synchronous
>>> bar()          # Adapted async -> sync
Asynchronous
>>> async def test():
      await foo()    # Adapted sync -> async
      await bar()

>>> import asyncio
>>> loop = asyncio.get_event_loop()
>>> loop.run_until_complete(test())
Synchronous
Asynchronous
>>>
```

A possible benefit of an adapted function is that a single implementation could be used seamlessly in either a synchronous or asynchronous context (the function would just "work" regardless of where you called it). A downside is that the mismatched use case might suffer a hidden performance penalty of some kind: for instance, the extra overhead of passing an operation over to a thread-pool or in creating the event loop. Perhaps the decorator could be extended to issue a warning message if this was a concern.

A subtle feature of this decorator is that an adapted function must use the normal calling convention of the current I/O context. So, if you had this function:

```
@adaptiveio
def send_json(sock, obj):
   data = json.dumps(obj)
   sock.sendall(data.encode('utf-8'))
```

you would use `send_json()` in synchronous code, and you would use `await send_json()` in asynchronous code.

Dual Implementation

Another possible strategy might be to bind separate synchronous and asynchronous functions to a common name. The following decorator allows an "awaitable" asynchronous implementation to be attached to an existing synchronous function.

```
import sys
import inspect
from functools import wraps

def awaitable(syncfunc):
   def decorate(asyncfunc):
      assert (inspect.iscoroutinefunction(asyncfunc) and
            not inspect.iscoroutinefunction(syncfunc))
      @wraps(asyncfunc)
      def wrapper(*args, **kwargs):
         called_async = sys._getframe(1).f_code.co_flags
               & 0x80
         if called_async:
            return asyncfunc(*args, **kwargs)
         else:
            return syncfunc(*args, **kwargs)
      return wrapper
   return decorate
```

With this decorator, you write two functions as before, but give them the same name. The appropriate function is used depending on the calling context. For example:

```
>>> def spam():
...     print('Synchronous')
...
>>> @awaitable(spam)
... async def spam():
...     print('Asynchronous')
...
>>> spam()
Synchronous
>>> async def test():
...     await spam()
...
>>> import asyncio
>>> loop = asyncio.get_event_loop()
>>> loop.run_until_complete(test())
Asynchronous
>>>
```

The main benefit of such an approach is that you could write code with a uniform API—the same function names would be used in either synchronous or asynchronous code. Of course, it doesn't solve the problem of having repetitive code. For example, the `send_json()` function would have two implementations like this:

```
def send_json(sock, obj):
   data = json.dumps(obj)
   sock.sendall(data.encode('utf-8'))

@awaitable(send_json)
async def send_json(sock, obj):
   loop = asyncio.get_event_loop()
   data = json.dumps(obj)
   await loop.sock_sendall(sock, data.encode('utf-8'))
```

Of course, all of this might just be a bad idea as heads explode while trying to figure out which function is being called during debugging. It's hard to say.

Thoughts

At this point, it's too early to tell how Python's emerging asynchronous support will play out except that libraries will likely have to side with one particular approach (asynchronous or synchronous). It seems possible that various metaprogramming techniques might be able to make the overall environment slightly more sane: for example, preventing common errors, adapting code, or making it easier to present a uniform programming interface. However, to my knowledge, this is not an approach being taken at this time.

Somewhere in the middle of this mess are libraries such as gevent [4]. gevent provides support for asynchronous programming but implements its concurrency at the level of the interpreter implementation itself (as a C extension). As a result, there is no obvious distinction between synchronous and asynchronous code—in fact, the same code can often run in both contexts. At this time, support for Python 3 in gevent is a bit new, and its whole approach runs in a different direction from the built-in `asyncio` library. Nevertheless, there's still a distinct possibility that this approach will prove to be the most sane in light of the difficulties associated with having code split into asynchronous and synchronous factions. Saying more about gevent, however, will need to be the topic of a future article.

In the meantime, if you're looking for some uncharted waters, you should definitely take a look at Python's emerging asynchronous I/O support. May your code be interesting.

References

[1] What Color Is Your Function: http://journal.stuffwithstuff.com/2015/02/01/what-color-is-your-function.

[2] PEP 0492 - Coroutines with async and await syntax: https://www.python.org/dev/peps/pep-0492.

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

[4] gevent: http://www.gevent.org.