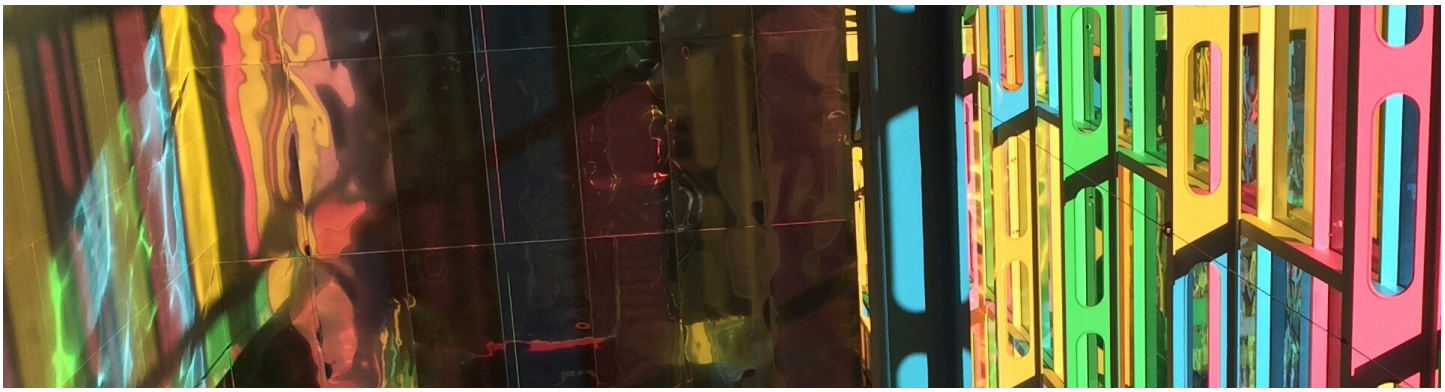

# The Function Colour Myth

## Or: async/await is not what you think it is.

Make no mistake: we are living through a new asynchronous programming renaissance. The programming community has spent the 2010s rediscovering the techniques that we used to use to handle highly-concurrent workloads when we didn't have the memory available to spawn an operating system thread to handle every single connection. This trend is incredibly apparent when we look at the languages and language features that are heavily discussed and debated. Consider the meteoric rise of Node.JS (the ultimate async-first language), or the incredible prominence of the goroutine and green threading primitives in Go, or the incredible number of languages that are hopping on the `async` / `await` train that C♯ got started.

While there are many interesting things to talk about in that prior list of languages and features[1], I really want to focus on the last one: `async` / `await`. In particular, I want to talk about a fairly popular blog post that takes aim at the `async` / `await` syntactic feature: What Color is Your Function?. For those who are new to the topic, I'll very briefly explain what `async` / `await` *are*. However, I will not summarise the prior blog post: it's a good piece of work and should be read in its entirety. If you're totally unfamiliar with this discussion, read my little `async` / `await` summary, then read the blog post, then come back here. I promise the blog post won't go away while you're gone.

One last note: I am explicitly discussing only *concurrent* programming here, not *parallel* programming. If you're not comfortable with this distinction, go watch Rob Pike explain the difference and come back. Fundamentally, this discussion is related to parallelism but not really affected by it in any profound way.

## An Async/Await Primer

What is `async` / `await` at a high level?

Basically, they're tools for explicitly marking concurrent control flow using language keywords. If you have an operation that is going to need to wait for some I/O or for some background process to complete, you can wrap that in a function (call it "function A") and mark that function as `async`. Then, you can call that function from some other function ("function B") using the keyword `await`, which is a signal to the language runtime that it should pause execution of function B until function A's I/O has completed.

Put another way, `async` / `await` are keywords that signal to the language that your code is willing to pause its execution at this point to let other code run. It's cooperative multitasking!

Under the hood of different languages `async` / `await` do different things, but the net effect is always that they allow you to write concurrent code in a somewhat linear manner. In effectively-single-threaded languages like Python, they let you turn a series of callbacks written as individual functions into a series of what appear to be mostly ordinary functions but magically can yield their flow of control to someone else. This is a neat trick, and it's extremely helpful from the perspective of clearing up some of the downsides of traditional concurrent code: specifically, callback hell.

That's all the time I'm willing to spend on this primer here. If you want more detail on how they work in Python (and I recommend you get it, this is a cool idea well worth understanding), Brett Cannon has a great blog post that covers this.

# Get To The Point Already

So, lets talk about function colour. Bob Nystrom's great blog post about function colour that I linked above uses the idea of coloured functions to talk about the fact that the `async` / `await` syntactic feature effectively means that a programming language has two types (or "colours") of function. One type is what you'd think of as your standard function: for the rest of this post I'll call that a "synchronous" function. The other type is a function that is annotated as `async`: what we'll call (inventively) "asynchronous" functions.

Bob points out that there are rules for these kinds of function. Specifically, while synchronous functions can be called from any kind of function, *asynchronous* functions must be called only from other asynchronous functions by means of the `await` keyword. Put in Python-specific terms, consider the following functions:

```
def function_a(some_socket_or_something):
    return some_socket_or_something.recv(8192)

async def function_b(some_streamreader_or_something):
    return (await some_streamreader_or_something.read(8192))
```

If you want to call `function_a`, you can call that from anywhere, even inside `function_b`. However, you cannot call `function_b` from within `function_a`. In Python, if you try to do this, you'll find that `function_b` never actually executes: it returns immediately and does nothing. In more statically-typed languages like C♯, you'll get a compiler error instead. Python's behaviour here (subtly failing to execute your code) is actually one of the most common bugs hit by people using Python async functions: forgetting to annotate them with `await`.

From Bob's perspective, this dichotomy makes your life sufficiently hard that it is more-or-less unbearable (specifically, Bob will "start grinding [his] teeth" when he hears about this kind of thing). That's fair enough: people like what they like, and there is nothing wrong with saying that you'd rather approach this a different way. People should write code in whatever way makes them most productive, and if that means you'd like to use thousands of threads and write in Visual Basic 6 then hell, you go Glen Coco. Godspeed.

However, recently we have seen discussions in the Python community around the pervasiveness of `async` / `await` that have rather misunderstood *why* `async` / `await` is worth having. In particular, from time to time you'll catch someone saying that *all* non-threaded concurrent code (that is, all code that uses some variant of the Reactor pattern) is necessarily coloured code, and that `async` / `await` exist to simply make those colours obvious.

I think that fundamentally this idea, which is parallel to Bob's, is a severe misunderstanding of the way async code works. In the rest of this post I'd like to outline what I mean.

# Colour Isn't Mandatory

So let's get something out of the way right now: function colour is a *tool* for writing async code, not a requirement of it.

This is extremely obviously true. For example, in a prior job I worked on a gigantic C codebase that was written entirely in a cooperative multitasking style. That is, it was very similar to Node.JS, but written entirely in C. Now, C does not have any method of describing

function colour, nor does it have any mechanism of giving a function a colour. In C, your code executes synchronously, step by step, forever. However, we were perfectly capable of writing a real, serious application, entirely in C, that didn't use a callstack-based resumption model like the one Bob discusses. So it's clear that function colour isn't *intrinsic* to this problem.

Moreover, it's not even intrinsic to Python. I know that Twisted is (cruelly and undeservedly) un-loved in the wider Python community, but consider a very basic Twisted program.

```
from twisted.internet.protocol import Protocol

class PrinterProtocol(Protocol):
    def connectionMade(self, transport):
        self.transport = transport
        self.transport.write(
            b'GET / HTTP/1.1\r\n'
            b'Host: http2bin.org\r\n'
            b'Connection: close\r\n'
            b'\r\n'
        )

    def dataReceived(self, data):
        print data,

    def connectionLost(self, reason):
        print "Connection lost"
```

This is a basic Twisted Protocol, which is responsible for managing a network connection. This is a core demonstration of how to write concurrent code: all of these functions are intended to be callbacks.

My question is: where is the function colour here? None of these functions are `async` functions, so that's not it. Are they implicitly coloured? Well, if they were that could only be because they'd be unable to be called from synchronous code[2]. But that's not true at all. For example, you can use this `Protocol` entirely from synchronous code:

```
import socket

def main():
    p = PrinterProtocol()
    s = socket.create_connection(('http2bin.org', 80))
    s = s.makefile()
    p.connectionMade(s)
```

```
    while True:
        data = s.read(8192)
        if data:
            p.dataReceived(data)
        else:
            p.connectionLost(None)
            break

    s.close()
```

There's no async here at all! Look at all these blocking socket calls. We spend so much time waiting for the kernel to do stuff it's *crazy*. All of these functions are synchronous functions in the grand tradition of synchronous functions. There is no requirement for function colour at all here.

What about Futures (or, as Twisted calls them, Deferreds)? Bob says this in his post:

> You still can't call a function that returns a future from synchronous code. (Well, you can, but if you do, the person who later maintains your code will invent a time machine, travel back in time to the moment that you did this and stab you in the face with a #2 pencil.)

Well, that statement is true-*ish*. It's certainly true of `asyncio.Future`, because `asyncio.Future` is a Python *awaitable*, and also requires a running event loop to execute any code. However, it's *not* true of Twisted's `Deferred`. Twisted's `Deferred`, unlike asyncio's `Future`, is not an event loop construct: it does not require an event loop to execute[3]. When you call `callback` on a `Deferred`, Twisted will *immediately* execute any attached callbacks *synchronously*, in-line with the function that just fired the `Deferred`. This means that `Deferred`s are *also* not coloured objects, and you can do things like refactor the code above to this:

```
from twisted.internet.defer import Deferred
from twisted.internet.protocol import Protocol

def connectionLossCallback(connection):
    print "Connection lost: %s" % connection

class PrinterProtocol(Protocol):
    def connectionMade(self, transport):
        self.transport = transport
        self.connectionLostDeferred = Deferred()
        self.transport.write(
```

```
            b'GET / HTTP/1.1\r\n'
            b'Host: http2bin.org\r\n'
            b'Connection: close\r\n'
            b'\r\n'
        )

    def dataReceived(self, data):
        print data,

    def connectionLost(self, reason):
        self.connectionLostDeferred.callback(self)
```

This code *also* runs perfectly happily using the skeleton code I outlined above. Still no function colours here: everything is just a synchronous function call.

Hopefully, this will convince you that there is nothing *mandatory* about function colours when it comes to writing asynchronous code. You don't have to use them. You can always construct your world out of a collection of synchronous functions that expect to run as callbacks. The only moment that function colour shows up is the moment you start making blocking I/O calls, and that there is where this discussion gets interesting.

# Why We Use Function Colour

So hopefully I've just made it clear to you that you can write asynchronous code using entirely synchronous function calls: there is no requirement for coloured functions. So why does function colour come up at all?

A hint at the answer can be found in Glyph's 2014 blog post Unyielding. This blog post is nominally about green threading vs regular threading, but along the way it touches on the four modes of concurrent programming:

> 1. Straight callbacks: Twisted's `IProtocol`, JavaScript's `on<foo>` idiom, where you give a callback to something which will call it later and then return control to something (usually a main loop) which will execute those callbacks,
> 2. "Managed" callbacks, or Futures: Twisted's `Deferred`, JavaScript's `Promises/A[+]`, E's Promises, where you create a dedicated result-that-will-be-available-in-the-future object and return it for the caller to add callbacks to,
> 3. Explicit coroutines: Twisted's `@inlineCallbacks`, Tulip's `yield from` coroutines, C#'s `async/await`, where you have a syntactic feature that explicitly suspends the current routine,

> 4. and finally, implicit coroutines: Java's "green threads", Twisted's Corotwine, eventlet, gevent, where any function may switch the entire stack of the current thread of control by calling a function which suspends it.

If we reframe option 4 to just say "threads", green or otherwise, then we can write our own list:

1. Straight callbacks: uncoloured.
2. Managed callbacks: usually uncoloured, but they may be coloured in some cases.
3. Explicit coroutines: usually coloured, but they may be uncoloured in some cases (e.g. Twisted's `@inlineCallbacks`, which returns an uncoloured `Deferred`).
4. Threads: uncoloured.

Given that colour is clearly not mandatory for any of these, why does Bob talk so vehemently about colour?

Well, fundamentally, Bob's argument can be reframed. He gives himself away at the end of the post, when he asks "What language *isn't* colored?" and then says this:

> Any guess what [uncoloured langauges] have in common?
>
> *Threads*. Or, more precisely: *multiple independent callstacks that can be switched between*. It isn't strictly necessary for them to be operating system threads. Goroutines in Go, coroutines in Lua, and fibers in Ruby are perfectly adequate.

So, here we get to the rub. Bob's rant about function colour is *actually* a rant about proactor vs reactor programming. His argument is that it is better to write code that blocks on I/O than code that expects its I/O to be done for it elsewhere in the code. It is better to write functions that do I/O and then have their call stack switched out than it is to write functions that expect to return all the way up their call stack for the I/O to occur.

There are two problems with that argument.

One of them is that the argument is poorly framed. You can see that Bob allows "multiple independent callstacks that can be switched between", which is *exactly what* `async` / `await` *allows*! The only difference is that function colours are involved. So what Bob *actually* wants is for multiple independent callstacks *and* no coloured functions. That's fine.

The second problem is that options 1 through 3 in my above list exist in part because of the drawbacks of threaded programming. If we allow green threading with small stack sizes as an option (à la goroutines), then the reason we avoid threading in the first place is because we are concerned about shared-state pre-emptive multitasking. As Glyph says in Unyielding:

> When you're looking at a routine that manipulates some state, in a single-tasking, nonconcurrent system, you only have to imagine the state at the beginning of the routine, and the state at the end of the routine. To imagine the different states, you need only to read the routine and imagine executing its instructions in order from top to bottom. This means that the number of instructions you must consider is *n*, where *n* is the number of instructions in the routine. By contrast, in a system with arbitrary concurrent execution – one where multiple threads might concurrently execute this routine with the same state – you have to read the method *in every possible order, making the complexity* $n^n$.

Put another way, *threads are hard*. Before you get your pitchforks out, note that I said *hard*, not impossible. If you find working with shared-state pre-emptive multitasking easy, or that it maps well onto your thought processes, more power to you. I am not here to tell you to stop using threads. I am here to tell you why we have function colours.

So, if threads are hard (for some programmers), what are our other options? Well, the other option is cooperative multitasking. That is, only one bit of my code runs at once, and then it stops running to let other bits run. Options 1 through 3 of my above list give us that power. And, as I've demonstrated *at length* through this post, function colour is not *necessary* to achieve this kind of cooperative multitasking.

What it is, though, is *convenient*! That's really it. Despite Bob's many arguments about how annoying `async` / `await` is, at least in Python it reinstates to the world of cooperative multitasking the one thing he says he wants: *multiple independent callstacks that can be switched between*. It does so at the cost of providing coloured functions. These are inconvenient, but in return for paying the cost of that inconvenience they allow programmers to avoid being stuck in callback hell or requiring them to reason through the complex ways a callback chain can fire or propagate errors. This makes it easier for programmers to reason about control flow and error propagation and exception flow and all the other nice things that we want to think about.

*This* is why we have coloured functions: because they make lives easier for many programmers. They are an optional language tool you can use to write codebases that are

easier for some programmers to reason about. For me, I am happy to work with uncoloured callbacks: I have done it my entire programming career, and I'm finally at the point where I'm really pretty good at it. But, at least anecdotally, I can tell you that *many* (maybe even most) programmers are *not* comfortable doing it: they would like the clarity that `async` / `await` brings.

However, given that we're going to provide this convenience for our users, we should discuss briefly how to avoid the pitfalls of coloured functions. Because Bob is right: if you start colouring all your functions willy-nilly, you do start making things trickier.

# How To Live With Coloured Functions

Don't colour them.

I'm serious: every time you write `async` in your code you have made a small admission of defeat. You couldn't come up with a clear way to construct your code that didn't involve you needing to do some form of I/O in the middle of it. That's fine: as programmers our job is to balance the tradeoffs between expediency (using `async` whenever necessary) and good code structure (writing as little code as possible that requires I/O), and I'm sure as hell not going to judge you for choosing to put `async` everywhere.

But you don't *need* to do it. With care, there is no reason for more than about 10% of your codebase to be `async` functions. Almost all the rest of your code can just transform in-memory data representations from one form to another. You don't *have* to do this, and certainly if you don't care about code reusability or the various costs of writing `async` functions you should just go ahead and do that. But with care, and precision, most of your code doesn't need it.

So when we have discussions about extending the `async` paradigm to another Python language feature (currently, generators), we shouldn't get bogged down in whether we're creating two parallel worlds of functions. Whether we do that is a choice of the community. If the community is careful, we can write tools and implementations that *enable* `async`-using programs without *requiring* them. We can write as much code as possible in a synchronous style. We can potentially replace the core of most of our libraries with code written like this, allowing only the tiniest amount of wrapper code to transform from synchronous to asynchronous code.

All of this is possible. The community just needs to get comfortable with the idea that `async` is a tool for improving code clarity, rather than a requirement for writing asynchronous code. And I know I've spoken at length about Twisted here, but I think it's important to remember that Twisted has learnt these lessons and learnt them well. Perhaps more importantly, though, it taught those lessons to asyncio. Every time you think you'll use asyncio's Streams API, stop and consider whether you can use Protocols instead. Because streams are coloured, but Protocols aren't, and using Protocols will force you into a style where you avoid colouring your functions unnecessarily. This gives you the opportunity to write a codebase that is protected from the difficulties of coloured functions, where your code is testable without an event loop, where it can be run in a synchronous mode if you require it, and where it provides the opportunity for substantial code reuse.

## 2 Σ ↑

Code colour is a *convenience*, not a requirement. And we shouldn't get too worked up about it, or worry that we're segmenting our code. Instead, we should focus on using code colour sparingly, because while it adds clarity it also hurts reusability. Making that trade off well is one of the skills of being a programmer, and we should inform our community about that trade off, and then trust them to make the right call.

**Footnotes:**

1. For example, that Go's concurrency primitives are not different than threads and queues, or that there is a pervasive meme that states that async programming is too hard for novices even though we get novices to write iPhone and Android apps (and webapps!) all the time and those *are* async programs, or that Node.JS *still* basically doesn't have user threads and yet somehow the world hasn't collapsed down around that language. ↩

2. I should note that, of course, at some point you have to have synchronous code: the assembly-level main function is synchronous! What I mean for the rest of this article when I say "unable to be called from synchronous code" is actually "requires that it be run inside a coroutine-runner, or requires that it uses entirely non-blocking APIs". ↩

3. To be clear, a `Deferred` executes its callbacks by just calling them (e.g. `func()`), while asyncio's `Future` executes its callbacks by using `call_soon`. It is worth considering why asyncio's `Future` ended up this way, given that it was so heavily inspired by Twisted's `Deferred`. The answer is that it was a conscious choice, and was done for a few reasons. First, it makes `Future` "fairer" (for a specific definition of fair): it ensures that long chains of callbacks off multiple `Future`s are interleaved, rather than the first `Future` to fire getting executed to completion and stopping the other. Secondly, it prevents your code needing to worry about re-entrancy: a Twisted codebase needs to constantly be aware that firing a `Deferred` can cause the flow of execution to re-enter the method that is currently executing. This extra defensiveness can be a genuine cognitive burden and lead to subtle bugs, so there is some advantage in avoiding it. However, as I go on to explain here: this choice causes the `Future` to become, subtly, a coloured object, while a `Deferred` is not. ↩

## Written by Cory Benfield

He's just this guy, you know?

Updated July 29, 2016