**How the heck does async/await work in Python 3.5?**

* **Brett Cannon**

Read [more posts](https://snarky.ca/author/brett/) by this author.

[BRETT CANNON](https://snarky.ca/author/brett/)

11 FEB 2016 • 18 MIN READ

Being a core developer of [Python](https://www.python.org/) has made me want to understand how the language generally works. I realize there will always be obscure corners where I don't know every intricate detail, but to be able to help with issues and the general design of Python I feel like I should try and understand its core semantics and how things work under the hood.

But until recently I didn't understand how [async/await worked in Python 3.5](https://docs.python.org/3/whatsnew/3.5.html#whatsnew-pep-492). I knew that [yield from in Python 3.3](https://docs.python.org/3/whatsnew/3.3.html#pep-380) combined with [asyncio in Python 3.4](https://docs.python.org/3/library/asyncio.html#module-asyncio) had led to this new syntax. But having not done a lot of networking stuff -- which asyncio is not limited to but does focus on -- had led to me not really paying much attention to all of this async/await stuff. I mean I knew that:

yield from iterator

was (essentially) equivalent to:

for x in iterator:

yield x

And I knew that asyncio was an event loop framework which allowed for asynchronous programming, and I knew what those words (basically) meant on their own. But having never dived into the async/await syntax to understand how all of this came together, I felt I didn't understand asynchronous programming in Python which bothered me. So I decided to take the time and try and figure out how the heck all of it worked. And since I have heard from various people that they too didn't understand how this new world of asynchronous programming worked, I decided to write this essay (yes, this post has taken so long in time and is so long in words that my wife has labeled it an essay).

Now because I wanted a properly understanding of how the syntax worked, this essay has some low-level technical detail about how CPython does things. It's totally okay if it's more detail than you want or that you don't fully understand it as I don't explain every nuance of CPython internals in order to keep this from turning into a book (e.g., if you don't know that code objects have flags, let alone what a code object is, it's okay and you don't need to care to get something from this essay). I have tried to provide a more accessible summary at the end of every section so that you can skim the details if they turn out to be more than you want to deal with.

**A history lesson about coroutines in Python**

According to [Wikipedia](https://www.wikipedia.org/), "[Coroutines](https://en.wikipedia.org/wiki/Coroutine) are computer program components that generalize subroutines for nonpreemptive multitasking, by allowing multiple entry points for suspending and resuming execution at certain locations". That's a rather technical way of saying, "coroutines are functions whose execution you can pause". And if you are saying to yourself, "that sounds like generators", you would be right.

Back in [Python 2.2](https://docs.python.org/3/whatsnew/2.2.html), generators were first introduced by [PEP 255](https://www.python.org/dev/peps/pep-0255/) (they are also called generator iterators since generators implement the [iterator protocol](https://docs.python.org/3/library/stdtypes.html#iterator-types)). Primarily inspired by the [Icon programming language](http://www.cs.arizona.edu/icon/), generators allowed for a way to create an iterator that didn't waste memory when calculating the next value in the iteration in a dead-simple manner (you could always create a class that implemented \_\_iter\_\_() and \_\_next\_\_() that didn't store every value of the iterator, but that's a lot more effort). For instance, if you wanted to create your own version of range(), you could do it in an eager fashion by creating a list of integers:

def eager\_range(up\_to):

"""Create a list of integers, from 0 to up\_to, exclusive."""

sequence = []

index = 0

while index < up\_to:

sequence.append(index)

index += 1

return sequence

The problem with this, though, is that if you want a large sequence like the integers from 0 to 1,000,000, you have to create a list long enough to hold 1,000,000 integers. But when generators were added to the language, you could suddenly create an iterator that didn't need to create the whole sequence upfront with zero effort. Instead, all you had to do is have enough memory for one integer at a time.

def lazy\_range(up\_to):

"""Generator to return the sequence of integers from 0 to up\_to, exclusive."""

index = 0

while index < up\_to:

yield index

index += 1

Having a function pause what it is doing whenever it hit a yield expression -- although it was a statement until Python 2.5 -- and then be able to resume later is very useful in terms of using less memory, allowing for the idea of infinite sequences, etc.

But as you may have noticed, generators are all about iterators. Now having a better way to create iterators is obviously great (and this is shown when you define an \_\_iter\_\_() method on an object as a generator), but people knew that if we took the "pausing" part of generators and added in a "send stuff back in" aspect to them, Python would suddenly have the concept of coroutines in Python (but until I say otherwise, consider this all just a concept in Python; concrete coroutines in Python are discussed later on). And that exact feature of sending stuff into a paused generator was added in [Python 2.5](https://docs.python.org/3/whatsnew/2.5.html) thanks to [PEP 342](https://www.python.org/dev/peps/pep-0342/). Among other things, PEP 342 introduced the send() method on generators. This allowed one to not only pause generators, but to send a value back into a generator where it paused. Taking our range() example further, you could make it so the sequence jumped forward or backward by some amount:

def jumping\_range(up\_to):

"""Generator for the sequence of integers from 0 to up\_to, exclusive.

Sending a value into the generator will shift the sequence by that amount.

"""

index = 0

while index < up\_to:

jump = yield index

if jump is None:

jump = 1

index += jump

if \_\_name\_\_ == '\_\_main\_\_':

iterator = jumping\_range(5)

print(next(iterator)) # 0

print(iterator.send(2)) # 2

print(next(iterator)) # 3

print(iterator.send(-1)) # 2

for x in iterator:

print(x) # 3, 4

Generators were not mucked with again until [Python 3.3](https://docs.python.org/3/whatsnew/3.3.html) when [PEP 380](https://www.python.org/dev/peps/pep-0380/) added yield from. Strictly speaking, the feature empowers you to refactor generators in a clean way by making it easy to yield every value from an iterator (which a generator conveniently happens to be).

def lazy\_range(up\_to):

"""Generator to return the sequence of integers from 0 to up\_to, exclusive."""

index = 0

def gratuitous\_refactor():

nonlocal index

while index < up\_to:

yield index

index += 1

yield from gratuitous\_refactor()

By virtue of making refactoring easier, yield from also lets you chain generators together so that values bubble up and down the call stack without code having to do anything special.

def bottom():

# Returning the yield lets the value that goes up the call stack to come right back

# down.

return (yield 42)

def middle():

return (yield from bottom())

def top():

return (yield from middle())

# Get the generator.

gen = top()

value = next(gen)

print(value) # Prints '42'.

try:

value = gen.send(value \* 2)

except StopIteration as exc:

value = exc.value

print(value) # Prints '84'.

**Summary**

Generators in Python 2.2 let the execution of code be paused. Once the ability to send values back into the paused generators were introduced in Python 2.5, the concept of coroutines in Python became possible. And the addition of yield from in Python 3.3 made it easier to refactor generators as well as chain them together.

**What is an event loop?**

It's important to understand what an event loop is and how they make asynchronous programming possible if you're going to care about async/await. If you have done GUI programming before -- including web front-end work -- then you have worked with an event loop. But since having the concept of asynchronous programming as a language construct is new in Python, it's okay if you don't happen to know what an event loop is.

Going back to Wikipedia, an [event loop](https://en.wikipedia.org/wiki/Event_loop) "is a programming construct that waits for and dispatches events or messages in a program". Basically an event loop lets you go, "when A happens, do B". Probably the easiest example to explain this is that of the JavaScript event loop that's in every browser. Whenever you click something ("when A happens"), the click is given to the JavaScript event loop which checks if any onclick callback was registered to handle that click ("do B"). If any callbacks were registered then the callback is called with the details of the click. The event loop is considered a loop because it is constantly collecting events and loops over them to find what to do with the event.

In Python's case, asyncio was added to the standard library to provide an event loop. There's a focus on networking in asyncio which in the case of the event loop is to make the "when A happens" to be when I/O from a socket is ready for reading and/or writing (via the [selectors module](https://docs.python.org/3/library/selectors.html#module-selectors)). Other than GUIs and I/O, event loops are also often used for executing code in another thread or subprocess and have the event loop act as the scheduler (i.e., [cooperative multitasking](https://en.wikipedia.org/wiki/Cooperative_multitasking)). If you happen to understand Python's GIL, event loops are useful in cases where releasing the GIL is possible and useful.

**Summary**

Event loops provide a loop which lets you say, "when A happens then do B". Basically an event loop watches out for when something occurs, and when something that the event loop cares about happens it then calls any code that cares about what happened. Python gained an event loop in the standard library in the form of asyncio in Python 3.4.

**How async and await work**

**The way it was in Python 3.4**

Between the generators found in Python 3.3 and an event loop in the form of asyncio, Python 3.4 had enough to support asynchronous programming in the form of [concurrent programming](https://en.wikipedia.org/wiki/Concurrent_computing). *Asynchronous programming* is basically programming where execution order is not known ahead of time (hence asynchronous instead of synchronous). *Concurrent programming* is writing code to execute independently of other parts, even if it all executes in a single thread ([concurrency is **not** parallelism](http://blog.golang.org/concurrency-is-not-parallelism)). For example, the following is Python 3.4 code to count down every second in two asynchronous, concurrent function calls.

import asyncio

# Borrowed from http://curio.readthedocs.org/en/latest/tutorial.html.

@asyncio.coroutine

def countdown(number, n):

while n > 0:

print('T-minus', n, '({})'.format(number))

yield from asyncio.sleep(1)

n -= 1

loop = asyncio.get\_event\_loop()

tasks = [

asyncio.ensure\_future(countdown("A", 2)),

asyncio.ensure\_future(countdown("B", 3))]

loop.run\_until\_complete(asyncio.wait(tasks))

loop.close()

In Python 3.4, the [asyncio.coroutine decorator](https://docs.python.org/3/library/asyncio-task.html#asyncio.coroutine) was used to label a function as acting as a [coroutine](https://docs.python.org/3/reference/datamodel.html?#coroutine-objects) that was meant for use with asyncio and its event loop. This gave Python its first concrete definition of a coroutine: an object who implemented the methods added to generators in [PEP 342](https://www.python.org/dev/peps/pep-0342/) and represented by the [collections.abc.Coroutine abstract base class](https://docs.python.org/3/library/collections.abc.html#collections.abc.Coroutine). This meant that suddenly all generators implemented the coroutine interface even if they weren't meant to be used in that fashion. To fix this, asyncio required that all generators meant to be used as a coroutine had to be [decorated with asyncio.coroutine](https://docs.python.org/3/library/asyncio-task.html#asyncio.coroutine).

With this concrete definition of a coroutine (which matched an API that generators provided), you then used yield from on any [asyncio.Future object](https://docs.python.org/3/library/asyncio-task.html#future) to pass it down to the event loop, pausing execution of the coroutine while you waited for something to happen (being a future object is an implementation detail of asyncio and not important). Once the future object reached the event loop it was monitored there until the future object was done doing whatever it needed to do. Once the future was done doing its thing, the event loop noticed and the coroutine that was paused waiting for the future's result started again with its result sent back into the coroutine using its send() method.

Take our example above. The event loop starts each of the countdown() coroutine calls, executing until it hits yield from and the asyncio.sleep() function in one of them. That returns an asyncio.Future object which gets passed down to the event loop and pauses execution of the coroutine. There the event loop watches the future object until the one second is over (as well as checking on other stuff it's watching, like the other coroutine). Once the one second is up, the event loop takes the paused countdown() coroutine that gave the event loop the future object, sends the result of the future object back into the coroutine that gave it the future object in the first place, and the coroutine starts running again. This keeps going until all of the countdown() coroutines are finished running and the event loop has nothing to watch. I'll actually show you a complete example of how exactly all of this coroutine/event loop stuff works later, but first I want to explain how async and await work.

**Going from yield from to await in Python 3.5**

In Python 3.4, a function that was flagged as a coroutine for the purposes of asynchronous programming looked like:

# This also works in Python 3.5.

@asyncio.coroutine

def py34\_coro():

yield from stuff()

In Python 3.5, the [types.coroutine decorator](https://docs.python.org/3/library/types.html#types.coroutine) has been added to also flag a generator as a coroutine like asyncio.coroutine does. You can also use async def to syntactically define a function as being a coroutine, although it cannot contain any form of yield expression; only return and await are allowed for returning a value from the coroutine.

async def py35\_coro():

await stuff()

A key thing async and types.coroutine do, though, is tighten the definition of what a coroutine is. It takes coroutines from simply being an interface to an actual type, making the distinction between any generator and a generator that is meant to be a coroutine much more stringent (and the [inspect.iscoroutine() function](https://docs.python.org/3/library/inspect.html#inspect.iscoroutine) is even stricter by saying async has to be used).

You will also notice that beyond just async, the Python 3.5 example introduces await expressions (which are only valid within an async def). While await operates much like yield from, the objects that are acceptable to an await expression are different. Coroutines are definitely allowed in an await expression since the concept of coroutines are fundamental in all of this. But when you call await on an object , it technically needs to be an [*awaitable* object](https://docs.python.org/3/reference/datamodel.html?#awaitable-objects): an object that defines an \_\_await\_\_() method which returns an iterator which is **not** a coroutine itself . Coroutines themselves are also considered awaitable objects (hence why collections.abc.Coroutine inherits from collections.abc.Awaitable). This definition follows a Python tradition of making most syntax constructs translate into a method call underneath the hood, much like a + b is a.\_\_add\_\_(b) or b.\_\_radd\_\_(a) underneath it all.

How does the difference between yield from and await play out at a low level (i.e., a generator with types.coroutine vs. one with async def)? Let's look at the bytecode of the two examples above in Python 3.5 to get at the nitty-gritty details. The bytecode for py34\_coro() is:

>>> dis.dis(py34\_coro)

2 0 LOAD\_GLOBAL 0 (stuff)

3 CALL\_FUNCTION 0 (0 positional, 0 keyword pair)

6 GET\_YIELD\_FROM\_ITER

7 LOAD\_CONST 0 (None)

10 YIELD\_FROM

11 POP\_TOP

12 LOAD\_CONST 0 (None)

15 RETURN\_VALUE

The bytecode for py35\_coro() is :

>>> dis.dis(py35\_coro)

1 0 LOAD\_GLOBAL 0 (stuff)

3 CALL\_FUNCTION 0 (0 positional, 0 keyword pair)

6 GET\_AWAITABLE

7 LOAD\_CONST 0 (None)

10 YIELD\_FROM

11 POP\_TOP

12 LOAD\_CONST 0 (None)

15 RETURN\_VALUE

Ignoring the difference in line number due to py34\_coro() having the asyncio.coroutine decorator, the only visible difference between them is the [GET\_YIELD\_FROM\_ITER opcode](https://docs.python.org/3/library/dis.html#opcode-GET_YIELD_FROM_ITER) versus the [GET\_AWAITABLE opcode](https://docs.python.org/3/library/dis.html#opcode-GET_AWAITABLE). Both functions are properly flagged as being coroutines, so there's no difference there. In the case of GET\_YIELD\_FROM\_ITER, it simply checks if its argument is a generator or coroutine, otherwise it calls iter() on its argument (the acceptance of a coroutine object by the opcode for yield from is only allowed when the opcode is used from within a coroutine itself, which is true in this case thanks to the types.coroutine decorator flagging the generator as such at the C level with the CO\_ITERABLE\_COROUTINE flag on the code object).

But GET\_AWAITABLE does something different. While the bytecode will accept a coroutine just like GET\_YIELD\_FROM\_ITER, it will **not** accept a generator if has not been flagged as a coroutine. Beyond just coroutines, though, the bytecode will accept an *awaitable* object as discussed earlier. This makes yield from expressions and await expressions both accept coroutines while differing on whether they accept plain generators or awaitable objects, respectively.

You may be wondering why the difference between what an async-based coroutine and a generator-based coroutine will accept in their respective pausing expressions? The key reason for this is to make sure you don't mess up and accidentally mix and match objects that just happen to have the same API to the best of Python's abilities. Since generators inherently implement the API for coroutines then it would be easy to accidentally use a generator when you actually expected to be using a coroutine. And since not all generators are written to be used in a coroutine-based control flow, you need to avoid accidentally using a generator incorrectly. But since Python is not statically compiled, the best the language can offer is runtime checks when using a generator-defined coroutine. This means that when types.coroutine is used, Python's compiler can't tell if a generator is going to be used as a coroutine or just a plain generator (remember, just because the syntax says types.coroutine that doesn't mean someone hasn't earlier done types = spam earlier), and thus different opcodes that have different restrictions are emitted by the compiler based on the knowledge it has at the time.

One very key point I want to make about the difference between a generator-based coroutine and an async one is that only generator-based coroutines can actually pause execution and force something to be sent down to the event loop. You typically don't see this very important detail because you usually call event loop-specific functions like the [asyncio.sleep() function](https://docs.python.org/3/library/asyncio-task.html#asyncio.sleep) since event loops implement their own APIs and these are the kind of functions that have to worry about this little detail. For the vast majority of us, we will work with event loops rather than be writing them and thus only be writing async coroutines and never need to really care about this. But if you're like me and were wondering why you couldn't write something like asyncio.sleep() using only async coroutines, this can be quite the "aha!" moment.

**Summary**

Let's summarize all of this into simpler terms. Defining a method with async def makes it a *coroutine*. The other way to make a coroutine is to flag a generator with types.coroutine -- technically the flag is the CO\_ITERABLE\_COROUTINE flag on a code object -- or a subclass of collections.abc.Coroutine. You can only make a coroutine call chain pause with a generator-based coroutine.

An *awaitable object* is either a coroutine or an object that defines \_\_await\_\_() -- technically collections.abc.Awaitable -- which returns an iterator that is not a coroutine. An await expression is basically yield from but with restrictions of only working with awaitable objects (plain generators will not work with an await expression). An async function is a coroutine that either has return statements -- including the implicit return None at the end of every function in Python -- and/or await expressions (yield expressions are not allowed). The restrictions for async functions is to make sure you don't accidentally mix and match generator-based coroutines with other generators since the expected use of the two types of generators are rather different.

**Think of async/await as an API for asynchronous programming**

A key thing that I want to point out is actually something I didn't really think deeply about until I watched [David Beazley's Python Brasil 2015 keynote](https://www.youtube.com/watch?v=lYe8W04ERnY). In that talk, David pointed out that async/await is really an API for asynchronous programming (which [he reiterated to me on Twitter](https://twitter.com/dabeaz/status/696028946220056576)). What David means by this is that people shouldn't think that async/await as synonymous with asyncio, but instead think that asyncio is a framework that can utilize the async/await API for asynchronous programming.

David actually believes this idea of async/await being an asynchronous programming API that he has created the [curio project](https://pypi.python.org/pypi/curio) to implement his own event loop. This has helped make it clear to me that async/await allows Python to provide the building blocks for asynchronous programming, but without tying you to a specific event loop or other low-level details (unlike other programming languages which integrate the event loop into the language directly). This allows for projects like curio to not only operate differently at a lower level (e.g., asyncio uses future objects as the API for talking to its event loop while curio uses tuples), but to also have different focuses and performance characteristics (e.g., asyncio has an entire framework for implementing transport and protocol layers which makes it extensible while curio is simpler and expects the user to worry about that kind of thing but also allows it to run faster).

Based on the (short) history of asynchronous programming in Python, it's understandable that people might think that async/await == asyncio. I mean asyncio was what helped make asynchronous programming possible in Python 3.4 and was a motivating factor for adding async/await in Python 3.5. But the design of async/await is purposefully flexible enough to **not** require asyncio or contort any critical design decision just for that framework. In other words, async/await continues Python's tradition of designing things to be as flexible as possible while still being pragmatic to use (and implement).

**An example**

At this point your head might be awash with new terms and concepts, making it a little hard to fully grasp how all of this is supposed to work to provide you asynchronous programming. To help make it all much more concrete, here is a complete (if contrived) asynchronous programming example, end-to-end from event loop and associated functions to user code. The example has coroutines which represents individual rocket launch countdowns but that appear to be counting down simultaneously . This is asynchronous programming through concurrency; three separate coroutines will be running independently, and yet it will all be done in a single thread.

import datetime

import heapq

import types

import time

class Task:

"""Represent how long a coroutine should wait before starting again.

Comparison operators are implemented for use by heapq. Two-item

tuples unfortunately don't work because when the datetime.datetime

instances are equal, comparison falls to the coroutine and they don't

implement comparison methods, triggering an exception.

Think of this as being like asyncio.Task/curio.Task.

"""

def \_\_init\_\_(self, wait\_until, coro):

self.coro = coro

self.waiting\_until = wait\_until

def \_\_eq\_\_(self, other):

return self.waiting\_until == other.waiting\_until

def \_\_lt\_\_(self, other):

return self.waiting\_until < other.waiting\_until

class SleepingLoop:

"""An event loop focused on delaying execution of coroutines.

Think of this as being like asyncio.BaseEventLoop/curio.Kernel.

"""

def \_\_init\_\_(self, \*coros):

self.\_new = coros

self.\_waiting = []

def run\_until\_complete(self):

# Start all the coroutines.

for coro in self.\_new:

wait\_for = coro.send(None)

heapq.heappush(self.\_waiting, Task(wait\_for, coro))

# Keep running until there is no more work to do.

while self.\_waiting:

now = datetime.datetime.now()

# Get the coroutine with the soonest resumption time.

task = heapq.heappop(self.\_waiting)

if now < task.waiting\_until:

# We're ahead of schedule; wait until it's time to resume.

delta = task.waiting\_until - now

time.sleep(delta.total\_seconds())

now = datetime.datetime.now()

try:

# It's time to resume the coroutine.

wait\_until = task.coro.send(now)

heapq.heappush(self.\_waiting, Task(wait\_until, task.coro))

except StopIteration:

# The coroutine is done.

pass

@types.coroutine

def sleep(seconds):

"""Pause a coroutine for the specified number of seconds.

Think of this as being like asyncio.sleep()/curio.sleep().

"""

now = datetime.datetime.now()

wait\_until = now + datetime.timedelta(seconds=seconds)

# Make all coroutines on the call stack pause; the need to use `yield`

# necessitates this be generator-based and not an async-based coroutine.

actual = yield wait\_until

# Resume the execution stack, sending back how long we actually waited.

return actual - now

async def countdown(label, length, \*, delay=0):

"""Countdown a launch for `length` seconds, waiting `delay` seconds.

This is what a user would typically write.

"""

print(label, 'waiting', delay, 'seconds before starting countdown')

delta = await sleep(delay)

print(label, 'starting after waiting', delta)

while length:

print(label, 'T-minus', length)

waited = await sleep(1)

length -= 1

print(label, 'lift-off!')

def main():

"""Start the event loop, counting down 3 separate launches.

This is what a user would typically write.

"""

loop = SleepingLoop(countdown('A', 5), countdown('B', 3, delay=2),

countdown('C', 4, delay=1))

start = datetime.datetime.now()

loop.run\_until\_complete()

print('Total elapsed time is', datetime.datetime.now() - start)

if \_\_name\_\_ == '\_\_main\_\_':

main()

As I said, it's contrived, but if you run this in Python 3.5 you will notice that all three coroutines run independently in a single thread and yet the total amount of time taken to run is about 5 seconds. You can consider Task, SleepingLoop, and sleep() as what an event loop provider like asyncio and curio would give you. For a normal user, only the code in countdown() and main() are of importance. As you can see, there is no magic to async, await, or this whole asynchronous programming deal; it's just an API that Python provides you to help make this sort of thing easier.

**My hopes and dreams for the future**

Now that I understand how this asynchronous programming works in Python, I want to use it all the time! It's such an awesome concept that's so much better than something you would have used threads for previously. The problem is that Python 3.5 is so new that async/await is also very new. That means there are not a lot of libraries out there supporting asynchronous programming like this. For instance, to do HTTP requests you either have to construct the HTTP request yourself by hand (yuck), use a project like the [aiohttp framework](https://pypi.python.org/pypi/aiohttp) which adds HTTP on top of another event loop (in this case, asyncio), or hope more projects like the [hyper library](https://pypi.python.org/pypi/hyper) continue to spring up to provide an abstraction for things like HTTP which allow you to use whatever I/O library you want (although unfortunately hyper only supports HTTP/2 at the moment).

Personally, I hope projects like hyper take off so that we have a clear separation between getting binary data from I/O and how we interpret that binary data. This kind of abstraction is important because most I/O libraries in Python are rather tightly coupled to how they do I/O and how they handle data coming from I/O. This is a problem with the [http package in Python's standard library](https://docs.python.org/3/library/http.html#module-http) as it doesn't have an HTTP parser but a connection object which does all the I/O for you. And if you were hoping requests would support asynchronous programming, [your hopes have already been dashed](https://github.com/kennethreitz/requests/issues/2801) because the synchronous I/O that requests uses is baked into its design. This shift in ability to do asynchronous programming gives the Python community a chance to fix a problem it has with not having abstractions at the various layers of the network stack. And we have the perk of it not being hard to make asynchronous code run as if its synchronous, so tools filling the void for asynchronous programming can work in both worlds.

I also hope that Python gains some form of support in async coroutines for yield. Maybe this will require yet another keyword (maybe something like anticipate?), but the fact that you actually can't implement an event loop system with just async coroutines bothers me. Luckily, it turns out [I'm not the only one who thinks this](https://twitter.com/dabeaz/status/696014754557464576), and since the author of [PEP 492](https://www.python.org/dev/peps/pep-0492/) agrees with me, I think there's a chance of getting this quirk removed.

**Conclusion**

Basically async and await are fancy generators that we call coroutines and there is some extra support for things called awaitable objects and turning plain generators in to coroutines. All of this comes together to support concurrency so that we have better support for asynchronous programming in Python. It's awesome and much easier to use than comparable approaches like threads -- I wrote an end-to-end example of asynchronous programming in under 100 lines of commented Python code -- while still being quite flexible and fast (the [curio FAQ](http://curio.readthedocs.org/en/latest/#questions-and-answers) says that it runs faster than twisted by 30-40% but slower than gevent by 10-15%, and all while being implemented in pure Python; remember that [Python 2 + Twisted can use less memory and is easier to debug than Go](https://news.ycombinator.com/item?id=10402307), so just imagine what you could do with this!). I'm very happy that this landed in Python 3 and I look forward to the community embracing it and helping to flesh out its support in libraries and frameworks so we can all benefit from asynchronous programming in Python.