Native Coroutines

This lesson introduces the concept of native coroutines in Python.

Native Coroutine

In the previous section we looked at generator based coroutines. Unless you are working with older code that uses generator based coroutines, you don't need to use them. In fact `@asyncio.coroutine` will be deprecated in 3.10. Remember a coroutine is a method that has the ability to cede control during its execution and resume from where it left off when asked to do so. When generators were enhanced with additional methods via PEP-342, coroutines became possible in Python. Later on addition of **`yield from`** and asyncio framework allowed asynchronous programming using **generator based coroutines**. However, in Python 3.5 the language introduced support for native coroutines. By *native* it is meant that the language introduced syntax to specifically define coroutines, making them first class citizens in the language. Native coroutines can be defined using the **`async/await`** syntax. Before getting into further details, here is an example of a very simple native coroutine:

Native coroutine example

```
async def coro():
    await asyncio.sleep(1)
```

The above coroutine can be run with an event loop as follows:

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

The below code widget executes the above example.

```
import asyncio


async def coro():
    await asyncio.sleep(1)


if __name__ == "__main__":
    # run the coroutine
    loop = asyncio.get_event_loop()
    loop.run_until_complete(coro())
```

Async

We can create an native coroutine by using **`async def`**. A method prefixed with **`async def`** automatically becomes a native coroutine.

```
async def useless_native_coroutine():
  pass
```

The **`inspect.iscoroutine()`** method would return **`True`** for a coroutine object returned from the above coroutine function. Note that **`yield`** or **`yield from`** can't appear in the body of an async-defined method, else the occurrence would be flagged as a syntax error.

```
import inspect
import asyncio

async def useless_native_coroutine():
  pass

if __name__ == "__main__":
  coro = useless_native_coroutine()
  print(inspect.iscoroutine(coro))
```

Await

`await` can be used to obtain the result of a coroutine object's execution. We use await as:

```
await <expr>
```

where **<expr>** must be an awaitable object. Awaitable objects must implement the **`__await__()`** method that should return an iterator. If you recall **`yield from`** also expects its argument to be an iterable from which an iterator can be obtained. Under the hood **`await`** borrows implementation from **`yield from`** with an additional check if its argument is indeed an awaitable. The following objects are awaitable:

- A native coroutine object returned from calling a native coroutine function.
- A generator based coroutine object returned from a generator decorated with `@types.coroutine` or `@asyncio.coroutine`.

Decorated generator-based coroutines are awaitables, even though they do not have an `__await__()` method.

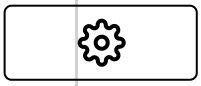
- `Future` objects are awaitable.
- `Task` objects are awaitable and `Task` is a subclass of `Future`.
- Objects defined with CPython C API with a `tp_as_async.am_await()` function, returning an iterator (similar to `__await__()` method).

Additionally, `await` must appear inside an async-defined method, else it's a syntax error.

Evolution of Coroutines

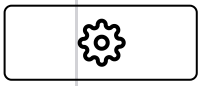
If you read literature offline or online regarding Python coroutines, you are likely to get confused as there have been several changes introduced in successive versions of Python with respect to coroutines. Additionally, Python deviates slightly from the standard definition of what a coroutine or a generator is and the fact that generator based coroutines can be used with native coroutines causes further confusion for a first-time reader. The table below documents the evolution of coroutines in different versions of Python. Feel free to refer to it as whenever in doubt.

Version	Genera-tors	Coroutines	Generator Based Coroutines	Native Coroutines
2.2	• Gen-	Don't exist	Don't exist	Don't exist



erators are introduced in Python via PEP-255

- Generator is essentially an iterator.
- Presence of **`yield`** in a function's body makes it a generator.



- Python generators conform with the standard definition of generators. `next()` can be used with generators to retrieve the next value in the

	se-quence.			
2.5	With the ability to send and receive data, Python generators satisfy the standard definition of a coroutine but are still labelled as generators in Python world.	• PEP-342 endows existing generators with three new methods, `send`, `throw` and `close` and coroutines are formally	Don't exist	Don't exist

usn-
ered
into
Pytho
n.

- **`yiel d`** gets the status of an ex-pres-sion.
- Both corou tines and gen-era-tors are repre-sent-ed by a *gen-erator object*. The dis-tinc-tion is con-

		ceptu- al wher e a gen- erator func- tion can only re- turn val- ues wher eas a corou tine can send and re- ceive val- ues.		
3.3	• **`yield from`** is in-	Python coroutines can be chained	Don't exit	Don't exit

			troduced via PEP-380 and allows a generator to delegate part of its operations to a subgenerator. **`yield from`** also establishes a transparent bidirectional	similar to how Unix commands can be piped.		

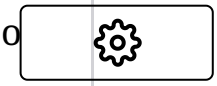
	com- muni-cation chan-nel be-tween the inner-most gen-erator and the caller of the outer-most gen-era-tor.			
3.4 (asyncio module intro-duced)	Behave as in previ-ous ver-sion. Don't run with asyncio's event loop.	Behave as in previ-ous ver-sion. A generator returns values, a coroutine	• PEP-3156 for-mally intro-duces asyn-	Don't exist

receives

and can return values and a generator-based coroutine must include **`yield from`** in it's function body.

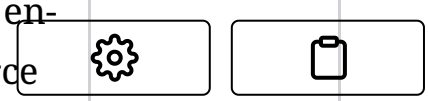
cio module.

- Generator based coroutines can be run using asyncio's event loop.
- Existence of **`yield from`** in the body of a function makes it a coroutine -



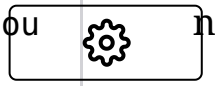
also known as a generator based coroutine.

- The decorator `@asyncio.coroutine` is introduced to mark generator based coroutines with. The decorator isn't strict-



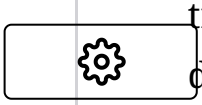
ly enforced and used for documentation purposes. Additionally if the decorator is applied on a function that is neither a generator nor a coroutine, the decorator will

			wrap the function in a coroutine. • The decorator is slated for removal in 3.10.	
3.5	Behave as in previous version	Behave as in previous version	• Must be decorated with `@asyncio.coroutine` or `@types.-`	• PEP-492 makes coroutines first class citizens in Pytho



corou tine in order to be interoperable with native coroutines.

- Native coroutines become part of Python with the introduction of **`async/await`** syntax via PEP-492.
- `@types.coroutine` decorator is in-



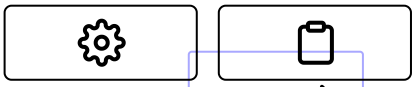
duced to allow generator based coroutines to work with native coroutines.

- **Going forward `async/await` is the preferred syntax to write coroutines.**

Needless to say the terminology around coroutines gets messy. We already mentioned that generators are considered a specialized case of coroutines in the world outside Python. Within the Python world, the community started with vanilla generators, then enhanced them with new methods. A generator object that received values from outside was referred to as a coroutine even though the object belonged to the same class as a generator object that produced values and was referred to as a generator. At this point, the distinction between coroutines and generators was defined by their ability to produce or consume data. Next, when version 3.4 came out, generator functions which had **`yield from`** in their function bodies were called as **generated-based coroutines**. And finally, version 3.5 brought with it native coroutines. Inevitably, the changes over the years cause confusion, because for instance, if you read an article written prior to version 3.4, a coroutine means a different entity compared to a coroutine referred in an article post version 3.5.

As things stand now, generators are used to refer to functions that produce values only, vanilla coroutines receive values only, generator-based coroutines are identified via the presence of **`yield from`** in the method body and finally native coroutines are defined using the **`async/await`** syntax.

Another way to summarize our discussion is: generators return values using yield for their invokers, generators that can receive values from outside are coroutines, generators with yield from in their function bodies are generator-based coroutines and methods defined using async-def are native coroutines. Use the `@asyncio.coroutine` or `@types.coroutine` decorators on generator-based coroutines to make them compatible with native co-routines.



Back

Next

... continued

... continued

Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/native-coroutines__asyncio__python-concurrency-for-senior-engineering-interviews)