(/learn)

Generator Based Coroutines

This lesson goes into the details of generator-based coroutines in Python.

Generator Based Coroutines

Some of the readers might want to skip this lesson and head straight to native coroutines section but following through the discussion on legacy generator based coroutines can be useful when working with older code-bases.

Coroutines formally became part of Python in version 2.4 when generators were enhanced via PEP-342 (https://www.python.org/dev/peps/pep-0342/). And in Python 3.4, asyncio framework introduced an event loop which enabled asynchronous programs in Python. Note that with PEP-342, all generators received the new methods. So all generators fit the textbook definition of a coroutine. In fact, a post PEP-342 *Python generator* is functionally equivalent of what the world outside Python would call a coroutine. However, Python created a distinction between Python generators and generators that were meant to be used as coroutines. These coroutines are called **generator based coroutines** and require the decorator `@asynio.coroutine` to be added to the function definition though this isn't strictly enforced. The decorator enables compatibility with **`async def`** coroutines, and also serves as documentation.

Generator based coroutines use **`yield from`** syntax instead of **`yield`**. A coroutine can:

- yield from another coroutine
- yield from a future
- return an expression
- raise exception

To sum up, a function that uses **`yield from`** becomes a coroutine and requires the `@asyncio.coroutine` decorator. If a function doesn't use **`yield from`** adding the decorator will make it a coroutine. Consider the following method

```
@asyncio.coroutine
def hello_world():
    print("hello world")
```

The above method becomes a coroutine with the addition of the decorator and can be run using the event loop as follows:

```
coro_obj = hello_world()
asyncio.get_event_loop().run_until_complete(coro_obj)
```

In the runnable script below remove the decorator and observe the event loop throw an exception.

```
import asyncio

@asyncio.coroutine
def hello_world():
    print("hello world")


if __name__ == "__main__":
    asyncio.get_event_loop().run_until_complete(hello_world())
```

Coroutine Function vs Coroutine Object

In the above example, the function `hello_world()` is called a **coroutine function** whereas the object `coro_obj` received by calling the coroutine function is called the **coroutine object**. At times, the two are used interchangeably but where disambiguation is required we can use appropriate jargon.

Generators vs Generator-based Coroutines

Python introduces this distinction between what it calls *generators* and *generator based coroutines* when in fact both of them are just coroutines according to standard definitions (outside of Python). This causes confusion since Python introduces its own idiosyncratic notion of a generator and a coroutine. Consider the following useless generator function.

Coroutine

```
def silly():
    item = yield None
    print(item)
```

The `silly()` method is a coroutine function by standard definition (outside of Python) because it can suspend and resume execution but in Python world it is a coroutine because it can receive values. And interestingly enough the coroutine object is of type generator! One way to tell apart these differences is to use the `inspect` module APIs. Consider the code below:

```
    s = silly()

    # prints false
    print(inspect.isawaitable(s))

    # prints true
    print(isinstance(s, types.GeneratorType))
```

In the code above we use the method **`inspect.isawaitable()`** introduced in Python 3.5 to detect if an object is a coroutine object. An *awaitable* object is one that can be used in an **`await`** expression. We'll cover it in greater depth in later sections but for now consider it a litmus test to detect a coroutine object or to distinguish between generator-based coroutines and just plain generators.

Compatibility with Native Coroutines

Generator-based coroutines predate the newer **`async/await`** syntax. They are Python generators that use yield from expressions to await on futures and other coroutines.

If we add the `@asyncio.coroutine` decorator then **`silly()`** becomes awaitable and **`inspect.isawaitable(s)`** will return true. The decorator allows the generator based coroutines to work with native coroutines defined using the newer **`async/await`** syntax. Remember the two conventions a generator based coroutine must adhere to in order to work with async.io framework:

- Use the `@asyncio.coroutine` decorator
- Use **`yield from`** instead of **`yield`**

```
import asyncio
import inspect
import types

# Uncomment the following line to observe isawaitable return True.
# If the widget uses Python 3.5, an int might be returned instead of True.

#@asyncio.coroutine
def silly():
    item = yield None
    print(item)


if __name__ == "__main__":
    s = silly()
    print(inspect.isawaitable(s))
    print(isinstance(s, types.GeneratorType))
```

The decorator `@asyncio.coroutine` is not strictly enforced. You can have a coroutine without the decorator as the below example demonstrates. However, remember this same example would not work in versions of Python post 3.5

Coroutine without decorate in Python 3.5

```
import asyncio

def test():
    yield from asyncio.sleep(1)


if __name__ == "__main__":
    asyncio.get_event_loop().run_until_complete(test())
```

The code widget below will run without errors if it runs Python 3.5 under the hood. If upgraded to 3.7, the example would not run successfully.

```
import asyncio

def test():
    yield from asyncio.sleep(1)


if __name__ == "__main__":
    asyncio.get_event_loop().run_until_complete(test())
```

Back
... continued

Next
... continued