Chaining Coroutines

This lesson explains how to chain coroutines with an example.

Chaining Coroutines

One of the most prominent uses of coroutines is to chain them to process data pipelines. We can chain coroutines in a fashion similar to how we pipe Unix commands in a shell.

The idea is that the input passes through the first coroutine, which may perform some actions on the input and then passes on the modified data to the second coroutine which may perform additional operations on the input. The input travels through the chain of coroutines with each coroutine applying some operation on the input until the input reaches the last coroutine from where it is yielded to the original caller. Let's consider the following example, which computes the values for the expression $\mathbf{x^2 + 3}$ for the first hundred natural numbers. We manually work the data pipeline using the `next()` method so we'll setup chain without worrying about the changes required to make it work with the asyncio's event loop. The setup is as follows:

- The first coroutine produces natural numbers starting from 1.
- The second coroutine computes the square of each passed in input.
- The last function is a generator and adds 3 to the value passed into it and yields the result.

```
def coro3(k):
    yield (k + 3)


def coro2(j):
    j = j * j
    yield from coro3(j)


def coro1():
    i = 0
    while True:
        yield from coro2(i)
        i += 1


if __name__ == "__main__":

    # The first 100 natural numbers evaluated for the followin
g expression
    # x^2 + 3

    cr = coro1()
    for v in range(100):
        print("f({0}) = {1}".format(v, next(cr)))
```

```
import asyncio

def coro3(k):
    yield (k + 3)


def coro2(j):
    j = j * j
    yield from coro3(j)


def coro1():
    i = 0
    while True:
        yield from coro2(i)
        i += 1


if __name__ == "__main__":

    # The first 100 natural numbers evaluated for the following expression
    # x^2 + 3

    cr = coro1()
    for v in range(100):
        print("f({0}) = {1}".format(v, next(cr)))
```

In the example above, the end of the chain consists of a generator, however, this chain wouldn't run with the asyncio's event loop since it doesn't work with generators. One way to fix is to change the last generator into an ordinary function that returns a future with the result computed. The method **`coro3()`** would change to:

```
def coro3(k):
    f = Future()
    f.set_result(k + 3)
    f.done()
    return f
```

```
import asyncio
from asyncio import Future

def coro3(k):
    future = Future()
    future.set_result(k+3)
    future.done()
    return future


def coro2(j):
    j = j * j
    result = yield from coro3(j)
    return result

def coro1():
    i = 0
    while i < 100:
        final_result = yield from coro2(i)
        print("f({0}) = {1}".format(i, final_result))
        i += 1


if __name__ == "__main__":

    # The first 100 natural numbers evaluated for the following expression
    # x^2 + 3

    cr = coro1()
    loop = asyncio.get_event_loop()
    loop.run_until_complete(cr)
```

Yet another way is to tack on the `@asyncio.coroutine` onto the **`coro3()`** and return from it instead of yielding. The change would look like as follows:

```
@asyncio.coroutine
def coro3(k):
    return k + 3
```

An important caveat to consider is that if we instead used the `@types.coroutine` decorator the program would fail. This is because `@asyncio.coroutine` can convert an ordinary function into a coroutine but `@types.coroutine` can't as noted in the previous section.

```
import asyncio

@asyncio.coroutine
def coro3(k):
    return k + 3


def coro2(j):
    j = j * j
    result = yield from coro3(j)
    return result

def coro1():
    i = 0
    while i < 100:
        final_result = yield from coro2(i)
        print("f({0}) = {1}".format(i, final_result))
        i += 1


if __name__ == "__main__":

    # The first 100 natural numbers evaluated for the following expression
    # x^2 + 3

    cr = coro1()
    loop = asyncio.get_event_loop()
    loop.run_until_complete(cr)
```

Note that in the previous examples we didn't decorate **`coro1()`** and **`coro2()`** with `@asyncio.coroutine`. Both the functions are generator-based coroutine functions because of the presence of **`yield from`** in

their function bodies. Additionally, the appearance of the decorator isn't strictly enforced but if you put on the decorators the program would still work correctly.

Chaining Native Coroutines

Similar to generators and generator-based coroutines we can also chain native coroutines. The above example is refactored as a chain of native coroutines below:

```
import asyncio


async def coro3(k):
    return k + 3


async def coro2(j):
    j = j * j
    res = await coro3(j)
    return res


async def coro1():
    i = 0
    while i < 100:
        res = await coro2(i)
        print("f({0}) = {1}".format(i, res))
        i += 1


if __name__ == "__main__":
    # The first 100 natural numbers evaluated for the followin
g expression
    # x^2 + 3
    cr = coro1()
    loop = asyncio.get_event_loop()
    loop.run_until_complete(cr)
```

```
import asyncio


async def coro3(k):
    return k + 3


async def coro2(j):
    j = j * j
    res = await coro3(j)
    return res


async def coro1():
    i = 0
    while i < 100:
        res = await coro2(i)
        print("f({0}) = {1}".format(i, res))
        i += 1


if __name__ == "__main__":
    # The first 100 natural numbers evaluated for the following expression
    # x^2 + 3
    cr = coro1()
    loop = asyncio.get_event_loop()
    loop.run_until_complete(cr)
```

Back

Mixing Native & Generator Based Cor...

Next

Future & Tasks

Mark as Completed

Report an Issue

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