Mixing Native & Generator Based Coroutines

This lesson demonstrates how generator-based coroutines and native coroutines can be interoperable.

Mixing Native & Generator Based Coroutines

In order to maintain backward compatibility, generator-based coroutines can work with native coroutines by using appropriate decorators. In this lesson, we first look at the differences between the two types of coroutines and then examine the decorators that can be used to make the two work with each other.

Native Coroutines vs Generator-based Coroutines

Generator based coroutines and native coroutines have differences between themselves which are listed below:

- Native coroutines don't implement the **`__iter__()`** and **`__next__()`** methods and therefore can't be iterated upon.
- Generator based coroutines can't **`yield from`** a native coroutine. The following will result in a syntax error:

```
def gen_based_coro():
```

```
def gen_based_coro():
    yield from asyncio.sleep(10)
```

However, if we decorate the **`gen_based_coro()`** with the decorator `@asyncio.coroutine` then it is allowed to **`yield from`** a native coroutine. The following is thus legal:

```
@asyncio.coroutine
def gen_based_coro():
    yield from asyncio.sleep(10)
```

- Methods **`inspect.isgenerator()`** and **`inspect.isgeneratorfunction()`** return false for native coroutine objects while true for generator-based coroutine objects and functions.

@asyncio.coroutine

Adding the `@asyncio.coroutine` decorator makes generator based coroutines compatible with native coroutines. Without the decorator it would not be possible to **`yield from`** a native coroutine inside of a generator based coroutine. Consider the example below:

```
import asyncio


@asyncio.coroutine
def gen_based_coro():
    yield from asyncio.sleep(1)


if __name__ == "__main__":
    gen = gen_based_coro()
    next(gen)
```

The decorator also allows a generator based coroutine to be awaited in a native coroutine. Consider the below example:

```
import asyncio

@asyncio.coroutine
def gen_based_coro():
    return 10

async def main():
    rcvd = await gen_based_coro()
    print("native coroutine received: " + str(rcvd))

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

```
import asyncio

@asyncio.coroutine
def gen_based_coro():
    return 10

async def main():
    rcvd = await gen_based_coro()
    print("native coroutine received: " + str(rcvd))

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

Note that @asyncio.coroutine decorator is slated to be removed starting in Python 3.10.

@types.coroutine

The `@types.coroutine` was introduced in PEP-492 and did the same job as `@asyncio.coroutine` that is make generator-based coroutines compatible with native coroutines. However, there's a slight but mostly unimportant difference, the `@asyncio.coroutine` will allow a function that isn't a generator to become a generator and be compatible with native coroutines. Consider the following example:

```
async def main():
    await useless()

@asyncio.coroutine
def useless():
    pass


if __name__ == "__main__":

    loop = asyncio.get_event_loop()
    loop.run_until_complete(main())

    print(isinstance(useless(), types.GeneratorType))
```

```
import asyncio
import types


async def main():
    await useless()

@asyncio.coroutine
def useless():
    pass


if __name__ == "__main__":

    loop = asyncio.get_event_loop()
    loop.run_until_complete(main())

    # prints true
    print("isinstance(useless(), types.GeneratorType) : " + str(isinstance(useless(

    # prints true
    print("asyncio.iscoroutinefunction(useless) : " + str(asyncio.iscoroutinefuncti
```

In the example above, the method **`useless()`** is an ordinary function without the `@asyncio.coroutine` decorator. Adding the decorator makes the ordinary function into a coroutine and compatible with native coroutines.

Contrast the above output with the one below, where we switch the decorator to `@types.coroutine` on the **`useless()`** method. The code throws an exception and the **`isinstance`** method returns false demonstrating that `@types.coroutine` decorator doesn't turn an ordinary function into a coroutine. Also, a function decorated with `@asyncio.coroutine` will test true for **`asyncio.iscoroutinefunction()`**, while one decorated with `@types.coroutine` will test false.

```
import asyncio
import types

async def main():
    await useless()

@types.coroutine
def useless():
    pass


if __name__ == "__main__":

    # prints false
    print("isinstance(useless(), types.GeneratorType) : " + str(isinstance(useless(

    # prints false
    print("asyncio.iscoroutinefunction(useless) : " + str(asyncio.iscoroutinefuncti

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

← Back

... continued

Next →

Chaining Coroutines

☑ Mark as Completed

Report an Issue

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