... continued

This lesson continues the discussion on yield-from.

Returning Values from Yield from

`yield from` can also receive the value returned by the subgenerator. Consider the example below. The **`nested_generator()`** returns the value 999 and is printed by the **`outer_generator()`** .

```
def nested_generator():
    i = 0
    while i < 5:
        i += 1
        yield i

    # return a value from the sub-generator
    return 999


def outer_generator():
    nested_gen = nested_generator()
    # outer generator receives the return value of the sub-generator
    ret_val = yield from nested_gen
    print("received in outer generator: " + str(ret_val))


if __name__ == "__main__":

    gen = outer_generator()

    for item in gen:
        print(item)
```

```
def nested_generator():
    i = 0
    while i < 5:
        i += 1
        yield i

    return 999


def outer_generator():
    nested_gen = nested_generator()
    ret_val = yield from nested_gen
    print("received in outer generator: " + str(ret_val))


if __name__ == "__main__":

    gen = outer_generator()

    for item in gen:
        print(item)
```

The following two sub sections talk about yielding from generator based coroutines and futures. You may come back to these sections once you read the generator based coroutines and futures sections. However, since the concepts are related to **`yield from`** we present them here.

Yielding from Coroutines

It may seem confusing that we can **`yield from`** generator-based coroutines, after all, the Python documentation says that **`yield from`** can be used as **`yield from <expression>`** where expression must be an iterable from which one can retrieve an iterator. But one may wonder how can one substitute coroutine for an iterable. Well, coroutines are also generators and an **isinstance()** test for a coroutine and a generator returns true. Consider the below example:

```
# hello_routine isn't a traditional generator that
# returns items in a sequence
@asyncio.coroutine
def hello_routine():
    print("hello world")


# coro becomes a generator based coroutine itself on
# account of using the yield from expression
def coro():
    yield from hello_routine()


if __name__ == "__main__":
    gen = coro()

    # hello_routine is an iterable, generator and a
    # generator based coroutine
    print(isinstance(hello_routine(), types.GeneratorType))
    print(isinstance(hello_routine(), Iterable))
    print(isinstance(hello_routine(), Generator))

    # in fact, coro which yields from hello_routine is also
    # an iterable and a generator based coroutine
    print(isinstance(gen, types.GeneratorType))
    print(isinstance(gen, Iterable))
    print(isinstance(gen, Generator))

    for _ in gen:
        pass
```

```
import asyncio
import types
from collections.abc import Iterable, Generator

# hello_routine isn't a traditional generator that
# returns items in a sequence
@asyncio.coroutine
def hello_routine():
    print("hello world")


# coro becomes a generator based coroutine itself on
# account of using the yield from expression
def coro():
    yield from hello_routine()


if __name__ == "__main__":
    gen = coro()

    # hello_routine is an iterable, generator and a
    # generator based coroutine
    print(isinstance(hello_routine(), types.GeneratorType))
    print(isinstance(hello_routine(), Iterable))
    print(isinstance(hello_routine(), Generator))

    # in fact, coro which yields from hello_routine is also
    # an iterable and a generator based coroutine
    print(isinstance(gen, types.GeneratorType))
    print(isinstance(gen, Iterable))
    print(isinstance(gen, Generator))

    for _ in gen:
        pass
```

Yielding from Native Coroutines

PEP-492 introduced the **`await`** keyword which allows us to suspend execution until the result of **`await`**'s argument (a coroutine) becomes available. **`await`** is similar to **`yield from`** and borrows most of the implementation from **`yield from`**. Legacy code based on generator implementation from **yield from**. Legacy code based on generator-based coroutines can't use **await** as **await** can only be used inside an **asyn** function. Such generator-based coroutines can use **yield from** to await results of native coroutines.

In the example below a generator-based coroutine awaits completion of a native coroutine using **yield from**.

```
import asyncio
import types

# generator-based coroutine using yield-from
# to await native coroutine asyncio.sleep()
@asyncio.coroutine
def gen_based_coro():
    yield from asyncio.sleep(1)


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

Yielding from Futures

We can also yield from **asyncio.Future**. In fact, futures are made iterable with the addition of the **__iter__()** method. Don't confuse **asyncio.Future** with **concurrent.futures.Future**. The former implements the **__iter__()** method and the latter doesn't. Consider the following example.

```
from asyncio import Future

if __name__ == "__main__":
    f = Future()

    # Future has an __iter__() method
    it = f.__iter__()
    print(next(it))

    # set the future's result and mark it done
    f.set_result("hello")
    f.done()

    try:
        next(it)
    except StopIteration as si:
        print(si.value)
```

> Last but not the least, we can also **`yield from`** task objects as **`Task`** is a subclass of **`Future`**. An example appears below:

```
import asyncio

def yield_from_task_example():
    # create a task to sleep for 5 seconds
    task = asyncio.get_event_loop().create_task(asyncio.sleep(5))
    yield from task



if __name__ == "__main__":

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

> The point is you may come across code that yields from coroutines, futures or tasks and it may not make sense since the two don't return items in a sequence as traditional generators do. In fact, any class that implements the iteration protocol can appear on the right-hand side of a `yield from` expression.

← Back

Yield From

Next →

Generator Based Coroutines

☑ Mark as Completed

Report an Issue

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