Cooperative Multitasking with Coroutines

Coroutines are a language construct designed for concurrent operation. A coroutine function creates a coroutine object when called, and the caller can then run the code of the function using the coroutine's `send()` method. A coroutine can pause execution using the `await` keyword with another coroutine. While it is paused, the coroutine's state is maintained, allowing it to resume where it left off the next time it is awakened.

Starting a Coroutine

There are a few different ways to have the `asyncio` event loop start a coroutine. The simplest is to use `run_until_complete()`, passing the coroutine to it directly.

```python
# asyncio_coroutine.py

import asyncio


async def coroutine():
    print('in coroutine')


event_loop = asyncio.get_event_loop()
try:
    print('starting coroutine')
    coro = coroutine()
    print('entering event loop')
    event_loop.run_until_complete(coro)
finally:
    print('closing event loop')
    event_loop.close()
```

The first step is to obtain a reference to the event loop. The default loop type can be used, or a specific loop class can be instantiated. In this example, the default loop is used. The `run_until_complete()` method starts the loop with the coroutine object and stops the loop when the coroutine exits by returning.

```
$ python3 asyncio_coroutine.py

starting coroutine
entering event loop
in coroutine
closing event loop
```

Returning Values from Coroutines

The return value of a coroutine is passed back to the code that starts and waits for it.

```python
# asyncio_coroutine_return.py

import asyncio


async def coroutine():
    print('in coroutine')
    return 'result'


event_loop = asyncio.get_event_loop()
try:
    return_value = event_loop.run_until_complete(
        coroutine()
    )
    print('it returned: {!r}'.format(return_value))
finally:
    event_loop.close()
```

In this case, run_until_complete() also returns the result of the coroutine it is waiting for.

```
$ python3 asyncio_coroutine_return.py

in coroutine
it returned: 'result'
```

Chaining Coroutines

One coroutine can start another coroutine and wait for the results. This makes it easier to decompose a task into reusable parts. The following example has two phases that must be executed in order, but that can run concurrently with other operations.

```
# asyncio_coroutine_chain.py
```

```
import asyncio


async def outer():
    print('in outer')
    print('waiting for result1')
    result1 = await phase1()
    print('waiting for result2')
    result2 = await phase2(result1)
    return (result1, result2)


async def phase1():
    print('in phase1')
    return 'result1'


async def phase2(arg):
    print('in phase2')
    return 'result2 derived from {}'.format(arg)


event_loop = asyncio.get_event_loop()
try:
    return_value = event_loop.run_until_complete(outer())
    print('return value: {!r}'.format(return_value))
finally:
    event_loop.close()
```

The await keyword is used instead of adding the new coroutines to the loop, because control flow is already inside of a coroutine being managed by the loop so it isn't necessary to tell the loop to manage the new coroutines.

```
$ python3 asyncio_coroutine_chain.py

in outer
waiting for result1
in phase1
waiting for result2
in phase2
return value: ('result1', 'result2 derived from result1')
```

Generators Instead of Coroutines

Coroutine functions are a key component of the design of asyncio. They provide a language construct for stopping the execution of part of a program, preserving the state of that call, and re-entering the state at a later time, which are all important capabilities for a concurrency framework.

Python 3.5 introduced new language features to define such coroutines natively using async def and to yield control using await, and the examples for asyncio take advantage of the new feature. Earlier versions of Python 3 can use generator functions wrapped with the asyncio.coroutine() decorator and yield from to achieve the same effect.

```
# asyncio_generator.py

import asyncio


@asyncio.coroutine
def outer():
    print('in outer')
    print('waiting for result1')
    result1 = yield from phase1()
    print('waiting for result2')
    result2 = yield from phase2(result1)
    return (result1, result2)


@asyncio.coroutine
def phase1():
    print('in phase1')
    return 'result1'


@asyncio.coroutine
def phase2(arg):
    print('in phase2')
    return 'result2 derived from {}'.format(arg)


event_loop = asyncio.get_event_loop()
try:
    return_value = event_loop.run_until_complete(outer())
    print('return value: {!r}'.format(return_value))
finally:
    event_loop.close()
```

The preceding example reproduces asyncio_coroutine_chain.py using generator functions instead of native coroutines.

```
$ python3 asyncio_generator.py

in outer
waiting for result1
in phase1
waiting for result2
in phase2
return value: ('result1', 'result2 derived from result1')
```

Asynchronous Concurrency Concepts

Scheduling Calls to Regular Functions

Quick Links

Starting a Coroutine
Returning Values from Coroutines
Chaining Coroutines
Generators Instead of Coroutines

This page was last updated 2016-12-18.

Navigation

Asynchronous Concurrency Concepts
Scheduling Calls to Regular Functions

The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

Looking for examples for Python 2?

This Site

Module Index
Index

Other Writing

Blog
The Python Standard Library By Example