... continued

Continues the discussion on generator based coroutines.

Generator-based Coroutine Example

The simplest generator based coroutine we can write is as follows:

Example Coroutine

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

The coroutine sleeps for one second. Note the decorator and the use of `yield from`. Without, either of them we wouldn't be able to use the coroutine with async.io. The `yield from` statement gives up control back to the event loop and resumes execution after the coroutine `asyncio.sleep()` has completed. Note that `asyncio.sleep()` is itself a coroutine. Let us modify this coroutine to call another coroutine which performs the sleep. The changes are shown below:

```
@asyncio.coroutine
def go_to_sleep(sleep):
    print("sleeping for " + str(sleep) + " seconds")
    yield from asyncio.sleep(sleep)


@asyncio.coroutine
def do_something_important(sleep):
    # what is more important than getting
    # enough sleep!
    yield from go_to_sleep(sleep)
```

Now imagine we invoke the coroutine **`do_something_important()`** thrice serially with the values 1, 2 and 3 respectively. Without using threads or multiprocessing the serial code will execute in **1 + 2 + 3 = 6 seconds**, however, if we leverage async.io the same code can complete in roughly **3 seconds** even though all of the invocations run in the same thread. The intuition is that whenever a blocking operation is encountered the control is passed back to the event loop and execution is only resumed when the blocking operation has completed. Run the snippet below and observe the total execution time.

```
from threading import current_thread
import asyncio
import time

@asyncio.coroutine
def go_to_sleep(sleep):
    print("sleeping for " + str(sleep) + " seconds in thread " + current_thread().g
    yield from asyncio.sleep(sleep)


@asyncio.coroutine
def do_something_important(sleep):
    yield from go_to_sleep(sleep)


if __name__ == "__main__":
    now = time.time()
    loop = asyncio.get_event_loop()
    loop.run_until_complete(
        asyncio.gather(do_something_important(1), do_something_important(2), do_som
    end = time.time()
    print("total time to run the script : " + str(end - now))
    loop.close()
```

Note the output of the above program shows that all three invocations of **`do_something_important()`** run in the same thread. To drive the point home, we now change the **`go_to_sleep()`** to use **`time.sleep()`** and the execution becomes serial and execution times equals 6 seconds.

```
from threading import current_thread
import asyncio
import time

@asyncio.coroutine
def go_to_sleep(sleep):
    print("sleeping for " + str(sleep) + " seconds in thread " + current_thread().g
    time.sleep(sleep)


@asyncio.coroutine
def do_something_important(sleep):
    yield from go_to_sleep(sleep)


if __name__ == "__main__":
    now = time.time()
    loop = asyncio.get_event_loop()
    loop.run_until_complete(
        asyncio.gather(do_something_important(1), do_something_important(2), do_som
    end = time.time()
    print("total time to run the script : " + str(end - now))
    loop.close()
```

Back

Generator Based Coroutines

Next

Native Coroutines

Mark as Completed

Report an Issue

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