(/learn)

... continued

In this lesson, we'll examine some of important APIs we can use to work with the event loop.

Running the Event Loop

With Python 3.7+ the preferred way to run the event loop is to use the **`asyncio.run()`** method. The method is a blocking call till the passed-in coroutine finishes. A sample program appears below:

```
async def do_something_important():
    await asyncio.sleep(10)


if __name__ == "__main__":

  asyncio.run(do_something_important())
```

If you are working with Python 3.5, then the **`asyncio.run()`** API isn't available. In that case, we explicitly retrieve the event loop using **`asyncio.new_event_loop()`** and run our desired coroutine using **`run_until_complete()`** defined on the loop object. The code widget below shows how to run a coroutine using the event loop. The code widget runs Python 3.5. so we present the snippet compatible with Python 3.5.

```
import asyncio
import time


async def do_something_important():
    await asyncio.sleep(10)


if __name__ == "__main__":

  start = time.time()

  # Python 3.7+ syntax
  # asyncio.run(do_something_important())

  # Python 3.5 syntax
  loop = asyncio.get_event_loop()
  loop.run_until_complete(do_something_important())

  print("Program ran for {0} seconds".format(time.time() - start))
```

Running Multiple Event Loops

You should never need to start an event loop yourself. Rather, utilize the higher-level APIs to submit coroutines. For instructional purposes, we'll demonstrate launching event loop per thread. The example in the code widget below uses the API **`asyncio.new_event_loop()`** to get a new event loop and then use it to run another coroutine. Examine the output and realize that each spawned thread is running its own event loop.

```
import asyncio, random
from threading import Thread
from threading import current_thread


async def do_something_important(sleep_for):
    print("Is event loop running in thread {0} = {1}\n".format(current_thread().getN
                                                                asyncio.get_event_loop().i

    await asyncio.sleep(sleep_for)


def launch_event_loops():
    # get a new event loop
    loop = asyncio.new_event_loop()

    # set the event loop for the current thread
    asyncio.set_event_loop(loop)

    # run a coroutine on the event loop
    loop.run_until_complete(do_something_important(random.randint(1, 5)))

    # remember to close the loop
    loop.close()


if __name__ == "__main__":
    t1 = Thread(target=launch_event_loops)
    t2 = Thread(target=launch_event_loops)

    t1.start()
    t2.start()

    print("Is event loop running in thread {0} = {1}\n".format(current_thread().getN
                                                                asyncio.get_event_loop().i

    t1.join()
    t2.join()
```

Invoking Callbacks

The event loop can be used to schedule regular functions to be executed as well. The asyncio loop has two APIs: **`asyncio.call_soon()`** and **`asyncio.call_later()`** for this purpose. The **`asyncio.call_soon()`** API schedules the callback for execution in the next iteration of the event loop while **`asyncio.call_later()`** takes in a delay parameter after which to invoke the callback.

In the example in the code widget below, we schedule a coroutine to run on the event loop. The coroutine creates a future and awaits it until the future is resolved by a callback.

```
import asyncio, random, time
from threading import Thread
from threading import current_thread
from asyncio import Future

def resolver(future):
    print("Is loop running in thread {0} = {1}\n".format(current_thread().getName()
                                                         asyncio.get_event_loop().i

    time.sleep(2)
    future.set_result(None)


async def coro():
    future = Future()

    loop = asyncio.get_event_loop()
    loop.call_later(5, resolver, future)

    print("Is loop running in thread {0} = {1}\n".format(current_thread().getName()
                                                         asyncio.get_event_loop().i


    await future
    print("coro exiting")


if __name__ == "__main__":

    loop = asyncio.get_event_loop()
    print("Is loop running in thread {0} = {1}\n".format(current_thread().getName()
                                                         asyncio.get_event_loop().i

    loop.run_until_complete(coro())
    print("main exiting")
```

Blocking the Event Loop

In the above example, if the callback **`resolver()`** took a long time to execute then the loop would be blocked from executing other coroutines. As an example, we'll change the wait time to 5 seconds coroutines. As an example, we'll change the wait time to 5 seconds. Additionally, we'll define a coroutine that will print an "Alive" message every one second. We'll run the this coroutine called **`monitor_coro()`** on the same event loop. The complete code appears in the code widget below.

```
import asyncio, random, time
from threading import Thread
from threading import current_thread
from asyncio import Future
shutdown = False

def resolver(future):
    print("Is loop running in thread {0} = {1}\n".format(current_thread().getName()
                                                         asyncio.get_event_loop().i

    time.sleep(10)
    future.set_result(None)


async def monitor_coro():
    global shutdown

    while shutdown == False:
        print("Alive at {0}".format(time.time()))
        await asyncio.sleep(1)

async def coro():
    global shutdown

    print("coro running")
    future = Future()

    loop = asyncio.get_event_loop()
    monitor_coro_future = asyncio.ensure_future(monitor_coro())
    loop.call_later(5, resolver, future)

    print("Is loop running in thread {0} = {1}\n".format(current_thread().getName()
                                                         asyncio.get_event_loop().i


    await future
    await asyncio.sleep(2)
    shutdown = True

    await monitor_coro_future


if __name__ == "__main__":

    loop = asyncio.get_event_loop()
    print("Is loop running in thread {0} = {1}\n".format(current_thread().getName()
                                                         asyncio.get_event_loop().i

    loop.run_until_complete(coro())
    print("main exiting")
```

If you examine the output from the code widget above, you'll notice that the "Alive" messages from the monitoring coroutine go silent for 10 seconds while the resolver callback makes a synchronous sleep call. This is an example of how an event loop can get blocked if a single method hogs the loop-time. Note that in the example above the execution takes place in the event loop of the main thread.

The fix for the above program is to sleep asynchronously in the **`resolver()`** method. In fact, the **`resolver()`** needs to be converted into a coroutine before we can asynchronously sleep within it. The fix appears in the code widget below. Contrast the output from the above widget with the one below and realize that the monitoring coroutine prints the "Alive" regularly at one-second intervals.

```
import asyncio, random, time
from threading import Thread
from threading import current_thread
from asyncio import Future
shutdown = False

async def resolver(future):
    print("Is loop running in thread {0} = {1}\n".format(current_thread().getName()
                                                         asyncio.get_event_loop().i


    await asyncio.sleep(10)
    future.set_result(None)


async def monitor_coro():
    global shutdown

    while shutdown == False:
        print("Alive at {0}".format(time.time()))
        await asyncio.sleep(1)

async def coro():
    global shutdown

    print("coro running")
    future = Future()

    loop = asyncio.get_event_loop()
    monitor_coro_future = asyncio.ensure_future(monitor_coro())
    resolver_future = asyncio.ensure_future(resolver(future))

    print("Is loop running in thread {0} = {1}\n".format(current_thread().getName()
                                                         asyncio.get_event_loop().i


    await future
    await asyncio.sleep(2)
    shutdown = True

    await monitor_coro_future, resolver_future


if __name__ == "__main__":

    loop = asyncio.get_event_loop()
    print("Is loop running in thread {0} = {1}\n".format(current_thread().getName()
                                                         asyncio.get_event_loop().i

    loop.run_until_complete(coro())
    print("main exiting")
```

Types of Event Loops

There are two types of event loops:

- SelectorEventLoop
- ProactorEventLoop

The SelectorEventLoop is based on the **`selectors`** module and is the default loop on all platforms. The **`selectors`** module contains the **`poll()`** and the **`select()`** APIs that form the secret sauce behind the event loop. ProactorEventLoop, on the other hand, uses Windows' **I/O Completion Ports** and is only supported on Windows. We'll not go into the finer implementation details of the two types but end on a note here that both the type and the associated *policy* with a loop control the behavior of the event loop.

Back
Event Loop

Next
Yield From

Mark as Completed

Report an Issue

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