... continued

Continues the discussion on coroutines.

You would realize that in the previous sections you were essentially dealing with coroutines when sending and receiving data. Similar to a generator, a coroutine executes in response to **`send()`** and **`next()`** calls. Simply creating a coroutine will not execute it. Let's revisit a coroutine example.

Coroutine Example

In Python 2.5 three APIs were added for generator objects, which allowed a generator to act as a coroutine which can collaborate with the caller, yielding and receiving values from the caller. These APIs are **`send()`**, **`throw()`** and **`close()`**. Consider the below example:

Coroutine example

```
def printer():
    item = None
    while True:
        item = yield
        print(str(item))


if __name__ == "__main__":

    coroutine_object = printer()
    next(coroutine_object)

    for i in range(0, 11):
        coroutine_object.send(i)
```

```
def printer():
    item = None
    while True:
        item = yield
        print(str(item))


if __name__ == "__main__":

    coroutine_object = printer()
    next(coroutine_object)

    for i in range(0, 11):
        coroutine_object.send(i)
```

Note the above example doesn't use the syntax for defining native coroutines using **`async def`**. For now we'll stick to coroutines

implemented using generators.

Another important aspect to realize is that the above example fits the definition of a Python coroutine. However, it will not be usable with the asyncio framework without some more work, which we'll discuss when we get to generator-based coroutines. Finally, if you print the class name of the **`coroutine_object`** variable it'll be a generator as shown below:

```
def printer():
    item = None
    while True:
        item = yield
        print(str(item))


if __name__ == "__main__":

    coroutine_object = printer()
    next(coroutine_object)

    print("class name: " + coroutine_object.__class__.__name__)
```

Priming a coroutine

In the above example, we execute **`next()`** in the main script to let the coroutine advance to the first occurrence of **`yield`**. This is also called as **priming** the coroutine. Later on we send the coroutine integers that it prints. The **`send()`** resumes the execution of the coroutine and the snippet **`item = yield`** assigns to item whatever we pass in the send call from the main script. The coroutine would wake up at the yield statement and continue execution until it encounters the next yield statement or it exits.

Think of **`yield`** as a control flow device that can be used to implement cooperative multitasking. Each coroutine yields control to a central scheduler so that other coroutines can be activated.

Difference with threads

Following are the differences between thread and coroutines:

- One of the major benefits of coroutines over threads is that coroutines don’t use as much memory as threads do.
- Coroutines don't require operating system support or invoke system calls.
- Coroutines don't need to worry about synchronizing access to shared data-structures or guarding critical sections. Mutexes, semaphore and other synchronization constructs aren't required.
- Coroutines are concurrent but not parallel.

Back
Coroutine
Next
Event Loop
Mark as Completed
Report an Issue
Ask a Question (https://discuss.educative.io/tag/-continued__asyncio__python-concurrency-for-senior-engineering-interviews)