Coroutine

This lesson discusses the general concept of a coroutine.

Coroutine

As we delve into the all-important topic of coroutines, we'll revisit some of the concepts discussed earlier to setup the context for our discussion around coroutines.

Cooperative Multitasking

Folks familiar with multithreading will know that a system achieves concurrency by context switching threads for CPU time. Threads can be taken off of the CPU if they engage in an I/O call or exhaust their time slice. The OS *prempts* a thread forcing it to give up the use of the CPU. Cooperative Multitasking, on the other hand, takes a different approach in which the running process voluntarily gives up the CPU to other processes. A process may do so when it is logically blocked, say while waiting for user input or when it has initiated a network request and will be idle for a while.

The process scheduler relies on the processes to *cooperate* amongst themselves while using the CPU and is the reason why this paradigm of multitasking is called cooperative multitasking or non-preemptive multitasking.

Coroutines in Python enable cooperative multitasking. The onus of scheduling processes in a reliable manner for CPU time falls on the programmer. A misbehaving process can hog the CPU without giving other processes a chance to execute and cause starvation.

What is a coroutine?

Coroutine isn't a concept specific to Python. In fact, it is a general programming concept also found in other programming languages. A coroutine can be defined as a special function that can give up control to its caller without losing its state. The methods or functions that we are used to, the ones that conclusively return a value and don't remember state between invocations, can be thought of as a specialization of a coroutine, also known as **subroutines**.

Difference with Generators

Generators are essentially iterators though they look like functions. The distinction between generators and coroutines, in general, is that:

- Generators yield back a value to the invoker whereas a coroutine yields control to another coroutine and can resume execution from the point it gives up control.
- A generator can't accept arguments once started whereas a coroutine can.
- Generators are primarily used to simplify writing iterators. They are a type of coroutine and sometimes also called as

semicoroutines.

In case, of Python, generators are used as producers of data and coroutines as consumers data. Before support for native coroutines was introduced in Python 3.5, coroutines were implemented using generators. Objects of both, however, are of type generator. However, since version 3.5, Python makes a distinction between coroutines and generators, as we'll see in the later sections.

History of Coroutines

Before we move forward, it is instructive to look at the history of coroutines in Python as it will help us understand the differences between generators, coroutines, generator-based coroutines, and native coroutines.

- Generators were added to Python 2.2 with PEP-255 (https://www.python.org/dev/peps/pep-0255/), which also talks about the influence of the Icon programming language on the design and implementation of generators.
- Generators were limited to acting as iterators until PEP-342 (https://www.python.org/dev/peps/pep-0342/) was introduced in Python 2.5. Generators could now be closed using `close()`, receive data using `send()` or have an exception thrown at them by the caller using `throw()`. Generators with these new abilities were called **coroutines**, but if you print the classname for such a coroutine object it would show a generator object. The difference was intellectual but not baked into the language. Additionally, PEP-342 (https://www.python.org/dev/peps/pep-0342/) also elevated `yield` from a statement to an expression.

- Later on in Python 3.3, a new syntax `yield from` was introduced via PEP-380 (https://www.python.org/dev/peps/pep-0380/). Briefly, `yield from` makes refactoring generators easier and allows chaining coroutines similar to how unix commands are piped.

- The `asyncio` module was implemented under PEP-3153 (https://www.python.org/dev/peps/pep-3153/) and PEP-3156 (https://www.python.org/dev/peps/pep-3156/) and added to the standard Python 3.4 library. Asyncio module introduced the event loop required for asynchronous programming. Coroutines using `yield from` were defined as generator based coroutines and were compatible with the event loop. As a side note, long before, another module by the name of `asyncore` existed since Python 1.5.2 but had numerous shortcoming which were filled in by `asyncio`.

- Finally PEP-492 (https://www.python.org/dev/peps/pep-0492/) exalted coroutines to a core language feature in Python 3.5. Coroutines can be defined using the keywords `async def`. Any brand new code written should follow the new syntax. With this version of Python, generator and native coroutines were represented by different objects unlike in the previous versions.

You should now be familiar with how coroutines came to be about in Python. As you can tell from the timeline, there are two ways to define coroutines in Python: an older generator based mechanism and the newer shiny `async def` syntax. The two types of Python coroutines are:

- Generator based coroutines

- Native coroutines

We'll examine each one in detail in the following lessons

We'll examine each one in detail in the following lessons.

Back

Next

... continued

... continued

Mark as Completed

Report an Issue

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