


Iterators, generators, coroutines

🕒 Created	@November 1, 2023 4:16 PM
👤 Owner	 Din Lester
🏷️ Tags	python
🌟 Status	Done

Sources:

- <https://www.integralist.co.uk/posts/python-generators/>
- <https://stackoverflow.com/questions/715758/coroutine-vs-continuation-vs-generator>
- <https://medium.com/analytics-vidhya/python-generators-coroutines-async-io-with-examples-28771b586578>
- Марк Лутц, learning python tom 2

Iterators

According to python docs iterator is:

| *An object representing a stream of data.*

As we know iterator object can handle a lot of data without memory leak because it stores only current value, so we can iterate it with `for..in` infinitely.

An 'iterator' is really just a container of some data. This 'container' must have an `__iter__` method which, according to the [protocol documentation](#), should return an iterator object (i.e. something that has the `__next__` method). It's the `__next__` method that moves forward through the relevant collection of data.

So, we can implement 2 methods in class or just 2 class with these methods, so `__iter__` will return instance of another class with `__next__` implemented.

[ADVANCED] As we understood, `iter` returns self with only one copy of state, so when we got `StopIteration` error, we cannot iterate over this object again, we need to create

another one. So, we can create 2 classes (as mentioned above) and handle

Generators

A Generator is a function that returns a 'generator iterator', so it acts similar to how `__iter__` works.

This works with help of yield statement, so when runtime go to yield, it returns control to caller of generator, however the function will remember where it stopped.

Generator expressions

Generator expressions are a high-performance, memory-efficient generalization of list comprehensions and generators.

With help of this, we can write one line expressions like list comprehension, but with memory efficiency.

Nested generators aka yield from

Help us reduce code, so when we have nested iterators we don't need to use for..in multiple times.

Coroutines

Coroutines are computer program components that generalize subroutines for non-preemptive multitasking, by allowing execution to be suspended and resumed.

Because coroutines can pause and resume execution context, they're well suited to concurrent processing, as they enable the program to determine when to 'context switch' from one point of the code to another.

This is why coroutines are commonly used when dealing with concepts such as an event loop (which Python's `asyncio` is built upon).

Generators use the `yield` keyword to return a value at some point in time within a function, but with coroutines the `yield` directive can **also** be used on the right-hand side of an `=` operator to signify it will **accept a value** at that point in time.

So, basically we can pass some data to generator and it will give us controll.