



Generator

This lesson discusses the concept of a generator in Python.

Generator

Functions containing a **yield** statement are compiled as generators. Using a yield expression in a function's body causes that function to be a generator. These functions return an object which supports the iteration protocol methods. The generator object created, automatically receives a **__next__()__** method. Going back to the example from the previous section we can invoke **__next__** directly on the generator object instead of using **next()**:

```
def keep_learning_asynchronous():  
    yield "Educative"  
  
if __name__ == "__main__":  
    gen = keep_learning_asynchronous()  
  
    str = gen.__next__()  
    print(str)
```

You can execute the above code in the code widget below:





```
def keep_learning_asynchronous():  
    yield "Educative"  
  
if __name__ == "__main__":  
    gen = keep_learning_asynchronous()  
  
    str = gen.__next__()  
    print(str)
```



Also note, the snippet **iter(gen) is gen** returns True. Thereby, confirming that a generator function returns a generator object which is an iterator.



```
def keep_learning_asynchronous():  
    yield "Educative"  
  
if __name__ == "__main__":  
    gen = keep_learning_asynchronous()  
  
    print(iter(gen) is gen)
```



Recap

Remember the following about generators:

- Generator functions allow us to procrastinate computing expensive values. We only compute the next value when required. This makes

generators memory and compute efficient. They  refrain  from

saving long sequences in memory or doing all expensive computations upfront.

- Generators when suspended retain the code location, which is the last yield statement executed, and their entire local scope. This allows them to resume execution from where they left off.
- Generator objects are nothing more than iterators.
- Remember to make a distinction between a **generator function** and the associated **generator object** which are often used interchangeably. A generator function when invoked returns a generator object and **next()** is invoked on the generator object to run the code within the generator function.

Example

We are familiar with how functions work, which may also be called methods or subroutines. Generators are methods too but they can be paused and resumed later. The rationale for writing a generator function is somewhat the same for lazy instantiation. We don't want to waste time and resources if we won't need a value. To make the concept concrete, consider a function that returns natural numbers (1, 2, 3) and let's assume that finding the next natural number is a very expensive operation. Also, we don't know how many natural numbers we'll ask for from our method **natural_nums()** but we do know that in the worst case it would not be more than 100. One approach is that the method **natural_nums()** computes all the first 100 natural numbers that **may** be required and return them as needed. However, with this approach, we incur the cost of calculating all the first 100 hundred natural numbers and maybe we use only the first 5 in a particular run.

The alternative approach is to use a generator function. The trick is two part:



- Instead of **return** use **yield** to return a value from a generator function.
- We pass the return value of a generator function to the **next()** method to get the actual value yielded or returned from the generator method.

Let's dive into an example:

Generator function example

```
def natural_nums():  
    i = 0  
    while True:  
        i += 1  
        yield i
```

And the usage would be as follows:

Generator function example

```
iter = natural_nums()  
  
next(iter)  
next(iter)
```

The above example can be run in the code widget below:





```
def natural_nums():
    i = 0
    while True:
        i += 1
        yield i

if __name__ == "__main__":

    # get the generator object
    iter = natural_nums()

    # print the next value from the generator function
    val = next(iter)
    print(val)

    # print another one
    val = next(iter)
    print(val)
```



The **yield** keyword lets the interpreter know to treat the function like an iterator. In fact **yield** syntax is just syntactic sugar that tells the Python interpreter to generate an iterable object from the function where it appears. Also it pauses the function and saves the local state so that the method can be resumed from where it left off.

Consider the example below. Note that the print statement isn't executed until **next()** is invoked on the **iter** variable. If you run the following snippet, nothing happens because we didn't invoke next on the generator returned.





```
def natural_nums():
    print("inside natural_nums function")
    i = 0
    while True:
        i += 1
        yield i

if __name__ == "__main__":
    iter = natural_nums()
```



To sum up, a snippet like **yield item** produces a value that is received by the caller of **next()**. At the same time, **yield** also cedes control, suspending the execution of the generator so that the caller may proceed until it's ready to consume another value by invoking **next()** again.

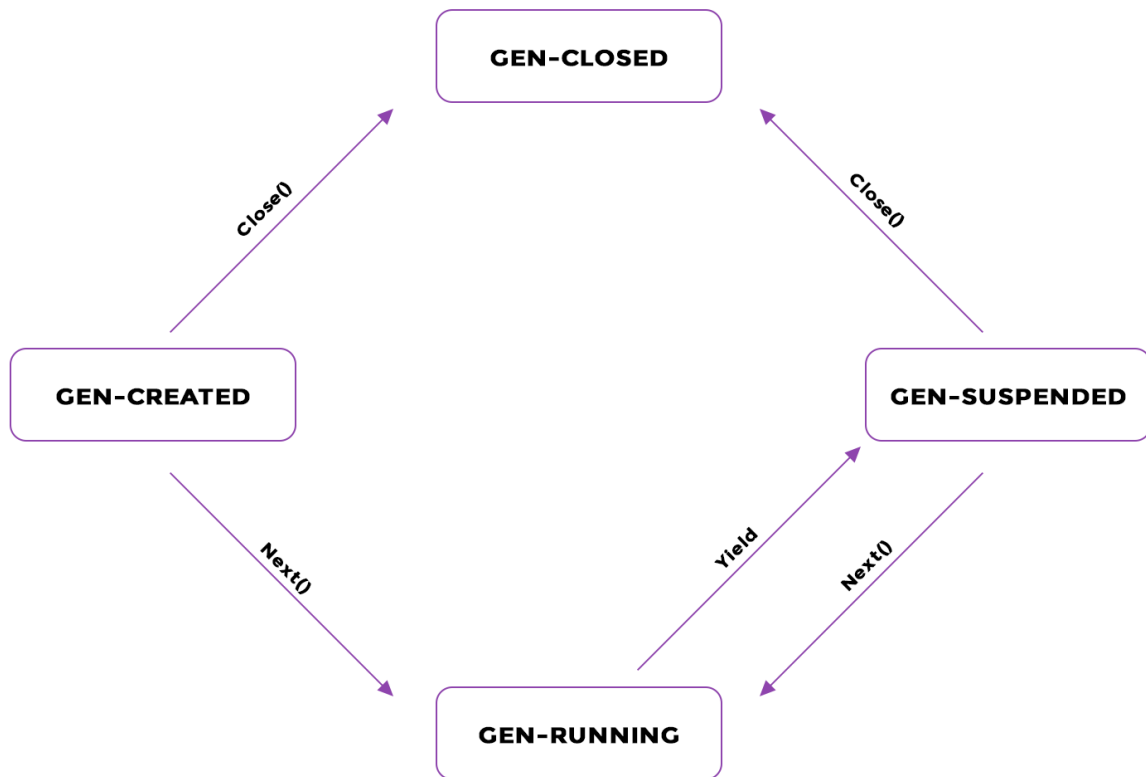
States of a generator

A generator goes through the following states:

- **GEN_CREATED** when a generator object has been returned for the first time from a generator function and iteration hasn't started.
- **GEN_RUNNING** when **next** has been invoked on the generator object and is being executed by the python interpreter.
- **GEN_SUSPENDED** when a generator is suspended at a **yield**
- **GEN_CLOSED** when a generator has completed execution or has been closed. We'll learn about closing generators later.



GENERATOR LIFECYCLE



The below runnable snippet displays three of the four states of a generator. We can use `inspect.getgeneratorstate()` method to get the state of the generator.





```
import inspect

def natural_nums():
    i = 0
    while True:
        i += 1
        yield i

if __name__ == "__main__":

    iter = natural_nums()
    print("generator state " + inspect.getgeneratorstate(iter))
    next(iter)
    print("generator state " + inspect.getgeneratorstate(iter))
    iter.close()
    print("generator state " + inspect.getgeneratorstate(iter))
```



It's a little tricky to demonstrate the state of a generator while it is executing using what we have learned so far. The example below uses a thread and event loop to show the generator in a running state. This example is for demonstration purposes and it's ok to hold off on trying to understand completely how it works. You can return to this example after we cover event loops when it would make more sense.





```
import asyncio
import inspect
import time
from threading import Thread

@asyncio.coroutine
def sleeping_generator():
    time.sleep(3)
    yield None

def run_generator(gen):
    loop = asyncio.new_event_loop()
    asyncio.set_event_loop(loop)
    next(gen)

if __name__ == "__main__":
    gen = sleeping_generator()
    Thread(target=run_generator, args=(gen,)).start()

    i = 0
    while i != 5:
        print("generator state " + inspect.getgeneratorstate(gen))
        time.sleep(0.1)
        i+=1
```



Reaching the End

In the previous lesson, we saw that a generator throws a **StopIteration** exception when it has yielded all its values. Invoking **next()** on a generator object that has been exhausted throws the **StopIteration** exception which wraps the return value from the generator if there is one. Recognize that if we iterate over a generator object using a for-loop the exception doesn't show up. The reason is that the for-loop catches the **StopIteration** and ends the for loop. The example in the code widget demonstrates our discussion.

widget demonstrates our discussion.



```
def get_item():  
  
    yield 3  
    yield 5  
    yield 7  
    # raise StopIteration at this point
```

```
def get_item():  
  
    yield 3  
    yield 5  
    yield 7  
    # raise StopIteration  
  
if __name__ == "__main__":  
  
    # no StopIteration is thrown  
    for num in get_item():  
        print(num)
```



Methods on Generator Objects

A generator object exposes different methods that can be invoked to manipulate the generator. These are:

- `throw()`
- `send()`
- `close()`

Using `close()`

Using `close()`



We'll discuss `throw()` and `send()` in the following section. The `close()` method is invoked by the interpreter when the generator object is garbage collected. It can also be invoked manually and doing so would make the generator unavailable for iteration.

Consider the example below, which throws a **`StopIteration`** exception after we invoke `close()` on a generator object and then attempt to iterate on it.

```
def get_natural_nums():
    i = 0
    while True:
        yield i
        i += 1

if __name__ == "__main__":
    gen = get_natural_nums()

    # Close the generator
    gen.close()

    # Attempt to iterate
    next(gen)
```

```
def get_natural_nums():
    i = 0
    while True:
        yield i
        i += 1

if __name__ == "__main__":
    gen = get_natural_nums()

    # Close the generator
    gen.close()
```



```
# Attempt to iterate  
next(gen)
```



The intent of the `close()` method is to give the generator a chance to clean-up before it exits. A generator function can catch the **GeneratorException** and perform clean-up actions. Below is an example:

```
def get_item():  
    try:  
        yield 5  
  
    except GeneratorExit:  
        print("GeneratorExit exception raised")
```

Run the above example in the code widget below and examine the output.

```
def get_item():  
    try:  
        yield 5  
  
    except GeneratorExit:  
        print("GeneratorExit exception raised")  
  
if __name__ == "__main__":  
    gen = get_item()  
  
    print(next(gen))  
    print("Main exiting")
```





Notice the exception raised statement appears in the output after the main exiting statement. The Python interpreter garbage collects the **gen** variable, which is a generator, and invokes **close()** on it, which is why the exception raised statement is printed after the main exiting statement. Contrast the above output with the one below, where we explicitly invoke the **close()** method on the **gen** object.

```
def get_item():
    try:
        yield 5

    except GeneratorExit:
        print("GeneratorExit exception raised")

if __name__ == "__main__":
    gen = get_item()

    print(next(gen))
    gen.close()
    print("main exiting")
```



In the output above, you'll notice that the exception raised statement appears before the main thread exits. Let's turn our attention to the other two methods exposed on a generator object in the next section.

[< Back](#)[Next >](#)

Yield

Send



Mark as Completed



Report an
Issue



Ask a Question

(https://discuss.educative.io/tag/generator__asyncio__python-concurrency-for-senior-engineering-interviews)