# Python Generators

**Summary**: in this tutorial, you'll learn about Python generators and how to use generators to create iterators

## Introduction to Python generators

Typically, Python executes a regular function from top to bottom based on the run-to-completion model.

It means that Python cannot pause a regular function midway and then resumes the function after that. For example:

```
def greeting():
    print('Hi!')
    print('How are you?')
    print('Are you there?')
```

When Python executes the `greeting()` function, it executes the code line by line from top to bottom.

Also, Python cannot pause the function at the following line:

```
print('How are you?')
```

… and jumps to another code and resumes the execution from that line.

To pause a function midway and resume from where the function was paused, you use the `yield` statement.

When a function contains at least one `yield` statement, it's a **generator function**.

By definition, a generator is a function that contains at least one yield statement.

When you call a generator function, it returns a new generator object. However, it doesn't start the function.

Generator objects (or generators) implement the iterator protocol. In fact, generators are lazy iterators. Therefore, to execute a generator function, you call the `next()` built-in function on it.

## A simple Python generator example

See the following example:

```
def greeting():
    print('Hi!')
    yield 1
    print('How are you?')
    yield 2
    print('Are you there?')
    yield 3
```

Since the `greeting()` function contains the `yield` statements, it's a generator function.

The `yield` statement is like a `return` statement in a function. However, there's a big difference.

When Python encounters the `yield` statement, it returns the value specified in the `yield` . In addition, it pauses the execution of the function.

If you "call" the same function again, Python will resume from where the previous `yield` statement was encountered.

When you call a generator function, it returns a generator object. For example:

```
messenger = greeting()
```

The `messenger` is a generator object, which is also an iterator.

To actually execute the body of the `greeting()` function, you need to use the `next()` built-in function:

```
result = next(messenger)
print(result)
```

When you the `greeting()` function executes, it shows the first message and returns 1:

```
Hi!
1
```

Also, it’s paused right at the first `yield` statement. If you “call” the `greeting()` function again, it’ll resume the execution from the last `yield` statement:

```
result = next(messenger)
print(result)
```

Output:

```
How are you?
2
```

And you can call it again:

```
result = next(messenger)
print(result)
```

Output:

```
Are you there?
3
```

However, if you execute the generator once more time, it’ll raise the StopIteration exception because it’s an iterator:

```
next(messenger)
```

Error:

```
StopIteration
```

## Using Python generators to create iterators

The following example defines an iterator that returns a square number of an integer.

```python
class Squares:
    def __init__(self, length):
        self.length = length
        self.current = 0

    def __iter__(self):
        return self

    def __next__(self):
        result = self.current ** 2

        self.current += 1

        if self.current > self.length:
            raise StopIteration

        return result
```

And you can use the `Squares` iterator to generate the square numbers of the first 5 integers from 0 to 5:

```python
length = 5
square = Squares(length)
for s in square:
    print(s)
```

This code works as we expected. But it has one issue that there's a lot of boilerplate.

And as you might guess, you use a generator function to build that iterator.

The following rewrites the `Squares` iterator as a generator function:

```python
def squares(length):
    for n in range(length):
        yield n ** 2
```

As you can see, it's much shorter and more expressive. The usage of the `squares` generator function is similar to the iterator above:

```python
length = 5
square = squares(length)
for s in square:
    print(s)
```

## Summary

- Python generators are functions that contain at least one yield statement.
- A generator function returns a generator object.
- A generator object is an iterator. Therefore, it becomes exhausted once there's no remaining item to return.