Generators

A normal Python function will always return one value, whether it be a list, an integer or some other object. But what if you wanted to be able to call a function and have it yield a series of values? That is where generators come in. A generator works by "saving" where it last left off (or yielding) and giving the calling function a value. So instead of returning the execution to the caller, it just gives temporary control back. To do this magic, a generator function requires Python's **yield** statement.

*Side-note: In other languages, a generator might be called a coroutine.**

Let's take a moment and create a simple generator!

```
def doubler_generator():
    number = 2
    while True:
        yield number
        number *= number

doubler = doubler_generator()
print (next(doubler))
#2

print (next(doubler))
#4

print (next(doubler))
#16

print (type(doubler))
#<class 'generator'>
```

This particular generator will basically create an infinite sequence. You can call **next** on it all day long and it will never run out of values to yield. Because you can iterate over a generator, a generator is considered to be a type of iterator, but no one really refers to them as such. But underneath the covers, the generator is also defining the **__next__** method that we looked at in our previous section, which is why the **next** keyword we just used worked.

Let's look at another example that only yields 3 items instead of an infinite sequence!

```
def silly_generator():
    yield "Python"
    yield "Rocks"
    yield "So do you!"
gen = silly_generator()
print (next(gen))
#'Python'

print (next(gen))
#'Rocks'

print (next(gen))
#'So do you!'

print (next(gen))
#Traceback (most recent call last):
#  File "/usercode/__ed_file.py", line 15, in <module>
# print (next(gen))
#StopIteration:
```

Here we have a generator that uses the **yield** statement 3 times. In each instance, it yields a different string. You can think of **yield** as the **return** statement for a generator. Whenever you call yield, the function stops and saves its state. Then it yields the value out, which is why you see something getting printed out to the terminal in the example above. If we'd had variables in our function, those variables would be saved too.

When you see **StopIteration**, you know that you have exhausted the iterator. This means that it ran out of items. This is normal behavior in all iterators as you saw the same thing happen in the iterators section.

Anyway when we call **next** again, the generator begins where it left off and yields whatever the next value is or we finish the function and the generator stops. On the other hand, if you never call next again, then the state will eventually go away.

Let's reinstantiate the generator and try looping over it!

```
gen = silly_generator()
for item in gen:
    print(item)

#Python
#Rocks
#So do you!
```

The reason we create a new instance of the generator is that if we tried looping over it, nothing would be yielded. This is because we already ran through all the values in that particular instance of the generator. So in this example, we create the new instance, loop over it and print out the values that are yielded. The **for** loop once again handles the **StopIteration** exception for us and just breaks out of the loop when the generator is exhausted.

One of the biggest benefits to a generator is that it can iterate over large data sets and return them one piece at a time. This is what happens when we open a file and return it line-by-line:

```
with open('file.txt') as fobj:
    for line in fobj:
        #process the line
```

Python basically turns the file object into a generator when we iterate over it in this manner. This allows us to process files that are too large to load into memory. You will find generators useful for any large data set that you need to work with in chunks or when you need to generate a large data set that would otherwise fill up your all your computer's memory.

Wrapping Up

At this point you should now understand what an iterator is and how to use one. You should also know the difference between an iterable and an iterator. Finally, we learned what a generator is and why you might want to use one. For example, a generator is great for memory efficient data processing. In the next chapter, we will dig into an iterator library that is included with Python that's called **itertools**.