

Iterators and Generators|

In this section of the course we will be learning about the difference between iteration and generation in Python and how to **construct** our own **Generators with the *yield* statement**. **Generators allow us to generate as we go along, instead of holding everything in memory.**

We've touch on this topic in the past when discussing the `range()` function in Python 2 and the similar `xrange()`, with the difference being the `xrange()` was a generator.

Lets explore a little deep. We've learned how to create functions with **`def`** and the **`return`** statement. **Generator functions allow us to write a function that can send back a value and then later resume to pick up where it left off.** This type of function is a generator in Python, allowing us to generate a sequence of values over time. The main difference in syntax will be the use of a **`yield` statement**.

In most aspects, a generator function will appear very similar to a normal function. The main difference is when a generator function is compiled they become an object that support an iteration protocol. That means when they are called in your code they don't actually return a value and then exit, the generator functions will automatically suspend and resume their execution and state around the last point of value generation. The main advantage here is that instead of having to compute an entire series of values upfront and the generator functions can be suspended, this feature is known as *state suspension*.

To start getting a better understanding of generators, lets go ahead and see how we can create some.

```
In [6]: # Generator function for the cube of numbers (power of 3)
def gencubes(n):
    for num in range(n):
        yield num**3
```

```
In [10]: for x in gencubes(10):
        print x
```

```
0
1
8
27
64
125
216
343
512
729
```

Great! Now since we have a generator function we don't have to keep track of every single cube we created.

Generators are best for calculating large sets of results (particularly in calculations that involve loops themselves) in cases where we don't want to allocate the memory for all of the results at the same time.

As we've noted in previous lectures (such as `range()`) many Standard Library functions that return lists in Python 2 have been modified to return generators in Python 3 because generators.

Lets create another example generator which calculates [fibonacci](https://en.wikipedia.org/wiki/Fibonacci_number) (https://en.wikipedia.org/wiki/Fibonacci_number) numbers:

```
In [1]: def genfibon(n):  
        ...  
        Generate a fibonnaci sequeunce up to n  
        ...  
        a = 1  
        b = 1  
        for i in range(n):  
            yield a  
            a,b = b,a+b
```

```
In [2]: for num in genfibon(10):  
        print num
```

```
1  
1  
2  
3  
5  
8  
13  
21  
34  
55
```

What is this was a normal function, what would it look like?

```
In [3]: def fibon(n):  
        a = 1  
        b = 1  
        output = []  
  
        for i in range(n):  
            output.append(a)  
            a,b = b,a+b  
  
        return output
```

```
In [4]: fibon(10)
```

```
Out[4]: [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

Notice that if we call some huge value of n (like 100000) the second function will have to keep

track of every single result, when in our case we actually only care about the previous result to generate the next one!

next() and **iter()** built-in functions

A key to fully understanding generators is the `next()` function and the `iter()` function.

The `next` function allows us to access the next element in a sequence. Lets check it out:

```
In [20]: def simple_gen():  
         for x in range(3):  
             yield x
```

```
In [21]: # Assign simple_gen  
         g = simple_gen()
```

```
In [22]: print next(g)
```

0

```
In [23]: print next(g)
```

1

```
In [24]: print next(g)
```

2

```
In [25]: print next(g)
```

```
-----  
StopIteration                                Traceback (most recent call last)  
<ipython-input-25-d013a5691c47> in <module>()  
----> 1 print next(g)
```

StopIteration:

After yielding all the values `next()` caused a **StopIteration error**. What this error informs us of is that all the values have been yielded.

You might be wondering that why don't we get this error while using a for loop? The for loop automatically catches this error and stops calling `next`.

Lets go ahead and check out how to use `iter()`. You remember that strings are iterables:

```
In [26]: s = 'hello'

#Iterate over string
for let in s:
    print let
```

```
h
e
l
l
o
```

But that doesn't mean the string itself is an *iterator*! We can check this with the `next()` function:

```
In [27]: next(s)
```

```
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-27-bc0566bea448> in <module>()
----> 1 next(s)
```

```
TypeError: str object is not an iterator
```

Interesting, this means that a string object supports iteration, but we can not directly iterate over it as we could with a generator function. The `iter()` function allows us to do just that!

```
In [28]: s_iter = iter(s)
```

```
In [29]: next(s_iter)
```

```
Out[29]: 'h'
```

```
In [30]: next(s_iter)
```

```
Out[30]: 'e'
```

Great! Now you know how to convert objects that are iterable into iterators themselves!

The main takeaway from this lecture is that using the `yield` keyword at a function will cause the function to become a generator. This change can save you a lot of memory for large use cases. For more information on generators check out:

[Stack Overflow Answer \(http://stackoverflow.com/questions/1756096/understanding-generators-in-python\)](http://stackoverflow.com/questions/1756096/understanding-generators-in-python)

[Another StackOverflow Answer \(http://stackoverflow.com/questions/231767/what-does-the-yield-keyword-do-in-python\)](http://stackoverflow.com/questions/231767/what-does-the-yield-keyword-do-in-python)

```
In [ ]:
```

