In this section of the course we will be learning 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 touched on this topic in the past when discussing certain built-in Python functions like **range()**, **map()** and **filter()**.

Let's explore a little deeper. 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 supports an iteration protocol. That means when they are called in your code they don't actually return a value and then exit. Instead, 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 up front, the generator computes one value and then suspends its activity awaiting the next instruction. This feature is known as *state suspension*.

In many cases it is not appropriate (or possible) to obtain all the data to be processed up front (for performance reasons, for memory reasons etc.). Instead lazily creating the data to be iterated over based on some underlying dataset, may be more appropriate. Generators are a special function that can be used to generate a sequence of values to be iterated over on demand (that is when the values are needed) rather than produced up front. The only thing that makes a generator a generator function is the use of the yield keyword (which was introduced in Python 2.3). The yield keyword can only be used inside a function or a method. Upon its execution the function is suspended, and the value of the yield statement is returned as the current cycle value. If this is used with a for loop, then the loop runs once for this value. Execution of the generator function is then resumed after the loop has cycled once and the next cycle value is obtained. The generator function will keep supplying values until it returns (which means that an infinite sequence of values can be generated).

**Defining a Generator Function**

A very simple example of a generator function is given below. This function is called the gen\_numbers() function:

**def** gen\_numbers():

**yield** 1 *#return 1*

**yield** 2

**yield** 3

**for** i **in** gen\_numbers():

print(i)

​

1

2

3

This is a generator function as it has at least one yield statement (in fact it has three). Each time the gen\_numbers() function is called within a for statement it will return one of the values associated with a yield statement; in this case the value 1, then the value 2 and finally the value 3 before it returns (terminates).

*# Using a Generator Function in a for Loop*

*# We can use the gen\_numbers() function with a for statement as shown below:*

**for** i **in** gen\_numbers():

print(i)

It is common for the body of a generator to have some form of loop itself. This loop is typically used to generate the values that will be yielded. However, as is shown above that is not necessary and here a yield statement is repeated three times. Note that gen\_numbers() is a function but it is a special function as it returns a generator object. This is a generator function returns a generator object which wraps up the generation of the values required but this is hidden from the developer.

When Do the Yield Statements Execute?

It is interesting to consider what happens within the generator function; it is actually

suspended each time a yield statement supplies a value and is only resumed when

the next request for a value is received. This can be seen by adding some additional

print statements to the gen\_numbers() function:

**def** gen\_numbers2():

print('Start')

**yield** 1

print('Continue')

**yield** 2

print('Final')

**yield** 3

print('End')

**for** i **in** gen\_numbers2():

print("Enter")

print(i)

​

*# Thus the generator executes the yield statements on an as needed basis and not*

*# all at once.*

Start

Enter

1

Continue

Enter

2

Final

Enter

3

End

*# Generator function for the cube of numbers (power of 3)*

**def** gencubes(n):

**for** num **in** range(n):

**yield** num**\*\***3

​

​

*# gencubes(10)*

**for** x **in** gencubes(10):

print(x)

0

1

8

27

64

125

216

343

512

729

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.

Let's create another example generator which calculates [fibonacci](https://en.wikipedia.org/wiki/Fibonacci_number" \t "_blank) numbers:

**def** genfibon(n):

"""

Generate a fibonnaci sequence up to n

"""

a **=** 1

b **=** 1

**for** i **in** range(n):

**yield** a

a,b **=** b,a**+**b

**for** num **in** genfibon(10):

print(num)

1

1

2

3

5

8

13

21

34

55

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

**def** fibon(n):

a **=** 1

b **=** 1

output **=** []

**for** i **in** range(n):

output.append(a)

a,b **=** b,a**+**b

**return** output

fibon(10)

Out[6]:

[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!

*# An Even Number Generator*

*# We could have used a generator to produce a set of even numbers up to a specific limit,*

*# as we did earlier with the Evens class, but without the need to create a class (and*

*# implement the two special methods \_\_iter\_\_() and \_\_next\_\_()). For example:*

**def** evens\_up\_to(limit):

value **=** 0

**while** value **<=** limit:

**yield** value

value **+=** 2

**for** i **in** evens\_up\_to(6):

print(i, end**=**', ')

​

​

0, 2, 4, 6,

​

This illustrates the potential benefit of a generator over an iterator; the evens\_up\_to() function is a lot simpler and concise then the Evens iterable class.

**Nesting Generator Functions**

You can even nest generator functions as each call to the generator function is encapsulated in its own generator object which captures all the state information needed by that generator invocation. For example:

**for** i **in** evens\_up\_to(4):

print('i:', i)

**for** j **in** evens\_up\_to(6):

print('j:', j, end**=**', ')

print('')

i: 0

j: 0, j: 2, j: 4, j: 6,

i: 2

j: 0, j: 2, j: 4, j: 6,

i: 4

j: 0, j: 2, j: 4, j: 6,

As you can see from this the loop variable i is bound to the values produced by the first call to evens\_up\_to() (which produces a sequence up to 4) while the j loop variable is bound to the values produced by the second call to evens\_up\_to() (which produces a sequence of values up to 6).

**Using Generators Outside a for Loop**

You do not need a for loop to work with a generator function; the generator object actually returned by the generator function supports the next() function. This function takes a generator object (returned from the generator function) and returns the next value in sequence.

Subsequent calls to next(evens) return no value; if required the generator can throw an error/exception.

​

evens **=** evens\_up\_to(4)

print(next(evens), end**=**', ')

print(next(evens), end**=**', ')

print(next(evens))

0, 2, 4

**Iterators**

An iterator is an object that will return a sequence of values. Iterators may be finite in length or infinite (although many container-oriented iterators provide a fixed set of values). The iterator protocol specifies the **next**() method. This method is expected to return the next item in the sequence to return or to raise the StopIteration exception. This is used to indicate that the iterator has finished supplying values

**The Iteration Related Methods**

To summarise then we have

• **iter**() from the Iterable protocol which is used to return the iterator object,

• **next**() from the Iterator protocol which is used to obtain the next value in a sequence of values.

Any data type can be both an Iterable and an Iterator; but that is not required. An Iterable could return a different object that will be used to implement the iterator or it can return itself as the iterator—it’s the designers choice.

The Itertools Module The itertools module provides a number of useful functions that return iterators constructed in various ways. It can be used to provide an iterator over a selection of values from a data type that is iterable; it can be used to combine iterables together etc

**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:

**def** simple\_gen():

**for** x **in** range(3):

**yield** x

*# Assign simple\_gen*

g **=** simple\_gen()

​

*# return value of generator function is iteratable object*

g

​

**for** i **in** simple\_gen():

print(i)

​

​

Out[29]:

<generator object mygen at 0x08527DF0>

print(next(g))

0

print(next(g))

1

print(next(g))

2

print(next(g))

**---------------------------------------------------------------------------**

**StopIteration** Traceback (most recent call last)

**<ipython-input-12-1dfb29d6357e>** 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? A for loop automatically catches this error and stops calling next().

Let's go ahead and check out how to use iter(). You remember that strings are iterables:

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:

next(s)

**---------------------------------------------------------------------------**

**TypeError** Traceback (most recent call last)

**<ipython-input-14-61c30b5fe1d5>** 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!

s\_iter **=** iter(s)

next(s\_iter)

Out[16]:

'h'

next(s\_iter)

Out[17]:

'e'

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

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)

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

*# Generators*

*# Generators are functions that return a sequence of values .it will be written like other functions but will be using yield st*

​

**def** mygen(x,y):

**while**(x **<=**y):

**yield** x

x**+=**1

g **=** mygen(5,10)

g

Out[3]:

<generator object mygen at 0x08527DF0>

**for** i **in** g:

print(i,end **=**' ')

5 6 7 8 9 10

*# if we wanted to retrive element by element from generator object, we can use next() function as*

**def** mygen():

**yield** 'A'

**yield** 'b'

**yield** 'c'

​

g**=** mygen() *# calling generator function and get generator object g*

​

*# display all the characters in the generator*

print(next(g))

print(next(g))

print(next(g))

​

*#print(next(g)) will give error*