CHAPTER

THREE

GENERATORS

First lets understand iterators. According to Wikipedia, an iterator is an object that enables a programmer to traverse a container, particularly lists. However, an iterator performs traversal and gives access to data elements in a container, but does not perform iteration. You might be confused so lets take it a bit slow. There are three parts namely:

- Iterable
- Iterator
- Iteration

All of these parts are linked to each other. We will discuss them one by one and later talk about generators.

3.1 Iterable

An `iterable` is any object in Python which has an `__iter__` or a `__getitem__` method defined which returns an **iterator** or can take indexes (Both of these dunder methods are fully explained in a previous chapter). In short an `iterable` is any object which can provide us with an **iterator**. So what is an **iterator**?

3.2 Iterator

An iterator is any object in Python which has a `next` (Python2) or `__next__` method defined. That's it. That's an iterator. Now let's understand **iteration**.

3.3 Iteration

In simple words it is the process of taking an item from something e.g a list. When we use a loop to loop over something it is called iteration. It is the name given to the process itself. Now as we have a basic understanding of these terms let's understand **generators**.

3.4 Generators

Generators are iterators, but you can only iterate over them once. It's because they do not store all the values in memory, they generate the values on the fly. You use them by iterating over them, either with a 'for' loop or by passing them to any function or construct that iterates. Most of the time `generators` are implemented as functions. However, they do not `return` a value, they `yield` it. Here is a simple example of a `generator` function:

```
def generator_function():
    for i in range(10):
        yield i

for item in generator_function():
    print(item)

# Output: 0
# 1
# 2
# 3
# 4
# 5
# 6
# 7
# 8
# 9
```

It is not really useful in this case. Generators are best for calculating large sets of results (particularly calculations involving loops themselves) where you don't want to allocate the memory for all results at the same time. Python modified a lot of Python 2 functions which returned `lists` to return `generators` in Python 3. It is because `generators` are not resource intensive. Here is an example which calculates fibonacci numbers:

```
# generator version
def fibon(n):
    a = b = 1
    for i in xrange(n):
        yield a
        a, b = b, a + b
```

Now we can use it like this:

```
for x in fibon(1000000):
    print(x)
```

This way we would not have to worry about it using a lot of resources. However, if we would have implemented it like this:

```
def fibon(n):
    a = b = 1
    result = []
    for i in xrange(n):
        result.append(a)
        a, b = b, a + b
    return result
```

It would have used up all our resources while calculating a large input. We have discussed that we can iterate over `generators` only once but we haven't tested it. Before testing it you need to know about one more built-in function of Python, `next()`. It allows us to access the next element of a sequence. So let's test out our understanding:

```
def generator_function():
    for i in range(3):
        yield i

gen = generator_function()
print(next(gen))
# Output: 0
print(next(gen))
# Output: 1
print(next(gen))
# Output: 2
print(next(gen))
# Output: Traceback (most recent call last):
#           File "<stdin>", line 1, in <module>
#         StopIteration
```

As we can see that after yielding all the values `next()` caused a `StopIteration` error. Basically this error informs us that all the values have been yielded. You might be wondering that why don't we get this error while using a `for` loop? Well the answer is simple. The `for` loop automatically catches this error and stops calling `next`. Do you know that a few built-in data types in Python also support iteration? Let's check it out:

```
my_string = "Yasoob"
next(my_string)
# Output: Traceback (most recent call last):
#      File "<stdin>", line 1, in <module>
#    TypeError: str object is not an iterator
```

Well that's not what we expected. The error says that `str` is not an iterator. Well it is right! It is an iterable but not an iterator. This means that it supports iteration but we can not directly iterate over it. How can we then iterate over it? It's time to learn about one more built-in function, `iter`. It returns an `iterator` object from an iterable. Here is how we can use it:

```
my_string = "Yasoob"
my_iter = iter(my_string)
next(my_iter)
# Output: 'Y'
```

Now that is much better. I am sure that you loved learning about generators. Do bear it in mind that you can fully grasp this concept only when you use it. Make sure that you follow this pattern and use `generators` whenever they make sense to you. You wont be disappointed!