Data streaming (aka lazy evaluation)

Iterables, Iterators and Iteration in Python

In this lesson we are going learn about Iterators. You often here these confusing words, Iterable, Iterator, Iteration. Let's break down and understand them

Iteration:

*It's a **repetition of a process**, for example when you use for-loop, you get an item from something, do something with that, right ?*

```
for x in something:
    do_something()
```

*Getting an item and doing something is a process, which is repeated, until all the items are exhausted. This is called **iteration**.*

Iterable:

Any object, which allows us to iterate through. In other words, a Python object which supports Iteration.

*In Python, an iterable object, has an **__iter__( )** method implemented, By this method iterable object is capable of creating an iterator.*

Built-in iterables:

Python is full of iterables, many objects that are built into Python or defined in modules are designed to be iterable. For example, open files in Python are iterable.

How to check an object is iterable? :

*If you check the names of any iterable with dir ( ), you will find **__iter__** dunder method. If the object has this method, that means its an iterable*

```
empty_list = [ ]

dir(empty_list)

#The dir() function can be used to see the list of available methods on the iterable object

Output:

#Other names are not shown.

[... '__iter__', … ]
```

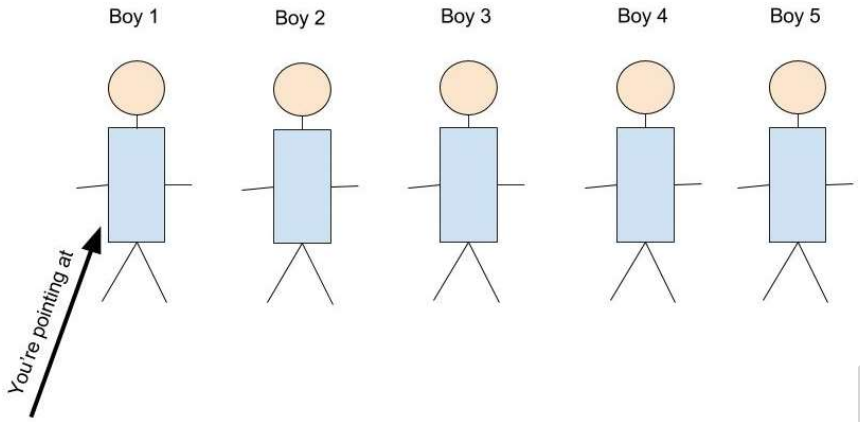
Iterator:

Iterator is an object, which has ability to iterate through each element of an iterable.

***Iterable creates iterator**, that iterator is used to iterate through iterable.*

In simple words, iterator is Python object, which performs iteration over iterable.

Suppose, A group of 5 boys are standing in a line. Your are pointing at the first boy and ask him about his name. Then, he replied. After that, you ask the next boy and so on.



*In this case, you are the **Iterator**!!!! Obviously, the group of boys is the **iterable** element*

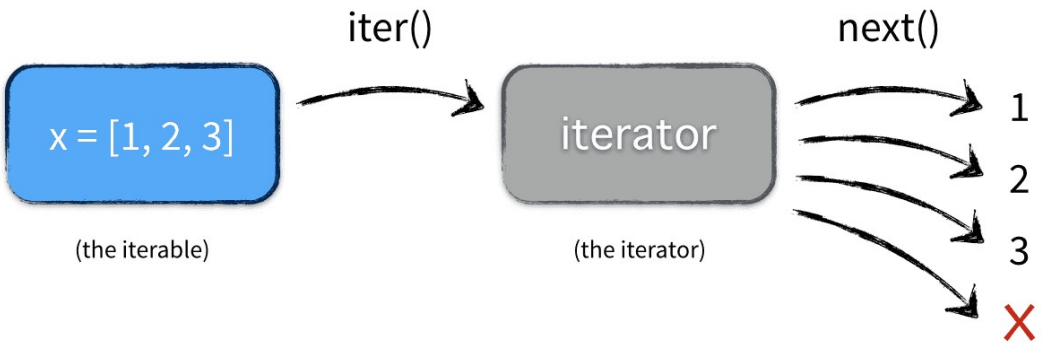
Iteration Protocol:

Iteration protocol is about how iterables actually work internally.It's just a fancy term.

*Iteration protocol is combination of **next( ) and iter( )** functions implementation.*

- ✓ *iter( ) creates an iterator. Iter( ) is called on the iterable to retrieve an iterator.*
- ✓ *next( ) is called on the iterator to sequentially retrieve elements from iterable.*
- ✓ *When no more elements are available, next( ) will raise StopIteration.*

To be more specific, we know a list (e.g, [1,2,3]) is iterable. iter(iterable) produces an iterater, and from this we can get stream of values



Iterator Syntax:

To use iterators, you can use the methods as defined above __iter__ and __next__ methods.

You can create an iterable object as per the below instruction:

```
iterable_object = iter ( my_object_to_iterate_through )
```

Once, you get a hold of the iterator, then use the following statement to cycle through it.

```
next(iterable_object)
```

Iterator Examples:

Each of the objects in the following example is an iterable and returns some type of iterator when passed to iter( ):

```
>>> iter('Sarah')                          # String
    <str_iterator object at 0x100ba8910>
>>> iter(['Sarah', 'Roark'])               # List
    <list_iterator object at 0x100ba8a10>
>>> iter(('Sarah 'Roark'))                 # Tuple
    <tuple_iterator object at 0x100ba8ad0>
>>> iter({'Sarah': 26, 'Roark': 25})       # Dict
    <dict_keyiterator object at 0x1008f81d0>
```

These object types, on the other hand, aren't iterable:

```
>>> iter(30)                                # Integer
    TypeError: 'int' object is not iterable
>>> iter(22.6)                              # Float
    TypeError: 'float' object is not iterable
```

All the data types we have encountered so far that are collection or container types, are iterable.

These include the string, list, tuple, dict, and set types

Once, you get a hold of the iterator The built-in function next( ) is used to obtain the next value from in iterator.

Here is an example using the same list as above:

```
>>> l = ['Sarah', 'Roark']
>>> itr = iter(l)
>>> itr
   <list_iterator object at 0x100ba8950>
>>> next(itr)
   'Sarah'
>>> next(itr)
   'Roark'
```

In this example, l is an iterable list and itr is the associated iterator, obtained with iter( ). Each next(itr) call obtains the next value from itr.

*Notice how an iterator **retains its state internally**. It knows which values have been obtained already, so when you call next( ), it knows what value to return next.*

What happens when the iterator runs out of values? Let's make one more next( ) call on the iterator above:

```
>>> next(itr)
   Traceback (most recent call last):
   File "<stdin>", line 1, in <module>
   StopIteration
```

*If all the values from an iterator have been returned already, a subsequent next( ) call raises a **Stop Iteration exception**. Any further attempts to obtain values from the iterator will fail.*

We can only obtain values from an iterator in one direction. We can't go backward. There is no prev( ) function which means we iterate on a iterator only once. But we can define two independent iterators on the same iterable object:

```
>>> l = ['Sarah', 'Roark', 30]
>>> itr1 = iter(l)
>>> itr2 = iter(l)
>>> next(itr1)
 'Sarah'
>>> next(itr1)
 'Roark'
>>> next(itr1)
 30
>>> next(itr2)
 'Sarah'
```

Even when iterator itr1 is already at the end of the list, itr2 is still at the beginning. Each iterator maintains its own internal state, independent of the other.

If we want to grab all the values from an iterator at once, we can use the built-in list( ) function. Among other possible uses, list( ) takes an iterator as its argument, and returns a list consisting of all the values that the iterator yielded:

```
>>> l = ['Sarah', 'Roark', 30]
>>> itr = iter(l)
>>> list(itr)
  ['Sarah', 'Roark', 30]
```

*However, we should not be making a habit of this. Part of the elegance of iterators is that they are "**lazy**." That means when we create an iterator, it doesn't generate all the items it can yield just then. It waits until we ask for them with next( ). Items are not created until they are requested.*

When you use list( ), tuple( ), or the like, we are forcing the iterator to generate all its values at once, so they can all be returned. If the total number of objects the iterator returns is very large, it may take a long time.

Behind the scenes of for loop :

let's consider a simple for loop:

```
>>> l = ['Sarah', 'Roark', 30]
>>> for item in l:
...     print(item)
...
Sarah
Roark
30
```

The above loop can be described entirely in terms of the concepts we have just learned about.

To carry out the iteration this for loop describes, Python does the following:

- ✓ *Calls iter( ) to obtain an iterator for l*
- ✓ *Calls next( ) repeatedly to obtain each item from the iterator in turn*
- ✓ *Terminates the loop when next( ) raises the StopIteration exception*

The loop body is executed once for each item next( ) returns, with loop variable item set to the given item for each iteration.

Building Custom Iterators:

*Building an iterator from scratch is easy in Python. We just have to implement the *dunder methods __iter__( ) and the __next__( ) methods, in our class objects. We will deal with them in Oops concepts.*

**Dunder methods, or magic methods, are special methods in Python that you can use to enrich your classes. The term "dunder" stands for "double under" and is used because these special methods have a double underscore in their prefix and suffix.*

There's an easier way to create iterators in Python is ***Generators***

Python Itertools

Before going to generators let's have a look at Standard Library module called itertools containing many functions that return iterables we can use for functional programming. We'll see some of the functions it offers.

a. count( ) in Python Itertools :

The function count( ) in python Itertools takes, as an argument, an integer number to begin count at. It then counts infinitely, unless we break out of the for-loop using an if-statement.

```
from itertools import count
```

```
for i in count(7):
        if I > 14 : break
        print(i)
Output :
7
.
.
14
```

Let's try calling it without an argument and without a terminating if-condition.

```
 for i in count():
        print(i)
Output :
1
……….
# will go until we make a keyboard interrupt
Traceback (most recent call last):
File "<pyshell#265>", line 2, in <module>
print(i)
Keyboard Interrupt
```

Here, we had to press Ctrl+C to interrupt this infinite iterator.

We can also give it a positive/ negative interval as a second argument.

```
for i in count(2,2):
    if i>8: break
    print(i)
Output :
2
4
6
8
```

Now, let's see another function.

b. cycle( ) in Python Itertool :

Cycle( ) function in python itertools infinitely iterates over a python iterables, unless we explicitly break out of the loop.

```
from itertools import cycle
c=0
for i in cycle(['red','blue']):
    if c>7: break
    print(i)
    c+=1
Output:
red     blue
red     blue
red     blue
red     blue
```

c. repeat ( ) in Python Itertool :

This one repeats an object infinitely unless explicitly broken out of.

```
from itertools import repeat
c=0
for i in repeat ([1,2,3]) :
    if c>7:break
    print(i)
    c+=1
Output:
[1, 2, 3]
[1, 2, 3]
[1, 2, 3]
[1, 2, 3]
[1, 2, 3]
[1, 2, 3]
[1, 2, 3]
[1, 2, 3]
```

We can also specify the number of times we want it to repeat, as a second argument.

```
for i in repeat([1,2,3],4):
    print(i)
Output :
[1, 2, 3]
[1, 2, 3]
[1, 2, 3]
```

Python Generators

- ✓ *Generator functions allow you to declare a function that behaves like an iterator, i.e. it can be used in a for loop.*

- ✓ *we can say that the generator provides a way of creating iterators. It solves the following common problem : In Python, to build an iterator. First, we require to write a class and implement the __iter__( ) and __next__( ) methods. Secondly, we need to manage the internal states and throw StopIteration exception when there is no element to return*

- ✓ *A generator in Python is a function with unique abilities. We can either suspend or resume it at run-time. It returns an iterator object which we can step through and access a single value in each iteration*

How to Create a Generator in Python?

There are two straightforward ways to create generators in Python.

a) Generator Function

b) Generator Expression

A. Generator Function :

We write a generator in the same style as we write a user-defined function.

The difference is that we use the ***yield statement*** *instead of the return. It notifies Python interpreter that the function is a generator and returns an iterator.*

```
# Generator Function Syntax
#
def gen_func(args):
    ...
    while [cond]:
        ...
        yield [value]
```

The return statement is the last call in a function, whereas the yield temporarily suspends the function, preserves the states, and resumes execution later.If a function contains at least one yield statement (it may contain other yield or return statements), it becomes a generator function. Both yield and return will return some value from a function.

Below is a simple example of a Python generator function

```
# A simple generator functions
def my_gen():
    n = 1
    print('This is printed first')
    # Generator function contains yield statements
     yield n
    n += 1
    print('This is printed second')
    yield n
    n += 1
    print('This is printed at last')
    yield n
```

Interview Q&A

What are Python Iterators and Iteration?

In very simple words, Python iterators are objects that we can iterate upon. Iterating means to repeat a process multiple numbers of times. A single process is called an iteration. Note: Iterators in Python will not print all elements of the object, it will print only one element at a time.

What are Python Generators?

A Python generator is a function that produces a sequence of results. It works by maintaining its local state, so that the function can resume again exactly where it left off when called subsequent times. Thus, you can think of a generator as something like a powerful iterator.

Why do we use a generator in Python?

- ✓ *It is an easy and clear way of creating our own iterator.*
- ✓ *It is memory efficient because it does not store any value because it just returning values using yield statement.*
- ✓ *It is used to produce an infinite stream of data.*
- ✓ *It can be used to perform pipeline operations.*

What is difference between yield and retrun?

You see, when a return statement is invoked inside a function, it permanently passes control back to the caller of the function. When a yield is invoked, it also passes control back to the caller of the function—but it only does so temporarily.

Whereas a return statement disposes of a function's local state, a yield statement suspends the function and retains its local state.

What is difference between Generator & iterator in Python?

An iterator in Python serves as a holder for objects so that they can be iterated over; a generator facilitates the creation of a custom iterator.

What are Benefits of Python Iterator?

Iterators can work on the principle of lazy evaluation: as you loop over an iterator, values are generated as required.

The main advantage of using the iterators is that the program is only holding one object at a time from a sequence or a collection. For example, to perform an additional operation on each element of a huge list like [1334, 5534, 5345, 345, 144, …….]. We will only hold one value at a time to perform operations on. There is no need to keep all the elements of the list in the memory. This saves the resources of computers

Key takeaways:

- ✓ *Iteration: The process of looping through the objects or items in a collection.*
- ✓ *Iterable: An object that can be iterated over.*
- ✓ *Iterator: The object that produces successive items or values from its . associated iterable.*
- ✓ *iter( ): The built-in function used to obtain an iterator from an iterable.*
- ✓ *next( ): Return next item in container. If there are no further items, raise the StopIteration exception.*
- ✓ *Generators produce values one-at-a-time as opposed to giving them all at once.*
- ✓ *There are two ways to create generators : generator functions and generator expressions.*
- ✓ *Generator functions yield, regular functions return.*
- ✓ *Generator expressions need ( ), list comprehensions use [].*
- ✓ *You can only use a generator once.*
- ✓ *There are two ways to get values from generators: the next() function and a for loop. The for loop is often the preferred method.*
- ✓ *We can use generators to read files and give us one line at a time*

'IT'S GOOD TO BE LAZY'