# ITERATORS

An **iterator** is an object that can be iterated upon, meaning that you can traverse through all the values. An **iterator** is an object which implements the **iterator** protocol, which consist of the methods \_\_iter\_\_() and \_\_next\_\_()

An object is called **iterable** if we can get an iterator from it. Most built-in containers in Python like: [list](https://www.programiz.com/python-programming/list), [tuple](https://www.programiz.com/python-programming/tuple), [string](https://www.programiz.com/python-programming/string) etc. are iterables.

The iter() function (which in turn calls the \_\_iter\_\_() method) returns an iterator from them.

**Iterating Through an Iterator**

We use the next() function to manually iterate through all the items of an iterator. When we reach the end and there is no more data to be returned, it will raise the StopIteration Exception. Following is an example.

# define a list

my\_list = [4, 7, 0, 3]

# get an iterator using iter()

my\_iter = iter(my\_list)

# iterate through it using next()

# Output: 4

print(next(my\_iter))

# Output: 7

print(next(my\_iter))

# next(obj) is same as obj.\_\_next\_\_()

# Output: 0

print(my\_iter.\_\_next\_\_())

# Output: 3

print(my\_iter.\_\_next\_\_())

# This will raise error, no items left

next(my\_iter)

**Output**

4

7

0

3

Traceback (most recent call last):

File "<string>", line 24, in <module>

next(my\_iter)

StopIteration

A more elegant way of automatically iterating is by using the [for loop](https://www.programiz.com/python-programming/for-loop). Using this, we can iterate over any object that can return an iterator, for example list, string, file etc.

>>> for element in my\_list:

... print(element)

...

4

7

0

3

## **WORKING OF FOR LOOP FOR ITERATORS**

for element in iterable:

# do something with element

Is actually implemented as.

# create an iterator object from that iterable

iter\_obj = iter(iterable)

# infinite loop

while True:

try:

# get the next item

element = next(iter\_obj)

# do something with element

except StopIteration:

# if StopIteration is raised, break from loop

break

So internally, the for loop creates an iterator object, iter\_obj by calling iter() on the iterable.

Inside the loop, it calls next() to get the next element and executes the body of the for loop with this value. After all the items exhaust, StopIteration is raised which is internally caught and the loop ends.

## **BUILDING CUSTOM ITERATORS**

Building an iterator from scratch is easy in Python. We just have to implement the \_\_iter\_\_() and the \_\_next\_\_() methods.

The \_\_iter\_\_() method returns the iterator object itself. If required, some initialization can be performed.

The \_\_next\_\_() method must return the next item in the sequence. On reaching the end, and in subsequent calls, it must raise StopIteration.

class PowTwo:

"""Class to implement an iterator

of powers of two"""

def \_\_init\_\_(self, max=0):

self.max = max

def \_\_iter\_\_(self):

self.n = 0

return self

def \_\_next\_\_(self):

if self.n <= self.max:

result = 2 \*\* self.n

self.n += 1

return result

else:

raise StopIteration

# create an object

numbers = PowTwo(3)

# create an iterable from the object

i = iter(numbers)

# Using next to get to the next iterator element

print(next(i))

print(next(i))

print(next(i))

print(next(i))

print(next(i))

**Output**

1

2

4

8

Traceback (most recent call last):

File "/home/bsoyuj/Desktop/Untitled-1.py", line 32, in <module>

print(next(i))

File "<string>", line 18, in \_\_next\_\_

raise StopIteration

StopIteration

We can also use a for loop to iterate over our iterator class.

>>> for i in PowTwo(5):

... print(i)

...

1

2

4

8

16

32