What is Python Iterator?

An Iterator is a collection of objects that holds multiple values and provides a mechanism to traverse through them.

Examples of inbuilt iterators in Python are **lists**, **dictionaries**, **tuples**, etc. It works according to the iterator protocol.

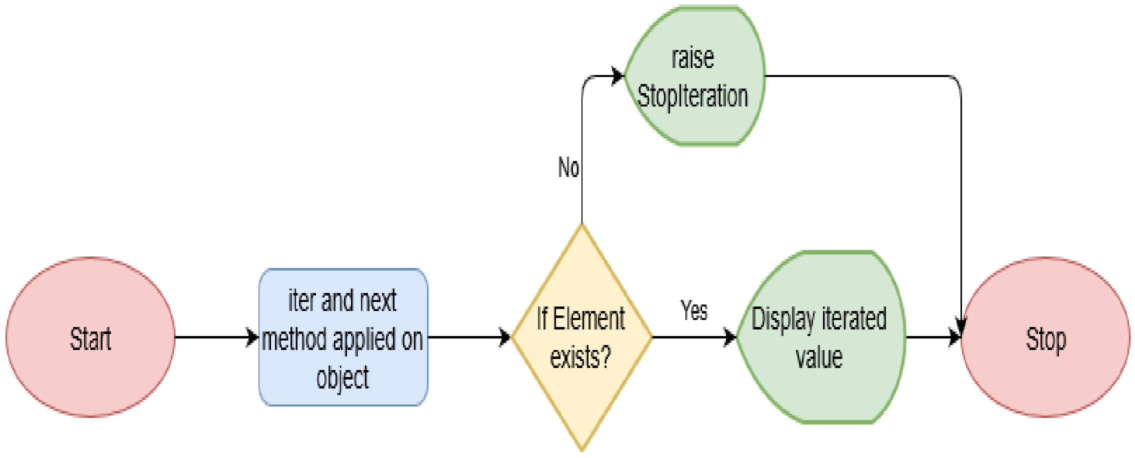
The protocol requires to implement two methods.
They are __iter__ and __next__.

1. __iter__() function returns an iterable object,
2. __next__() gives a reference of the following items in the collection.

How does Iterator work in Python?

Most of the time, we have to use an import statement for calling functions of a module in Python. However, iterators don't need one as we can use them implicitly.

The following flowchart attempts to simplify the concept.



create an iterable object as per the below instruction:

iterable_object = iter(object_to_iterate_through)

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

```
next(iterable_object)
```

Example :

1. ***Creating an iterable from Tuple***

```
Cubes = (1, 8, 27, 64, 125, 216)

cube = iter(Cubes)

print(next(cube))

print(next(cube))
```

Output

```
1

8
```

2. ***Creating an iterable from List***

```
Negative_numbers = [-1, -8, -27, -64, -125, -216]

Negative_number = iter(Negative_numbers)

print(next(Negative_number))

print(next(Negative_number))
```

Output

```
-1

-8
```

3. Iterating through an empty object

```
List = []
```

```
empty_element = iter(List)

print(next(empty_element))

print(next(empty_element))
```

Output

```
Traceback (most recent call last):

File "C:\Users\porting-dev\AppData\Local\Programs\Python\Python35\test11.py", line 3, in <module>

next(empty_element)

StopIteration
```

4. Iterating a non-existent object

```
List = [1,2,3,4]

empty = iter(List)

print(next(empty))

print(next(empty))
```

Output

```
1

2
```

5. Printing a list of natural numbers

```
class natural_numbers:

    def __init__(self, max = 0):

        self.max = max
    def __iter__(self):
        self.number = 1
        return self

    def __next__(self):
        if self.max == self.number:
            raise StopIteration
        else:
            number = self.number
            self.number += 1
            return number

numbers = natural_numbers(10)
i = iter(numbers)
print("# Calling next() one by one:")
print(next(i))
print(next(i))
print("\n")

# Call next method in a loop
print("# Calling next() in a loop:")
for i in numbers:
    print(i)
```

To execute the above program,

Output

```
# Calling next() one by one:
1
2


# Calling next() in a loop:
1
2
3
4
5
6
7
8
9
```

What are generators?

- 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.
- Alternatively, we can say that the generator provides a way of creating iterators.
- It solves the following common problem.

In Python, it is cumbersome to build an iterator.

1. First, we require to write a class and implement the **__iter__()** and **__next__()** methods.
2. Secondly, we need to manage the internal states and throw **StopIteration** exception when there is no element to return.

How to Create a Generator in Python?

The procedure to create the generator is as simple as writing a regular 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.

```
# Demonstrate Python Generator Function


def fibonacci(xterms):

    # first two terms

    x1 = 0

    x2 = 1

    count = 0


    if xterms <= 0:

       print("Please provide a +ve integer")

    elif xterms == 1:

       print("Fibonacci seq upto",xterms,":")
```

```
        print(x1)
    else:
        while count < xterms:
            xth = x1 + x2
            x1 = x2
            x2 = xth
            count += 1
            yield xth

fib = fibonacci(5)


print(next(fib))
print(next(fib))
print(next(fib))
print(next(fib))
print(next(fib))
print(next(fib))
```

The code prints the following output after execution.

```
1
2
3
5
```

```
8
Traceback (most recent call last):
  File "C:/Python/Python35/python_generator.py", line 29, in
    print(next(fib))
StopIterararation
```

Return vs. Yield

The return is **a final statement** of a function. It provides a way to **send some value back**. While returning, its **local stack also gets flushed**. And any new call will **begin execution from the very first statement.**

On the contrary, the yield **preserves the state between subsequent function calls.** It resumes **execution from the point where it gave back the control to the caller**, i.e., right after the last yield statement.

Generator vs. Function

We have listed down a few facts to let you understand the difference between a generator and a regular function.

- A generator uses the **yield statement** to send a value back to the caller whereas a function does it using the **return**.
- The generator function can have one or more than one yield call.
- The **yield** call pauses the execution and returns an iterator, whereas the return statement is the last one to be executed.
- The **next()** method call triggers the execution of the generator function.
- Local variables and their states retain between successive calls to the **next()** method.
- Any additional call to the **next()** will raise the **StopIteration** exception if the there is no further item to process.
-

When to use a Generator?

There are many use cases where generators can be useful. We have mentioned some of them here:

- Generators can help to process large amounts of data. They can let us do the calculation when we want, also known as the **lazy evaluation**.
- We can also stack the generators one by one and use them as pipes as we do with the Unix pipes.
- The generators can also let us establish concurrency.
- We can utilize Generators for reading a vast amount of large files. It will help in keeping the code cleaner by splitting the entire process into smaller entities.
- Generators are super useful for web scraping and help increasing crawl efficiency.
- They can allow us to fetch the single page, do some operation, and move on to the next.
- This approach is far more efficient and straightforward than retrieving all pages at once and then use another loop to process them.

Why use Generators?

Generators provide many programming-level benefits and extend many run-time advantages which influence programmers to use them.

```
def nextSquare():
    i = 1

    # An Infinite loop to generate squares
```

```
 while True:
        yield i*i
        i += 1                    # Next execution resumes from this point


# Driver code to test above generator function

for num in nextSquare():
    if num > 100:
        break
    print(num)
```