What are Iterators, Generators And Decorators in Python?

Iterators

- An iterator is an object that can be iterated upon which means that you can traverse through all the values. Lists, tuples, dictionaries, and sets are all iterable objects.
- To create an object as an iterator you have to implement the methods **iter**() and **next**() to your object where —
  - **iter**() returns the iterator object itself. This is used in for and in statements.
  - **next**() method returns the next value in the sequence. In order to avoid the iteration to go on forever, raise the StopIteration exception.

In [ ]:
```python
class example_range:
    def __init__(self, n):
        self.i = 4
        self.n = n
    def __iter__(self):
        return self
    def __next__(self):
        if self.i < self.n:
            i = self.i
            self.i += 1
            return i
        else:
            raise StopIteration()
n= example_range(10)
print(list(n))
n = example_range(15)
for i in n :
    print (i, end=',')
```

```
[4, 5, 6, 7, 8, 9]
4,5,6,7,8,9,10,11,12,13,14,
```

In [ ]:
```python
# Iterate through a list of string elements

# Using iterator
l = ['Subhash','Dixit','Anushka','Biswas']
iterator = iter(l)
while True:
    try:
        x = iterator.__next__()
        print(x, end=',')
    except StopIteration as e:
        print("\n")
        break

# Using For loop
l = ['Subhash','Dixit','Anushka','Biswas']
```

```
for i in l:
    print(i,end = ",")
print("\n")

# Iterate through a list of integer elements

# Using iterator
l = range(5)
iterator = iter(l)
while True:
    try:
        x = iterator.__next__()
        print(x, end=',')
    except StopIteration as e:
        print("\n")
        break

# Using For loop
l = range(5)
for i in l:
    print(i,end = ",")
print("\n")
```

```
Subhash,Dixit,Anushka,Biswas,

Subhash,Dixit,Anushka,Biswas,

0,1,2,3,4,

0,1,2,3,4,
```

Why use iterators?

- Iterators allow us to create and work with lazy iterable which means you can use an iterator for the lazy evaluation. This allows you to get the next element in the list without re-calculating all of the previous elements. Iterators can save us a lot of memory and CPU time.
- Python has many built-in classes that are iterators, e.g — enumerate, map ,filer , zip and reversed etc. objects are iterators.

Generators

- Generator functions act just like regular functions with just one difference they use the Python yield keyword instead of return.
- A generator function is a function that returns an iterator.
- A generator expression is an expression that returns an iterator. Generator objects are used either by calling the next method on the generator object or using the generator object in a “for in” loop.

In [ ]:
```
def test_sequence():
    num = 0
    while num < 10:
        yield num
        num += 1
for i in test_sequence():
       print(i, end=",")
```

```
0,1,2,3,4,5,6,7,8,9,
```

- A return statement terminates a function entirely but a yield statement pauses the function saving all its states and later continues from there on successive calls.

Python Generators with a Loop

In [ ]:
```
#Reverse a string
#  Using Generator
def reverse_str(test_str):
    length = len(test_str)
    for i in range(length - 1, -1, -1):
        yield test_str[i]
for char in reverse_str("Trojan"):
    print(char,end =" ")
print("\n")

# Using for loop
text = "Trojan"
for char in range(len(text)-1,-1,-1):
    print(text[char],end =" ")
```

```
n a j o r T

n a j o r T
```

Generator Expression

- Generator expressions can be used as function arguments. Just like list comprehensions, generator expressions allow you to quickly create a generator object within minutes with just a few lines of code.**

In [ ]:
```
# Initialize the list
test_list = [1, 3, 6, 10]

# list comprehension
list_comprehension = [x**3 for x in test_list]

# generator expression
test_generator = (x**3 for x in test_list)

print(list_comprehension)
print(type(test_generator))
print(tuple(test_generator))
```

```
[1, 27, 216, 1000]
<class 'generator'>
(1, 27, 216, 1000)
```

- The major difference between a list comprehension and a generator expression is that a list comprehension produces the entire list while the generator expression produces one item at a time as lazy evaluation. For this reason, compared to a list comprehension, a generator expression is much more memory efficient which can be understood from the profiling code below —

In [ ]:
```python
import sys

# List comprehension size
cubed_list = [i ** 3 for i in range(10000)]
print("List comprehension size(bytes):", sys.getsizeof(cubed_list))

# Generator comprehension size
cubed_generator = (i ** 3 for i in range(10000))
print("Generator Expression object(bytes):", sys.getsizeof(cubed_generator))
```

```
List comprehension size(bytes): 85176
Generator Expression object(bytes): 104
```

Decorator

- A decorator in Python is any callable Python object that is used to modify a function or a class. It takes in a function, adds some functionality, and returns it. Decorators are a very powerful and useful tool in Python since it allows programmers to modify/control the behaviour of a function or class. Decorators are usually called before the definition of a function you want to decorate. There are two different kinds of decorators in Python:
    - Function decorators
    - Class decorators

In [ ]:
```python
def test_decorator(func):
    def function_wrapper(x):
        print("Before calling " + func.__name__)
        res = func(x)
        print(res)
        print("After calling " + func.__name__)
    return function_wrapper

@test_decorator
def sqr(n):
    return n ** 2
sqr(4)
```

```
Before calling sqr
16
After calling sqr
```

Multiple Decorators to a Single Function

- When using Multiple Decorators for a single function, the decorators will be applied in the order they've been called.

```
In [ ]: #  Aplying decorator to lowercase the string first and then split
        def lowercase_decorator(function):
            def wrapper():
                func = function()
                make_lowercase = func.lower()
                return make_lowercase
            return wrapper

        def split_string(function):
            def wrapper():
                func = function()
                split_string = func.split()
                return split_string
            return wrapper

        @split_string
        @lowercase_decorator
        def test_func():
            return 'MOTHER OF DRAGONS'
        test_func()
```

```
Out[ ]: ['mother', 'of', 'dragons']
```

```
In [ ]: #  Aplying decorator to sqwuare the number first and then multiply by 2
        def square(function):
            def wrapper():
                func = function()
                sq = func*func
                return sq

            return wrapper

        def multiply_by_2(function):
            def wrapper():
                func1 = function()
                multiply = func1*2
                return multiply

            return wrapper

        @multiply_by_2
        @square
        def test_func():
            return 5

        r = test_func()
        print(r)
```

```
50
```

References

- https://rishikonapure.medium.com/what-are-iterators-generators-and-decorators-in-python-d3f9064184c6
- https://medium.com/nerd-for-tech/python-iterators-and-iterables-904abf5518e7

THE END