Comprehensions, Generators, Iterators, Lambda expression

CONTENTS

- List comprehensions: Basic and advanced. Nine versions.
- Implementation of iterator protocol using yield statement, generators, iterators.
- Functional programming: lambda expression, map, filter, reduce and functools.

List comprehensions:

List comprehension is an elegant way to define and create list in Python.

Example:

Ex. WAP to print all the vowels present in the word entered by user.

```
word = input("enter a word")
print({i for i in word if i in "aeiou"})
```

```
enter a wordMumbai
{'u', 'i', 'a'}
```

Implementation of iterator protocol using yield statement, generators, iterators.

The idea of generators is to calculate a series of results one-by-one on demand (on the fly). In the simplest case, a generator can be used as a **list**, where each element is calculated lazily. Let's compare a list and a generator that do the same thing - return powers of two:

Example:

```
the_list = [2**x for x in range(5)]

print(type(the_list))

# Iterate over list and print
for element in the_list:
    print(element, end=" ")

print("\nsize of list = ",len(the_list))

# Output:
<class 'list'>
1 2 4 8 16
size of list =  5
```

Example:

```
the_generator = (x+x for x in range(3))

print(type(the_generator))

# Iterate over items and print them
for element in the_generator:
       print(element, end=" ")

# Everything looks the same, but the length...
print("\nsize of list = ",len(the_generator))

# Output:
<class 'generator'>
0 2 4
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-55-e06b4974261c> in <module>()
      8
      9 # Everything looks the same, but the length...
---> 10 print("\nsize of list = ",len(the_generator))

TypeError: object of type 'generator' has no len()
```

yield()

The **yield** statement is used to define generators, replacing the **return** of a function to provide a result to its caller without destroying local variables. Unlike a function, where on each call it starts with new set of variables, a generator will resume the execution where it was left off.

Example:

```
# List of prime numbers between 1 - 50 using yeild()
import math

def prime_nos(x=2, y=50):
    for i in range(x, y):
        for j in range(2, int(math.sqrt(i)) + 1):
            if i % j == 0:
                break
        else:
            yield (i)

print(list(prime_nos()))

# OUTPUT:
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

iter()

The iter() method creates an object which can be iterated one element at a time.These objects are useful when coupled with loops like for loop, while loop.

The syntax of iter() method is:

```
iter(object[, sentinel])
```

Example:

```
# list of vowels
vowels = ['a', 'e', 'i', 'o', 'u']

vowelsIter = iter(vowels)

# prints 'a'
print(next(vowelsIter))

# prints 'e'
print(next(vowelsIter))

# prints 'i'
print(next(vowelsIter))

# prints 'o'
print(next(vowelsIter))

# prints 'u'
print(next(vowelsIter))
```

Functional programming: lambda expression, map, filter, reduce and functools.

Example:

```
func = lambda x,y : x+y
func(2,3)

#OUTPUT
5
```

Example:

```
l = ['house', 'car', 'aeroplane', 'bike']
l1 = list(filter(lambda x : len(x) <= 4 , l))
print(l1)

#OUTPUT
['car', 'bike']
```