Assignment_25

1) . What is the difference between enclosing a list comprehension in square brackets and parentheses?

List Comprehension with square brackets produces list.

List Comprehension with parentheses creates generators

In [1]:

```
lstSquare = [i for i in range(10)]
lstSquare
```

Out[1]:

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

If you are familiar with list comprehensions, then this look likes it might create a tuple which is (1,2,3,4,....), but it is actually a generator expression - this expression is a one time only iterator which will yield the values 1, 2, 3, 4.... in that order

In [2]:

```
lstParanthesis = (i for i in range(10))
lstParanthesis
```

Out[2]:

```
<generator object <genexpr> at 0x000001D79832DA50>
```

In [3]:

```
list(lstParanthesis)
```

Out[3]:

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

2) What is the relationship between generators and iterators?

An iterator is an object which contains a countable number of values and it is used to iterate over iterable objects like list, tuples etc

Using an iterator-

iter() keyword is used to create an iterator containing an iterable object.

next() keyword is used to call the next element in the iterable object.

Generator

It is another way of creating iterators in a simple way where it uses the keyword “yield” instead of returning it in a defined function

Generators are implemented using a function

Here, the yield function returns the data without affecting or exiting the function.

It will return a sequence of data in an iterable format where we need to iterate over the sequence to use the data as they won’t store the entire sequence in the memory

By using next() function we can iterate the output of generator function

In [13]:

```
# iterator
iter_list = iter(['Apple', 'Orange', 'Banana'])
print(next(iter_list))
print(next(iter_list))
print(next(iter_list))
```

```
Apple
Orange
Banana
```

In [24]:

```
# GEnerator

def sq_numbers(n):
    for i in range(1, n+1):
        yield i*i
```

In [25]:

```
sq_numbers(8)
```

Out[25]:

```
<generator object sq_numbers at 0x000001D797A1E200>
```

In [26]:

```
lst = sq_numbers(8)
```

In [27]:

```
lst
```

Out[27]:

```
<generator object sq_numbers at 0x000001D7979F8F20>
```

In [28]:

```
print(next(lst))
print(next(lst))
print(next(lst))
print(next(lst))
```

```
1
4
```

9
16

3) What are the signs that a function is a generator function?

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.

4) What is the purpose of a yield statement?

A yield statement looks much like a return statement, except that instead of stopping execution of the function and

returning, yield instead provides a value to the code looping over the generator and pauses execution of the generator

function

5) What is the relationship between map calls and list comprehensions? Make a comparison and contrast between the two.

Map function:

Suppose we have a function and we want to compute this function for different values in a single line of code . This is where map() function plays its role. map() function returns a map object(which is an iterator) of the results after applying the given function to each item of a given iterable (list, tuple etc.)

If we already have a function defined, it is often good to use map For example, map(sum, myLists) is more neat than [sum(x) for x in myLists]. You gain the elegance of not having to make up a dummy variable (e.g. sum(x) for x... or sum(_) for _... or sum(readableName) for readableName...) which you have to type twice, just to iterate.

List Comprehension:

List Comprehension is a substitute for the lambda function, map(), filter() and reduce()

Comparision :

1. List comprehension is more concise and easier to read as compared to map

2. List comprehension allows filtering. In map, we have no such facility

For example, to print all even numbers in range of 100, we can write [n for n in range(100) if n%2 == 0]. There is

no alternate for it in map

3. List comprehension are used when a list of results is required, where as map only returns a map object and does

not return any list.

4. List comprehension is faster than map when we need to evaluate expressions that are too long or complicated

to express

5. Map is faster in case of calling an already defined function (as no lambda is required)