**Course Name: Python Programming with Django**

**Name: Bipul Dutta**

**Batch ID: 13328**

**Order No: 4099423**

**Assignment- 10. Date: 21.05.22**

**\*\* Ques 01: Advantage and Disadvantage of Iterator**

**\*\* Advantages:**

1. Code reduction
2. Code redundancy is greatly solved
3. Reduces code complexity
4. It brings more stability into code
5. Easy to read in code

Iterator Coding:

# Iterator:  
my\_list = [1,2,3]  
iter\_obj = iter(my\_list)  
  
print(type(iter\_obj))  
print(next(iter\_obj))  
print(next(iter\_obj))  
print(next(iter\_obj))

\*\* **Disadvantage**

1. Need different data structure at same time.
2. It needs update regular basis
3. It needs to backtrack while processing through a list
4. Iterator may not work at all

**Ques 02: What is Infinite Iterator?**

Iterator in Python is any python type that can be used with a ‘for in loop’. Python lists, tuples, dictionaries, and sets are all examples of inbuilt iterators. But it is not necessary that an iterator object has to exhaust, sometimes it can be infinite. Such type of iterators are known as Infinite iterators.

Python provides three types of infinite iterators –

count(start, step): This iterator starts printing from the “start” number and prints infinitely. If steps are mentioned, the numbers are skipped else step is 1 by default. See the below example for its use with for in loop.

Example:

# Python program to demonstrate

# infinite iterators

import itertools

# for in loop

for i in itertools.count(5, 5):

if i == 35:

break

else:

print(i, end =" ")

**Output:** 5 10 15 20 25 30

**Ques 03: Advantage and Disadvantage of Generator**

**Advantages:**

1. Memory efficient method of generating sequence types in python.
2. Generator functions are better in case of memory utilization and code performance because they allow the function to avoid doing all work at a time..
3. Time-efficient when compared to list comparisons.

**Disadvantages:**

1. For the generator's work, you need to keep in memory the variables of the generator function.
2. Every time you want to reuse the elements in a collection it must be regenerated.

**Ques 04: Normal vs Generator Function?**

There are few points that can clear the difference between Generator Function and Normal Function.

|  |  |
| --- | --- |
| **Normal Function** | **Generator Function** |
| 1.Normal function has only one **return** statement in the loop whereas generator function can use one  or more **yield** statement in the loop. | 1. Presence of yield statement shows that the function is not normal function but a generator |
| 2.While calling the generator functions, the normal function take pause immediately and control has been transferred to the caller. | 2. Calling the function like normal functions, does not actually execute the function, but it creates an instance of the function. |
| 3.Local variable and their states are remembered between successive calls of the functions. | 3. The next ( ) method, executes the code written up to first yield statement and then returns the value generated. |
| 4.When the function terminates, StopIteration exception is raised automatically. | 4. When we call next () again, it will resume the processing of function counter, from the place where it was left last till the yield statement is met again. |

**\*\* Ques 05: Use cases of Generator?**

There is a lot of work in building an iterator in Python. We have to implement a class with \_\_iter\_\_() and \_\_next\_\_() method, keep track of internal states, and raise StopIteration when there are no values to be returned.

This is both lengthy and counterintuitive. Generator comes to the rescue in such situations.

Python generators are a simple way of creating iterators. All the work we mentioned above are automatically handled by generators in Python.

Simply speaking, a generator is a function that returns an object (iterator) which we can iterate over (one value at a time).

Example:

*"""*def generator\_function():  
 num = 3  
 yield num  
  
 num = num + 10  
 yield num  
  
my\_gen = generator\_function()  
  
  
print(next(my\_gen))  
print(next(my\_gen))  
for gen in my\_gen:  
 print(gen)