UNIT 15 PYTHON-ADVANCE CONCEPTS

Structure

15.0 INTRODUCTION

In unit number 11 of this course we learned about the functions, we learned that we have inbuilt functions and we also learned to develop our own functions just for the sake of revision, some of the examples are listed below

Say the following lines of code are saved in functions.py

```
# In-Built Function

Name = "IGNOU-SOCIS"

print(len(Name))
```

executing the functions.py by running the command $ python function.py we get the output 11, which is the length of the variable Name. here, len() is the in-built function of python, which can be used to calculate length of any string, we need not to write code agin and again to determne the length of any string simply we need to use this built-in function and get the job done.

We also learned to define our own function just to recapitulate lets write our function to add two numbers

```
# User-Defined Functions

defadd_two(a,b) :

returna+b

total = add_two(5,4)

print(total)
```

here add_two(a,b) is the user defined function that takes two numbers as input and return their sum, which is stored in the varable total, the result of total is printed subsequently.

We also learned the concept of anonymous functions i.e lambda functions or Lambda expressions, the concept was introduced in python and later it was adopted by many other languages be it C++ or Java. The advantage of using the lambda function is that, they can we defined with in the code by writing few lines and they doesn't need any name, that's amazing. One example of Lambda expressions is sited below, here we will write the lambda expression equivalent for the User-Defined Function add_two, which is given above

```
# Lambda Expressions (Anonymous Functions)

def add(a,b)

returna+b

add2 = lambda a,b : a+b

print(add2(2,3))
```

here, add2 collects the output of the lambda function, we can see the output by calling lambda function, by writing print(add2(2,3))

Generally lambda functions are not meant for the user defined functions but are used to facilitate the working of various built in functions like, map, reduce, filter, etc.

Modern programming language emphasizes on code reusability, where one need to work with the already written codes, but these codes needs to be customized, i.e. their functionality is required to be enhanced, and to enhance the functionality of already written functions, python has the concept of decorators. We will learn about decorators in our subsequent section 15.2

15.1 OBJECTIVES

After going through this unit, you will be able to:

- Enhance the functionality of functions by using decorators
- Understand the concept of iterators and iterables
- Appreciate the concept of generators
- Understand the concept cooperative multitasking using co-routines

15.2 DECORATORS

Modern programming paradigm recommends the technique of code reusability, where we need to customize the code as per our requirements, without disturbing the actual functionality of the code which is already written. To achieve this, python introduced the concept of decorators, this concept is used to enhance the functionality of functions, which are already written, for example say we have two functions func1() and func2() as given in the python code given below

```
# Decorators – to enhance functionality of other functions
deffuncA():
print (‘ this is functionA’)
deffuncB():
print (‘ this is functionB’)
funcA()
funcB()
```

on executing the above mentioned code by running deco.py we will get output

```
this is function1
this is function2
```

but without disturbing the existing code if we want that along with, “this is function1” the output should contain the line “this is a wonderful function”, i.e. to enhance the already existing functionality, we need to exercise the concept of decorators.

To understand this concept lets re-write the code given above

```
# Decorators – to enhance functionality of other functions
defdecorator_funcn (any_funcn):
defwrapper_funcn():
print(‘this is a wonderful function’)
any_funcn()
returnwrapper_funcn
deffuncA():
print (‘ this is functionA’)
deffuncB():
print (‘ this is functionB’)
variable = decorator_funcn (funcA)
variable( )
```

Lets understand the concept of Decorators through the above code. Here, decorator_funcn is defined to enhance the functionality of any function (may be funcA or funcnB) from printing “this is function A” to “this is a wonderful function this is function A” or “this is function B” to “this is a wonderful function this is function B”.

Without altering any line of code of already written functions. To achieve this a decorator function decorator_funcn is defined, this function takes any_funcn as input argument, it can be any function may it be funcnA or funcnB. Inside the decorator_funcn, a wrapper function named wrapper_funcn() is defined to wrapup the new features over the existing features of the function taken as argument by the decorator_funcn(). The body of the wrapper_funcn() includes the additional feature, in our case it is print('this is a wonderful function') an then the original function passed as an argument to the decorator_funcn() i.e. any_funcn() is called, and finally the output of wrapper_funcn is returned. To execute the decorator_funcn, the decorator_funcn with argument funcA is called and its return value is collected in variable, then variable() is executed and the output received is "this is a wonderful function this is function A" or "this is a wonderful function this is function B".

Lets run some shortcuts also to work with the concept of decorators, which involves one more concept of syntactic sugar, where w use @ symbol to use the decorators before executing any function, i.e. to enhance its functionality.

```
#  Decorators – to enhance functionality of other functions
# @ used for decorators
defdecorator_funcn (any_funcn):
defwrapper_funcn():
print('this is a wonderful function')
any_funcn()
returnwrapper_funcn
@decorator_funcn
deffuncA():
print (' this is functionA')
funcA()
@decorator_funcn
deffuncB():
print (' this is functionB')
funcB()
```

whenever we use @decorator_funcn before any function the output of that function preceeds with the output "this is a wonderful function", later the output of actual function turnsup. As in this code, calling funcA() leads to output "this is a wonderful function this is function A" or calling funcB() leads to output "this is a wonderful function this is function B".

In this section we learned, the concept of decorators, as functionality enhancers. Now we will study the concept of Iterators and Iterables.

☞ **Check Your Progress 1**

1) Compare built-in functions with user defined and Lambda functions
2) What are Decorators? Briefly discuss the utility of decorators.

15.3 ITERATORS

In Python Iterator is an object that can be iterated i.e. an object which will return data or one element at a time. Iterators exist every where, but they are implemented well in generators, comprehensions and even in for loops; but are generally hidden in plain sights. In this section we will try to understand this concept by exploring the functioning of for loops. In our earlier units we learned about the concepts of Lists, Tuples, Dictionaries, Sets , Strings etc., infact most of these built in containers are collectively called, iterables. An object is called iterable if we can get an iterator from it..In Python any iterator object must follow the iterator protocol i.e. the implementation of iter() and next() functions, the iter function returns an iterator, and an object is called iterable if we can get an iterator from it (for this we use iter()function). To understand what are iterables lets understand the functioning of for loop.

We learned that numbers = [1,2,3,4] is a list and if we want to print the elements of this list, we may use for loop for this purpose, and we will write

```
# Iterables

numbers = [1,2,3,4]

fori in numbers :

print (i)
```

now lets understand how for loop works behind the scenes, firstly the for loop calls a function called iter() function, this iter function changes the iterable to iterator i.e. it takes list as argument, so in out case we have iter(numbers), and this is now an iterator. Subsequently the next function is called whose argument is this iterator i.e. next(iter(numbers)), as the loop progresses the next function provides the values from the iterable i.e. the list numbers in our case, first run provides 1, second run provides 2 and so on. i.e to see how the for loop works we write the logic of for loop as follows

```
# Iterables

numbers = [1,2,3,4]

number_iter = iter(numbers)

print(next(number_iter))        # Output will be 1
print(next(number_iter))        # Output will be 2

print(next(number_iter))        # Output will be 3

print(next(number_iter))        # Output will be 4

print(next(number_iter))        # Output will be Error as the iterable numbers
is upto 4 only
```

In this way only the for loop works on any of the iterables i.e. List or Tuple or String. So, iterables are the python data types which uses the iter and next functions, but iterator can directly use the next function. The iterables uses the iter() function to generate iterator and then uses next function but iteartors don't use iter function they directly use the next function to get the job done. In Python any iterator object must follow the iterator protocol i.e. the implementation of iter() and next() functions, the iter function returns an iterator, and an object is called iterable if we can get an iterator from it (for this we use iter()function)

Now, we will learn about the iterators.

```
# Iterators

numbers = [1,2,3,4]     # Iterables

Squares  =map(lambda a : a**2, numbers)     #Iterator

print( next(squares))   # output will be square of 1 i.e.  1

print( next(squares))   # output will be square of 2 i.e.  4

print( next(squares))   # output will be square of 3 i.e.  9

print( next(squares))   # output will be square of 4 i.e.  16
```

In this section we learned about the concept of Iterators and Iterables, now we will extend our discussion to Generators, in the next section.

☞ **Check Your Progress 2**

1) What are Iterators? How iterators differ from iterables?
2) Discuss the execution of for loop construct, from iterators point of view

15.4 GENERATORS

Generators are also a kind of iterators, but they are quite memory efficient i.e. needs very less memory hence helps in improving the performance of any programme. We learned in last section that iterators involves production of any sequence, the generators are also generating the sequence but their modus operandi is quite different. Like List say L = [1,2,3,4] is a sequence but it is an iterable, the generator is also a sequence but it is an iterator not a iterable. You might be thinking that we already have a mechanism to refer a sequence i.e. say List, then why do we need a generator. To understand this we need to understand the memory utilization by list and generator. Say, we are having a list with many numbers, when we create a list then it will take some time , secondly these numbers will get stored in to the memory i.e memory usage will also be on higher side. But, I the case of Generators, at one time only one number is generated and the same is used for further processing, i.e. both time and memory space are saved, they are comparatively quite less in case of generators. So, while processing the list entire list is loaded and processed, but in generators one by one elements are generated and are processed accordingly.

Now you might be thinking that, when to use lists then ? the answer is that when you need to use your sequence again and again (may be to perform some functionality) then list is the best option, but when you simply need to use the sequence for one time only, then its better to go for generators, you will understand this, later in this section only.

Lets learn how to write a generator, for this you may use two techniques, i.e. you may use generator function or generator comprehension, as technique to develop your own generator. Generator comprehension is the technique which is quite similar to list comprehension, which is used to generate list.

Firstly we will discuss about generator function, say we need to define a function which is suppose to take a number as an argument and that function is suppose to print number from 1 to that number say 10

We can write the following code to achieve the task:

```
defnums(n)

fori in range (1, n+1) :          # n+1 because for loop goes upto n-1 th term

        print(i)

nums(10)        # calling the function nums with n = 10
```

this will print the numbers from 1 to 10, here function is not returning any thing, but simply printing the numbers. This was quite simple, as you leraned in your earlier units, but if we need to develop our generator to do the same task then you need to replace the print command with yield keyword, and the code will be

```
defnums(n)
```

```
fori in range (1, n+1) :           # n+1 because for loop goes upto n-1 th term

        yield(i)

nums(10)
```

This yield keyword will create a generator, on executing this code nothing will be printed, but if in place of nums(10) i.e. the last line of the above code, we write print(nums(10)) then on execution you will come to know a generator object nums is produced.

Note : a) In normal function we either print or return but in generator function we use yield keyword

b) yield is a keyword, and not a function, so we can write 'yield i' in place of 'yield(i)'

Now, lets understand how a generator function works, for this lets again explore the functioning with for loop, refer to the code given below

```
defnums(n)

fori in range (1, n+1) :           # n+1 because for loop goes upto n-1 th term

        yield(i)

numbers = nums(10)

fornum in numbers :

print(num)
```

Executing the above code the sequence of numbers from 1 to 10 will be generated, but if you again execute the for loop then nothing will be printed i.e.

```
defnums(n)

fori in range (1, n+1)  # n+1 because for loop goes upto n-1 th term

        yield(i)

numbers = nums(10) # here nums(10) is a generator, we can transform nums(10) into list by writing list(nums(10))

fornum in numbers :

print(num)
```

```
fornum in numbers :

print(num)
```

execution of the second for loop will not produce any result because the generaors generates the numbers one by one and they are not retained in to the memory as in the case of lists. So the execution of first for loop will produce a sequence from 1 to 10, i.e. the numbers are placed in to the memory one by one, which is there after refreshed, so past instance is lost. Thus the execution of second for loop has no databecausenums(n) function only exists before first for loop not after it. If the nums(n) also exists after first for loop then data for second for loop is also available.

```
defnums(n)

fori in range (1, n+1)  # n+1 because for loop goes upto n-1 th term

        yield(i)

numbers = list(nums(10))  #here nums(10) is a generator, we can transform nums(10) into list by writing list(nums(10))

fornum in numbers :

print(num)

fornum in numbers :

print(num)
```

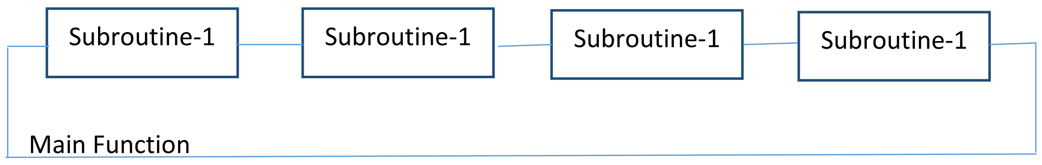
If we re-execute the above code with list and not generator i.e. with list(nums(10)) and not nums(10), then the sequence from 1 to 10 will be printed twice because the content of List persists in the memory, thus the execution of both for loops will produce a separate sequence from 1 to 10.

☞ **Check Your Progress 3**

1) What are generators in Python? Briefly discuss the utility of generators in python
2) Compare Generators and Lists

15.5 CO-ROUTINES

We learned about functions in our earlier units, and we knew that they are also referred as procedures, subroutines, sub-processes etc. In fact a function is packed unit of instructions, required to perform certain task. In a complex function the logic is to divide its working into several self-contained steps, which themselves are functions, such functions are called subroutines or helper functions, these subroutines have single entry point. The coordination of these subroutines is performed by the main function.



The generalized co-routines are referred as subroutines, the working of co-routines relates to cooperative multitasking i.e. when a process voluntarily passes on (yield) the control to another process, periodically or when idle. Co-routines are cooperative that means they link together to form a pipeline. One co-routine may consume input data and send it to other which process it. Finally there may be a co-routine to display result.

This feature of co-routine helps for simultaneous processing of multiple applications. The Co-routines are referred as generalized subroutines, but there is a difference between subroutine and co-routine, the same are given below:

Subroutines	**Co-Routines**
1. Co-routines have many entry points for suspending and resuming execution. 2. Co-routine can suspend its execution and transfer control to other co-routine and can resume again execution from the point it left off. 3. In Co-routines there is no main function to call co-routines in particular order and coordinate the results.	1. Subroutines have single entry point for suspending and resuming execution 2. Subroutines can't suspend its execution and transfer control to other subroutine and can resume again execution from the point it left off. 3. In Subroutines there is main function to call subroutines in particular order and coordinate the results.

From the above discussion it appears that co-routines are quite similar to threads, both seems to do the same job. But, there is a difference in between the thread and the Co-routine, in case of threads, it is the operating system i.e. the run time environment that performs switching in accordance with scheduler. But, in the case Co-routines the decision making for switching is performed by the programmer and programming language. In co-routines the cooperative multitasking, by suspending and resuming at set points is under the control of programmer.

In Python, co-routines are similar to generators but with few extra methods and slight change in how we use yield statement. Generators produce data for iteration while co-routines can also consume data. A generator is essentially a cut down (asymmetric) coroutine. The difference between a coroutine and generator is that a coroutine can accept arguments after it's been initially called, whereas a generator can't.In Python the Co-routines are declared with the async or await syntax, it is the preferred way of writing asyncio applications.

asyncio is a library to write concurrent code using the async/await syntax. asyncio is used as a foundation for multiple Python asynchronous frameworks that provide high-performance network and web-servers, database connection libraries, distributed task queues, etc. To understand co-routine refer to following example code

Example, the following snippet of code (requires Python 3.7+) prints "hello", waits 1 second, and then prints "world":

```
importasyncio

asyncdef main() :

        print ('hello')

        awaitasyncio.sleep(1)

        print ('world')

asyncio.run(main())
```

It is to be noted that by simply calling a co-routine, it will not be scheduled for the execution. To actually run a co-routine, asyncio provides three main mechanisms, listed below:

1. The asyncio.run() function to run the top-level entry point "main()" function (see the above example.)
2. Awaiting on a co-routine. (see the example given below)

Example - The following snippet of code will print "hello" after waiting for 1 second, and then print "world" after waiting for another 2 seconds:

```
importasyncio

import time

asyncdeftell_after(delay_time, what_to_tell):

        awaitasyncio.sleep(delay_time)

        print(what_to_tell)
```

```
asyncdef main():
        print(f"started at {time.strftime('%X')}")
        awaittell_after(1,'hello')
        awaittell_after(2,'world')
        print(f"finished at {time.strftime('%X')}")
asyncio.run(main())
```

Expected output:

```
Started at 17:11:52
hello
world
finished at 17:11:55
```

3. The asyncio.create_task() function to run coroutines concurrently as asyncio Tasks.

Let's modify the above example and run two tell_after coroutines concurrently:

```
asyncdef main() :
        task1 = asyncio.create_task(tell_after(1,'hello'))
        task2 = asyncio.create_task(tell_after(2,'world'))
        print(f"started at {time.strftime('%X')}")
        # wait until both tasks are completed (should take around 2 seconds)
        awaittell_after(1,'hello')
        awaittell_after(2,'world')
        print(f"finished at {time.strftime('%X')}")
```

Expected Output:

```
Started at 17:24:32
hello
world
```

```
finished at 17:24:34
```

Note that expected output now shows that the snippet runs 1 second faster than before

☞ **Check Your Progress 4**

1) What are Co-routines? How they support cooperative multi-tasking in python
2) Compare Subroutines and Co-routines
3) How Co-routines differ from threads

15.6 SUMMARY

In this you learned about decorators, a way to enhance the functionality of already written functions, which ia useful concept for code reusability. Further, the discussion was enhanced to the concepts of iterables and iterators, through the understanding of the execution of For loop. There after the concept of generators was discussed where the concept of yield keyword and print command were mentioned, the comparative analysis between the generator and a function also clars the concept of performance improvement in python programming. Finally, the understanding of co-routines cleared the learners understanding towards the cooperative multitasking.