Week - 3: Control Structures and Functions

Introduction:-

Welcome to the session on '**Control Structures and Functions**'.

Control structures are the essence of programming; they help computers do what they do best: automate repetitive tasks intelligently. The most common control structures are **if-else** statements, **for** and **while** loops, and list and dictionary comprehensions. This session will cover all these concepts.

Another crucial thing you will learn in this session is how to write your own functions. Almost every powerful program, be it a web app or a machine learning algorithm, is a set of functions written to perform specific tasks.

In this session, you will learn:

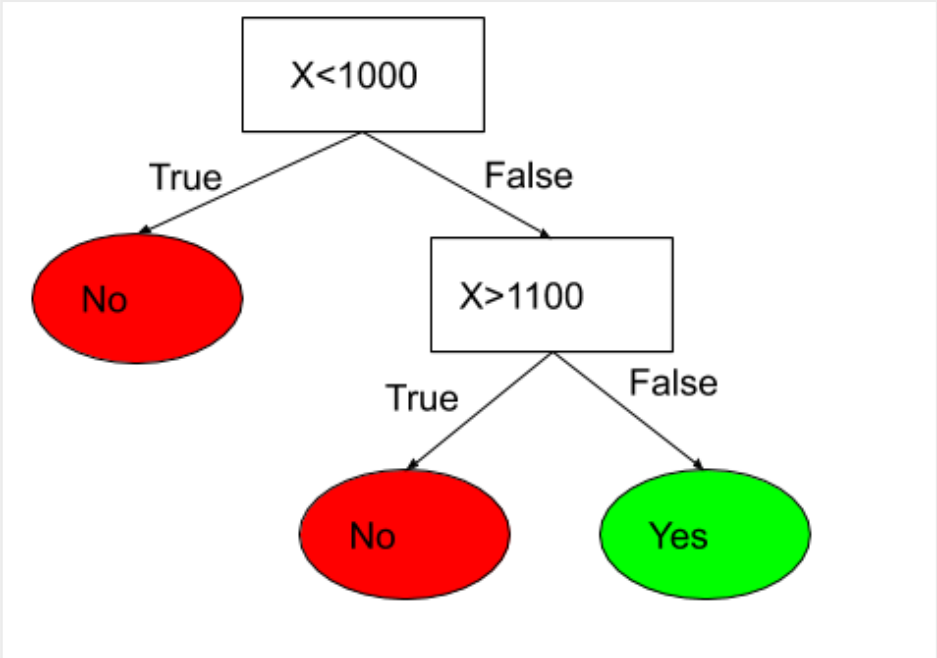
- Control structures
  - If-elif-else
  - For loop
  - While loop
  - List comprehensions
  - Dictionary comprehensions
- Functions
- Map
- Filter
- Reduce

Decision Making:-

If statements are a crucial part of the decision making constructs in Python. The construct of the statement is inspired by the natural language. For instance:

if *it rains today* ***then*** *I would order an umbrella.*

Having understood about relational operators, let's now look at the if-else construct and the role of these relational operations in its implementation.



In the example discussed above:

x = **450**

```
if x < 99:
    print(x, "is less than 99")
else:
  print(x, "is greater than 99")
```

Important things to note here are as follows:

The **colon** that indicates the start of block creation, The **indentation** here defines the scope of the if or else statement, Either code under the if block is executed or under the else block is executed, not both of them.

So now that you have understood the basic if-else construct, can you try to write a code to return YES if x lies in the range of 1000 to 1100, else return NO?

If you break this down, you can get the following conditions:

Implementation:

```
if (X<1000):
    print('No')
else:
    if (X>1100 ):
        print('No')
    else:
        print('Yes')
```

In cases like this, where you would like to make a decision based upon multiple conditions combined together - you can use the logical operators in Python to combine various conditions within a single if-else loop.

you learnt that there are two types of logical operators:

Logical Operator	Description
and	Is true when both the conditions attached to it are true
or	Is true when either one of the conditions is true

If you apply this concept to our earlier example for checking whether x lies in the range of 1000 to 1100, the code gets modified as shown below:

```
if (X > 1000 & X < 1100):
    print('Yes')
else:
    print('No')
```

Isn't this simple compared with the earlier code? That is how logical operators make our logic building easy.

Similar to an if-else construct, there is if-elif-else construct available in Python. The example used in the video is given below:

```
shoppinng_total = 550

if shoppinng_total >= 500:
    print("You won a discount voucher of flat 1000 on next purchase")
elif shoppinng_total >= 250:
    print("You won a discount voucher of flat 500 on next purchase")
elif shoppinng_total >= 100:
    print("You won a discount voucher of flat 100 on next purchase")
else:
    print("OOPS!! no discount for you!!!")
```

Note that in the elif construct, a particular block is executed if all the blocks above the considered block are false **and** this particular block is true. For instance, in the above example, if the shopping total is less than 500 **and** greater than 250 the output will be:

You won a discount voucher of flat 500 on next purchase

So the elif construct can also be used to replace the and operator in certain situations.

In the example shown in the video:

```
world_cups = {2019 : ['England', 'New Zealand'], 2015:["Australia", "New Zealand"], 2011 : ["India", "Sri Lanka"], 2007: ["Australia", "Sri Lanka"], 2003: ["Australia", "India"]}

year = int(input("Enter year to check New Zealand made it to Finals in 20th century : "))

if year in world_cups :
    if "New Zealand" in world_cups[year] :
        print("New Zealand made it to Finals")
    else:
        print("New Zealand could not make it to Finals")

else:
    print("World cup wasnt played in", year)
```

Using an if-else construct would also give us an answer, but imagine that you have 10 conditions like this; wouldn't it be clumsy? Instead, if you use something like an elif-construct, it would make the code more readable.

In this segment, you learnt about relational operators, logical operators, if-else, if-elif-else, and nested if-else constructs. You might be pretty confused about when to use each of them. Read the tips given below, which may help you to arrive at a conclusion faster.

Tips:

When there is more than one condition to check and you need to:

1. Perform a different operation at each condition involving a single variable, you use an if-elif-else condition.
2. Just return a boolean functionality, i.e., Yes or No, you use logical operators in a single if-else construct.
3. Perform a different operation in each condition involving different variables, you use a nested if-else construct.

Python figuring out which of the two conditions is true

Loops and Iterations:-

In the previous session, you learnt about lists and various other data structures.

Let’s look at a small example where you have a person’s income and expense data across five months in the form of a list, and you want to compute his savings across these five months. You may be thinking to do this manually by taking the first elements from the two lists and subtracting them, then again taking the second elements and subtracting, and so on. This may look simple, but let’s say this task has to be done for 10 or 20 years timeframe, in that case, would you have the same strategy?

In cases where we need to repeat a pre-set process n number of times the concept of iteration comes in handy, as you are repeating the same operation multiple times. With this in mind, let’s learn more about it.

While loop:

As you saw in the video, the while loop keeps on executing **until** the defined condition is true. Once the condition fails the loop is exited. Such kind of programs are used in everyday applications such as the pin code lock:

```
#Let's create a pin checker which we generally have in our phones or ATMs
pin = input("Enter your four digit pin: ")
while pin != '1234':                    #Pre-set pin is 1234
    pin = input('Invalid input, please try again: ')
print("Pin validation successful.")
```

But usually, you only get three attempts to unlock anything with a pin code.

To add the functionality of the number of attempts you use a counter variable. The code from the video is given below:

```
# Now if we want to add a maximum number of tries allowed we'll add an 'if loop'

import sys    #required for exiting the code and displaying an error

pin = input("Enter your four digit pin: ")
attempt_count = 1
while pin != '1234':
    if attempt_count >= 3:
     sys.exit("Too many invalid attempts")   #error code
    pin = input('Invalid input, please try again: ')
    attempt_count += 1
print("Pin validation successful.")
```

That was about the while loop, let's move on to the next looping

technique that is the 'for' loop.

'For loop' Control Structure:

By now, you have understood the syntax of a for loop.
for val **in** seq :
    statements

A few things to note in the syntax include:

1. **seq**, which represents a sequence; it can be a list, a tuple or a string. To put it in a simple way it can be any iterable object.
2. The **in** operator, which, as you may recall, is a membership operator that helps to check for the presence of an element in a sequence.

But what role is the '*in*' operator playing here?
When you say for val **in** seq, it means that for a value in sequence, it executes the set of statements in the code block once and returns to '**for**' again and then shifts the position to the next value.
Let's say you are at the last element in the seq. Here, the for block executes, and now, when it again goes to val in seq, there are no more elements, and hence, val in seq returns a false and stops the execution.

The for loop can iterate over any iterable object, we seen a few examples of this already. Dictionary data type is also an iterable object.

Now that you have learnt about the 'for statement', can you answer our initial question of calculating savings using a person's income and expenses data? Let's try this out.

Assume you have -
L1 = [10, 20, 30, 24, 18] (in thousands)
L2 = [8, 14, 15, 20, 10]

What you were doing manually is subtracting the first element of each list and then the second element, and so on. In other words, it is L1[i] - L2[i].

Since you need these indexes, let's create a dummy array of five elements that represent the index positions L3 = [0, 1, 2, 3, 4].

Let's implement the for loop using the list L3:
L1 = [**10**, **20**, **30**, **24**, **18**]
L2 = [**8**, **14**, **15**, **20**, **10**]
L3 = [**0**, **1**, **2**, **3**, **4**]

for i **in** L3:
    L3[i] = L1[i] - L2[i]

Here, you are updating elements of L3 at each iteration. Now think whether you can use the same approach for a list with 1000 elements? To answer this, let's look at our next video.

Note: Here 'i' is an iterator - An **iterable** is any Python object capable of returning its members one at a time, permitting it to be iterated over in a for loop.

Comprehensions:-

Comprehensions are syntactic constructs that enable sequences to be built from other sequences in a clear and concise manner. Here, we will cover list comprehensions, dictionary- comprehensions and set comprehensions.

Using list comprehensions is much more concise and elegant than explicit for loops. An example of creating a list using a loop is as follows:

```
L1 = [10, 20, 30, 24, 18]
L2 = [8, 14, 15, 20, 10]
L3 = []
for  i in range(len(L1)):
   L3.append(L1[i] - L2[i])
L3
```

You know this code from our earlier discussions. The same code using a list comprehension is as follows:

```
# using list comprehension
L1 = [10, 20, 30, 24, 18]
L2 = [8, 14, 15, 20, 10]
L3 = [L1[i] - L2[i] for i  in range(0, len(L1))]
L3
```

You can use list comprehension to iterate through two lists at a time.

```
for I,j in zip(l1,l2):
       print(I,' - ',j)
```

Let's look at an example to understand the dictionary comprehension better. First, using the traditional approach, let's create a dictionary that has the first ten even natural numbers as keys and the square of each number as the value to the key.

```
# Creating a dictionary consisting of even natural numbers as key and square of
```

each element as value
ordinary_dict ={}

```
for i in range(2,21):
    if i % 2 == 0:
        ordinary_dict[i] = i**2

print(ordinary_dict)
```

The same code in terms of a dictionary comprehension is as follows:

```
#Using dictionary comprehension
updated_dict = {i : i**2 for i in range(2,21) if i % 2 ==0}
print(updated_dict)
```

You can see that the comprehension is inside curly brackets, representing that it is dictionary comprehension. The expression inside the brackets first starts with the operation/output that you desire, and then loops and conditionals occur in the same order of the regular code.

The comprehension technique can work on sets as well. Let's look at an application of set comprehensions.

```
word = input("Enter a word : ")
vowels = {i for i in word if i in "aeiou"}
vowels
```

From pdf:-

List Comprehensions in Python
Suppose we need to create a list with first 10 multiple of 6 in it, So we may do this with a normal
for loop or with list comprehensions, Let's see both of them and understand the difference.
Normal For loop list1 =[]
for n in range(1,11): list1.append(n*6) print(list1)
Output:
[6, 12, 18, 24, 30, 36, 42, 48, 54, 60]
List comprehension
list1 = [n*6 for n in range(1,11)] print(list1)
Output:
[6, 12, 18, 24, 30, 36, 42, 48, 54, 60]
We got the same output using list comprehensions just by writing a line of

code.

In general list comprehension

[<the_expression> for <the_element> in <the_iterable>]

Comparing this with our example n*6 is **the expression** , n is **the element** , range(1,11) is **the iterable** .

Applying list comprehension with a condition

Now, Suppose we need to create a list of multiple of 6 for just even numbers between 1 to 10.

```
list1 =[]
for n in range(1,11):
if n%2==0:
list1.append(n*6) print(list1)
```

Output:

[12, 24, 36, 48, 60]

Using list comprehensions

```
list1 = [n*6 for n in range(1,11) if n%2==0] print(list1)
```

Output:

[12, 24, 36, 48, 60]

In general list comprehension

[<the_expression> for <the_element> in <the_iterable> if <the_condition>]

Comparing this with our example n*6 is **the expression** , n is **the element** , range(1,11) is **the iterable** and n%2==0 is **the condition.**

Applying list comprehension with if-else condition

Now, Suppose we need to create a list of multiple of 6 for even numbers between 1 to 10 and multiple of 5 for rest of the numbers.

```
list1 =[]
for n in range(1,11):
if n%2==0:
list1.append(n*6) else:
list1.append(n*5) print(list1)
```

Output:

[5, 12, 15, 24, 25, 36, 35, 48, 45, 60]

Using list comprehensions

```
list1 = [n*6 if n%2==0 else n*5 for n in range(1,11)] print(list1)
```

Output:

[5, 12, 15, 24, 25, 36, 35, 48, 45, 60]

In general list comprehension

[<the_expression> if <the_condition> else <other_expression> for <the_element> in
<the_iterable>]

Comparing this with our example n*6 is **the expression** , n%2==0 is **the condition,** n*5 is **the other expression** , n is **the element** and range(1,11) is **the iterable.**

Applying list comprehension with Nested loops

Now, Suppose we need to multiply n ranging from 1 to 10 with first 1 then 2 and then 3.

list1 =[]

for i in range(1,4):

for j in range(1,11): list1.append(i*j) print(list1)

Output:

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30]

list1 = [i*j for i in range(1,4) for j in range(1,11)] print(list1)

Output:

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30]

In general list comprehension

[<the_expression> for <element_a> in <iterable_a> (optional if <condition_a>)

for <element_b> in <iterable_b> (optional if <condition_b>)

for <element_c> in <iterable_c> (optional if <condition_c>)

... and so on ...]

Comparing this with our example i*j is **the expression** , i is **the element_a** , j is **the element_b** , range(1,4) is **the iterable_a** and range(1,11) is **the iterable_b.**

List Comprehensions in Python

Suppose we need to create a list with first 10 multiple of 6 in it, So we may do this with a normal for loop or with list comprehensions, Let's see both of them and understand the difference.

Normal For loop

```
list1 =[]
for n in range(1,11):
   list1.append(n*6)
print(list1)
```

Output:
[6, 12, 18, 24, 30, 36, 42, 48, 54, 60]

List comprehension

```
list1 = [n*6 for n in range(1,11)]
print(list1)
```

Output:
[6, 12, 18, 24, 30, 36, 42, 48, 54, 60]

We got the same output using list comprehensions just by writing a line of code.

In general list comprehension

[<the_expression> for <the_element> in <the_iterable>]

Comparing this with our example n*6 is **the expression**, n is **the element**, range(1,11) is **the iterable**.

Applying list comprehension with a condition
Now, Suppose we need to create a list of multiple of 6 for just even numbers between 1 to 10.

```
list1 =[]
for n in range(1,11):
   if n%2==0:
      list1.append(n*6)
print(list1)
```

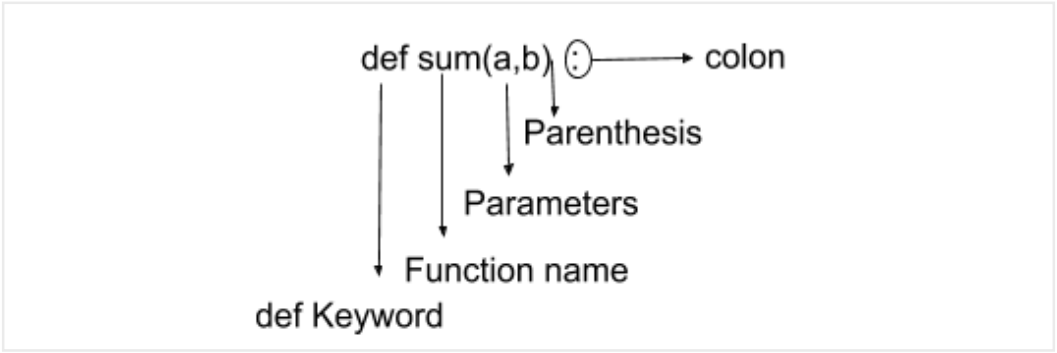
Output:
[12, 24, 36, 48, 60]

Functions in Python:-

In the previous segments, you saw how to take two lists and perform an element-wise subtraction. Now, imagine you are trying to build an app which the general public could use. For such an app, the 'expense calculator function' would be reused. Now instead of writing or copying the code each time you could build it as a function which is similar to methods you have seen earlier. Functions serve the purpose of reusable and customizable methods.

By the end of this segment, you would be able to create your own functions which could be customized to perform a task based on your preference.

Syntax of a function:



As you saw in the 'factorial function' in the earlier video, functions can take multiple parameters. When the functions get complicated, programmers often get confused while passing parameters to a function. Let us now look into more complex examples to build a stronger foundation of functions.

You heard about arguments and parameters a couple of times in the video and might have been feeling a little fuzzy about it, so here is a refresher-Function parameters are the variables used while defining the function, and the values used while calling this function are the function arguments.

There are four types of function parameters:

1. **Required parameters**: These parameters are necessary for the function to execute. While calling an inbuilt max() function, you need to pass a list or dictionary or other data structure. This means that a sequence is a required parameter for this function.
2. **Default arguments**: These parameters have a default value and will not throw any error if they are not passed while using the function.
3. **Keyword parameters**: These parameters are expected to be passed using a keyword. In the example of printing a list, you saw that passing end = ',' prints the elements of the list separated by a comma instead of the default newline format. This is a classic example of a keyword argument, the value of which is changed by using the end keyword and specifying the value.
4. **Variable-length parameters**: These enable the function to accept multiple arguments. Now, let’s try to write a function that takes two

lists as the input and performs an element-wise subtraction. It is simple: first, you define your function with two parameters:

```
def fun(L1,L2):
```

Now, from your previous experience, you know how to perform element-wise subtraction. So, in order to build it into a function for the same, you just need to put the code inside a function definition:

```
def list_diff(list1,list2):
    list3 = []
    for  i in range(0, len(list1)):
        list3.append(list1[i] - list2[i])
    return list3

L1 = [10, 20, 30, 24, 18]
L2 = [8, 14, 15, 20, 10]

print(list_diff(L1, L2))
```

Note: The return statement is very crucial in the whole function definition, as it is responsible for returning the output of the function.

Based on your understanding about functions from the previous videos, attempt the quiz given below:

Lambda functions are another way of defining functions to execute small functionalities occurring while implementing a complex functionality. These functions can take multiple parameters as input but can only execute a single expression; in other words, they can only perform a single operation.

The format of a lambda expression is as follows:

function_name = lambda <space> input_parameters : output_parameters

For example: diff = lambda x,y : x-y is a lambda function to find the difference of two elements.

Eg. of lambda function :-

```
import ast,sys
input_str = sys.stdin.read()
input_list = ast.literal_eval(input_str)
a = int(input_list[0])
b = int(input_list[1])
```

```
#Write your code here
greater = lambda x,y : x if x > y else y
print(greater(a,b))
```

Map, Filter and Reduce Functions:-

Now that we have covered more sophisticated methods like loops and comprehensions, let us also learn more about the map, filter and reduce methods, that offer us sophisticated and faster method implementation. Starting with Map, Map is a function that works like list comprehensions and for loops. It is used when you need to map or implement functions on various elements at the same time.

The syntax of the map function looks as shown below:

map(function,iterable object)

- The function here can be a lambda function or a function object.
- The iterable object can be a string, list, tuple, set or dictionary.

Let's look at an example to understand the map function better:

you are using a map function to create a list from a tuple with each of its elements squared.

```
list_numbers = (1,2,3,4)
sample_map = map(lambda x: x*2, list_numbers)
print(list(sample_map))
```

In the implementation of the map function, the lambda function lambda **x: x*2** would return a lambda object, but the map handles the job of passing each element of the iterable to the lambda object and storing the value obtained in a map object. Using the list function on the map object, you finally obtain the output list.

```
def multi(x):
    return x*2

list_numbers = [1,2,3,4]
sample_map = map(multi, list_numbers)

print(list(sample_map))
```

The difference between the previous code and this above-mentioned code is that instead of applying a lambda function, you can use the function object also.

Now let us look at the **Filter operation**:
'**Filter**' is similar to map function, only distinguishing feature being that it requires the function to look for a condition and then returns only those elements from the collection that satisfy the condition.

The syntax of the filter function looks as shown below:

filter(function,iterable object)

- The function object passed to the filter function should always return a boolean value.
- The iterable object can be a string, list, tuple, set or dictionary.

the filter command was used to create an application that can count the number of students whose age is above 18.

```
students_data = {1:['Sam', 15] , 2:['Rob',18], 3:['Kyle', 16], 4:['Cornor',19], 5:['Trump',20]}

len(list(filter(lambda x : x[1] > 18, students_data.values())))
```

Now, let us take a look at the last function in sequence i.e. the reduce function -

'Reduce' is an operation that breaks down the entire process into pair-wise operations and uses the result from each operation, with the successive element. The syntax of reduce function is given below.

reduce(function,iterable object)

- The function object passed to the reduce function decides what expression is passed to the iterable object.
- The iterable object can be a string, list, tuple, set or dictionary.
- Also, reduce function produces a single output.

Note: One important thing to note is that the reduce function needs to be imported from the 'functools' library.

```
from functools import reduce
```

Let's take a look at an example to understand the reduce function:

```
from functools import reduce
list_1 = ['Paul','Ted']
reduce(lambda x,y: x + y,list_1)
```

In the code snippet given above, you are importing functools library is being imported to access reduce function. In the implementation of the reduce function, the lambda function x,y : x+y, list_1 is appending two objects; remember here if the objects are strings it appends them if the objects are numbers it adds them.
For the example above, the reduce function will convert the list of string into a single list, the output is given below:

'PaulTed'

———————————————————————————————————————

```
import ast,sys
input_str = sys.stdin.read()
input_list = ast.literal_eval(input_str)
from functools import reduce
def greatestNum (x,y):
    if type(x)=='NoneType':
        x=0
    if x<y:
        x=y
answer = reduce(lambda x,y : x if x>y else y,input_list)#Type your answer here.

print(answer)
```

Summary:-

To conclude, in this session you learnt about the various control structures and functions supported by Python like the if-else constructs, nested if constructs, various loop concepts, and then finally learnt about functional programming in Python.

In the next session, you will be learning about object-oriented programing concepts in Python, where you will dive deep into the concepts of classes, objects and object-oriented programming methodologies.