4.4 Keywords and Identifiers

Keywords

- ➢ Keywords are the reserved words in Python and cannot be used as the names of variables, functions, classes and other identifiers.
- ➢ There are 35 keywords in Python and all of them are case sensitive. (Till Python 3.6 version, there were 33 keywords, but two more got added from 3.7 version onwards)
- ➢ Keywords are always available and we need not import them explicitly. We can use them directly in our code whereas the in-built functions, modules have to be imported explicitly before use.
- ➢ The statement import keyword is used to import the **keyword** class. The statement **print(keyword.kwlist)** prints the list of the available keywords in Python.

```
import keyword

#print(keyword.kwlist)
"Total number of keywords ", keyword.kwlist
```

```
('Total number of keywords ',
 ['and',
  'as',
  'assert',
  'break',
  'class',
  'continue',
  'def',
  'del',
  'elif',
  'else',
  'except',
  'exec',
  'finally',
  'for',
  'from',
  'global',
  'if',
  'import',
  'in',
  'is',
  'lambda',
  'not',
  'or',
  'pass',
  'print',
  'raise',
  'return',
  'try',
  'while',
  'with',
  'yield'])
```

Note: We are importing the keyword module, just to check the list of available keywords, which is possible only by accessing the **kwlist()** method of keyword class.

- ➢ If we want to access any method/variable of a class, we first have to import the class.
- ➢ If you want to use any one of these 35 keywords in your code, you need not import the **keyword** module. You can directly use them.

Identifiers

- ➢ Identifier is the name given to the entities like classes, functions, variables, etc. in Python. It helps in differentiating one entity from the other.

Rules for writing an Identifier

1) Identifiers can be a combination of lowercase letters (a-z) or uppercase letters (A-Z) or digits (0-9) or an underscore (_).
2) An identifier cannot begin with a digit, but can contain digits in the middle. (ie., the first character in the identifier should never be a digit and we can use digits anywhere else in the identifier except the first position.)
3) Keywords cannot be used as the identifiers. Below is an example shown at the timestamp 3:43 in the video.

```
: global = 1
  File "<ipython-input-3-d0026cf49b71>", line 1
    global = 1
           ^
SyntaxError: invalid syntax
```

4) Apart from underscore (_), no special character should be used in an identifier. Below is an example shown at the timestamp 5:22 in the video.

We cannot use special symbols like !, @, #, $, % etc. in our identifier.

```
: a@ = 10              #can't use special symbols as an identifier
  File "<ipython-input-5-b512271f00c8>", line 1
    a@ = 10              #can't use special symbols as an identifier
     ^
SyntaxError: invalid syntax
```

4.5 Comments, Indentation and Statements

Comments

- Comments are the lines written in the code and are not executed by the compiler and the interpreter.
- Comments are in general used to describe which part of code is doing what job.
- Comments make the code look readable and understandable by the humans and it is always a good practice to write comments in the code because when we write the code and return back after sometime, we may

not be able to understand and we may not remember the thought process followed while writing the code. Hence writing the comments make the code much more understandable.

- The comments should always begin with the symbol '#' and once if the interpreter/compiler encounters the '#' symbol in the code, the statement from that position, till the end of that line will be ignored and will not be executed.

Multi-line Comments

In case, if we come across a situation where we have to write too lengthy comments, you can split it into more than one line. We have two approaches.
For example, the comment is "**Below given example is the most appropriate one for recursion**".
So the 2 ways to write the comment in multiple lines are

Approach 1

```
# Below given example
# is the most appropriate one
# for recursion
```

Approach 2

```
"""Below given example
is the most appropriate one
for recursion"""
```

Out of both these ways, the first one is mostly used by the programmers.

DocString

- DocString is the shortest description of a function that has to be written as the first line inside the function block. The docstring should always be the first line inside the function.
- The docstring has to be enclosed within triple quotes.

Example

```
def add(a,b):
'''This function performs the addition of two numbers.''' # This is the doc-string.
```

```
    return a+b
```

Syntax to access the doc string of above function

```
print(add.__doc__)
```

Indentation

- ➢ Most of the programming languages like C++, Java use braces to define a block of code. But Python doesn't support the concept of using braces to define code blocks. In place of these braces, it supports Indentation (ie., whitespaces).
- ➢ If we find any line of code beginning with whitespaces, then it means a new block of code has begun and also this new block ends with the first unindented line.
- ➢ Generally one tab space or four white spaces are considered for indentation.

Example

```
a = 15
b = 8
if a>b:
    print('a is greater')
else:
    print('b is greater)
```

In the above example, as the **print()** begins with whitespaces, it means from there onwards a new block begins and the first unintended line is the **'else'** statement.

Example

```
for i in range(10,100):
    if i%10==0:
        print('{} is divisible by 10'.format(i))
print('Done with looping')
```

In the above example, we could see a block within another block and the first unindented line is the line with the 2nd print statement. If we miss any of these indentations, it throws an error.

Statements

- ➢ Python Statements are the instructions passed to the Python interpreter for execution.
- ➢ Comments do not come under the category of statements as they are not executed by the Python interpreter.

Example

a = 25
b = 10

These two above initializations are the statements and when we pass these instructions to the Python interpreter, then the two variables ‘a’ and ‘b’ get created in the memory.

Example

c = a + b
d = a – b

The above 2 statements perform addition and subtraction of two numbers respectively.

Multi-line Statements

We can write a single statement in multiple lines.

Example

**a = 1 + 2 + 3 + **
**4 + 5 + 6 + **
7 + 8 + 9

Here it is one statement, but written in 3 lines. Whenever we use this kind of syntax, we have to use the ‘\’ symbol to let the interpreter understand that the given statement is written in multiple lines.

b = (1 + 2 + 3 +
4 + 5 + 6 +
7 + 8 + 9)

Here the one statement is written in 2 lines. The other way to write the same code without using the '\' symbol is to enclose all the numbers in small braces '()'.

4.6 Variables and Datatypes in Python

Variables

- A Variable is a location in the memory and is used to store some value.
- Each variable has a unique name so that it could be differentiated from the other memory locations. The rules for naming a variable are same as that of an identifier.
- While initializing a variable, we just have to assign a value. Unlike in programming languages like C/C++/Java, we need not mention any data type for the variable. Deciding the data type and allocation of memory is done internally, after we assign a value.

Variable Assignment

We use the '=' operator to assign values to a variable. The below example was discussed at the timestamp 0:55 in the video.

```
: #We use the assignment operator (=) to assign values to a variable

  a = 10
  b = 5.5
  c = "ML"
```

Example in Java/C
int a = 10
float b = 5.5

In the first statement, we are declaring the datatype of the variable 'a' as 'integer' and assigning the integer 10 to it.
Similarly in the second statement, we are declaring the datatype of the variable 'b' as 'float' and assigning the decimal value 5.5 to it.

But when we take the same example in Python, the statement would be
a = 10
b = 5.5
c = 'ML'

In the first statement, after assigning the value 10 to the variable ‘a’, then it comes to know that ‘a’ is an integer variable.

Similarly after assigning the value 5.5 to the variable ‘b’, it comes to know that the variable ‘b’ is a float variable.

In the third statement, after assigning the value ‘ML’ to the variable ‘c’, it knows that the given variable is of string type and then allocates the memory accordingly.

Multiple Variable Assignments

Below is an example that has been discussed at the timestamp 2:40 in the video.

```
a, b, c = 10, 5.5, "ML"
```

```
a = b = c = "AI" #assign the same value to multiple variables at once
```

The same previous example can be written as an example for multiple variable assignments as

a, b, c = 10, 5.5, ’ML’

The functionality is the same in both these types of variable assignments, but the main advantage with this multiple variable assignment approach is the **reduction in the length of the code**.

If we want to assign same value to multiple variables, then the syntax would be

a = b = c = ‘AI’

Storage Locations

- The storage locations are the locations where the variables are stored. Each variable has a different and a unique storage location.
- Once after we assign a value to a variable, then immediately Python allocates a storage location to that variable. The storage location address is printed using the **id()** function in Python. Below is an example that has been discussed at the timestamp 3:30 in the video

```
|: x = 3

   print(id(x))                #print address of variable x
```

```
140372891159288
```

```
|: y = 3

   print(id(y))                #print address of variable y
```

```
140372891159288
```

Observation:

x and y points to same memory location

```
|: y = 2
   print(id(y))                #print address of variable y
```

```
140372891159312
```

Example

x = 3

y = 3

print(id(x)) # prints the address of the variable ‘x’

o/p: 140372891159288

print(id(y)) # prints the address of the variable ‘y’

o/p: 140372891159288

Here both ‘x’ and ‘y’ are pointing to the same location. Hence we got the same value.

Note: For integer data, if you declare two or more variables and assign any one value in the interval [-5,256] to all the variables, then all those variables will point to the same location.

If any value outside the interval [-5,256] is chosen and is assigned, then they all point to different locations, as each of those variables will have unique locations.

Example

x = 3

y = 2

print(id(x)) # prints the address of the variable ‘x’

o/p: 140372891159288

print(id(y)) # prints the address of the variable ‘y’

o/p: 140372891159312

Here as both ‘x’ and ‘y’ are having two different values, they both point to two different locations.

Data Types

- ➢ Every value in Python has a data type.
- ➢ Since Python is an object oriented programming language, every data type is a predefined class in Python and every variable is an instance of these predefined classes.

Numbers

- ➢ Integers, floating point numbers and complex numbers fall under the Python numbers category.
- ➢ These numbers are classified into int, float and complex classes respectively in Python.
- ➢ We have to use type() function to know which class a variable/value belongs to.
- ➢ We have to use isinstance() method to check if a given object belongs to a particular class or not.

Below is an example that has been discussed at the timestamp 6:15 in the video.

```
a = 5                                   #data type is implicitly set to integer
print(a, " is of type", type(a))
```

```
(5, ' is of type', <type 'int'>)
```

```
a = 2.5                                 #data type is changed to float
print(a, " is of type", type(a))
```

```
(2.5, ' is of type', <type 'float'>)
```

```
a = 1 + 2j                              #data type is changed to complex number
print(a, " is complex number?")
print(isinstance(1+2j, complex))
```

```
((1+2j), ' is complex number?')
True
```

Boolean Data Type

- ➢ Boolean data type represents the truth values **True** and **False**.
- ➢ It doesn’t take any value other than these two.

- ➢ Also **True** and **False** are not only the values of boolean data type, but also are the keywords in Python.

Example
a = True
b = False
print(type(a)) # prints the data type of the variable 'a'
o/p: <type 'bool'>
print(type(b)) # prints the data type of the variable 'b'
o/p: <type 'bool'>

Strings

- ➢ String is a sequence of Unicode characters.
- ➢ A string has to be enclosed either between single quotes or double quotes.
- ➢ Multi-line strings have to be enclosed between triple quotes. You can either use ''' (or) """.
- ➢ A string in Python consists of a series or a sequence of characters. It can have characters, numbers and special characters.
- ➢ Strings in Python also support indexing and we can access each character using an index. The first index would be 0.

Below is an example that has been discussed at the timestamp 9:30 in the video.

```
s = "This is Online AI course"
print(s)
```

```
This is Online AI course
```

```
print(s[0])
#last char s[len(s)-1] or s[-1]
```

```
T
```

```
#slicing
s[5:]
```

```
'is Online AI course'
```

Example

s = “This is Online AI course”

print(s[0]) # Prints the character present in 0th index

print(s[5:]) # Prints the substring starting from the 5th index till the end.

print(s[5:10]) # Prints the substring starting from the 5th index till the 9th index.

Python List

- ➢ List is an ordered sequence of items.
- ➢ It is one of the most used data types in Python and is very flexible.
- ➢ All the items in a list do not need to be of the same data type.
- ➢ Declaring a list is nothing but the items separated by commas are enclosed within brackets [ ]
- ➢ We can access each of the elements in the list using their indexes.
- ➢ Lists are mutable. It means the elements in the list can be changed as per our requirement whenever needed.

Below are the examples that are discussed in the video at the timestamp 15:45

```
a = [10, 20.5, "Hello"]
print(a[1])              #print 1st index element
```

```
20.5
```

Lists are mutable, meaning, value of elements of a list can be altered.

```
a[1] = 30.7
print(a)
```

```
[10, 30.7, 'Hello']
```

In the first example, we can see how we are able to access the elements of the list using their indexes.

In the second example, we can see how we can manipulate/modify the values in the list.

Python Tuple

- Tuple is an ordered sequence of elements, the same as a List.
- The only difference between a List and a Tuple is the List is mutable whereas a Tuple is immutable. It means once if a tuple is created, then the elements in it could not be modified.
- Also the elements in a List are enclosed within square brackets (ie., []) whereas the elements in a tuple should be enclosed between a pair of round brackets (ie., ())

Below is an example that has been discussed at the timestamp 18:15 in the video.

```
: t = (1, 1.5, "ML")

: print(t[1]) #extract particular element
1.5

: t[1] = 1.25
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-18-0ddc671fae60> in <module>()
----> 1 t[1] = 1.25

TypeError: 'tuple' object does not support item assignment
```

Python Set

- Set is an unordered collection of unique items.
- A set is defined by the values separated by command inside the curly braces (ie., {})
- Items in a set are unordered. The order of the elements in a set is decided by an internal mechanism.
- Sets do not allow repetition of elements and also sometimes the order in which the elements are present in the set might differ from the order in which we have inserted.
- We can perform operations like subtraction, union and intersection between two sets.

Below is an example that has been discussed at the timestamp 20:40 in the video.

```
a = {10, 30, 20, 40, 5}
print(a)
```

```
set([40, 10, 20, 5, 30])
```

```
print(type(a))              #print type of a
```

```
<type 'set'>
```

```
s = {10, 20, 20, 30, 30, 30}
print(s)                    #automatically set won't consider duplicate elements
```

```
{10, 20, 30}
```

```
print(s[1]) #we can't print particular element in set because
            #it's unorder collections of items
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-23-ad7511dba6cd> in <module>()
----> 1 print(s[1]) #we can't print particular element in set because
      2             #it's unorder collections of items

TypeError: 'set' object does not support indexing
```

In the above example, the first cell clearly shows us that the order in which the elements are inserted is different from the order in which they are displayed.

The third cell clearly shows us that a set doesn't allow repetition of elements. It means irrespective of how many times an element occurs, it will be stored only once in the set.

The fourth cell clearly shows us that a set object doesn't support indexing.

Python Dictionary

- Dictionary is an unordered collection of key-value pairs.
- Same like a set, the elements in a dictionary are also enclosed between curly braces, but the major difference here is the elements in a dictionary are present in **key-value pair** format.
- The order in which the key-value pairs are added to a dictionary is sometimes different from the order in which they appear.
- Dictionary doesn't allow duplication of keys, but allows duplication of values. It means a key once used should not be used again for another value, whereas the value once used can be used again for another key.

➢ The term present to the left side of the colon(:) is the **key** and the term present to the right side of the colon(:) is the value. There should be no keys repeated.

Below is an example discussed at the timestamp 24:25 in the video.

```
|: d = {'a': "apple", 'b': "bat"}
   print d['a']

    apple
```

```
: d = {'a': "apple", 'b': "bat"}
  print(d['c'])

  ---------------------------------------------------------------------------
  --
  KeyError                                  Traceback (most recent call las
  t)
  <ipython-input-60-744a428435ea> in <module>()
        1 d = {'a': "apple", 'b': "bat"}
  ----> 2 print(d['c'])

  KeyError: 'c'
```

If the entered **'key'** is not present in the dictionary, then it throws an error as shown above.

Conversion between Data Types

We can perform conversions from one data type to another using different available type casting functions like int(), float(), str(), etc.

Below are the examples that were discussed at the timestamp 26:56

```
float(5)     #convert interger to float using float() method
```

```
5.0
```

```
int(100.5)   #convert float to integer using int() method
```

```
100
```

```
str(20)      #convert integer to string
```

```
'20'
```

Conversion to and from string must contain compatible values.

```
: int('10p')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-27-b6ad1e4c556d> in <module>()
----> 1 int('10p')

ValueError: invalid literal for int() with base 10: '10p'
```

```
: user = "satish"
lines = 100

print("Congratulations, " + user + "! You just wrote " + str(lines) + " lines of code" )
#remove str and gives error
```

```
Congratulations, satish! You just wrote 100 lines of code
```

We can convert one sequence to other

```
: a = [1, 2, 3]

print(type(a))      #type of a is list

s = set(a)          #convert list to set using set() method

print(type(s))      #now type of s is set
```

```
<type 'list'>
<type 'set'>
```

```
: list("Hello")      #convert String to list using list() method
```

```
: ['H', 'e', 'l', 'l', 'o']
```

4.7 Standard Input and Output

Python Output

In order to display the result, we use the **print()** function.

Below are the examples starting from the timestamp 0:10 in the video.

```
: print("Hello World")
```

```
Hello World
```

```
: a = 10
  print("The value of a is", a) #python 3
  print "The value of a is " + str(a)
```

```
('The value of a is', 10)
The value of a is 10
```

Both the above print statements give the same result, but in the first one, we are printing a string and an integer side by side whereas in the second statement we are converting the integer into a string and then concatenating both the strings and are printing the result.

Output Formatting

The below examples begin from the timestamp 1:55 in the video.

```
: a = 10; b = 20 #multiple statements in single line.

  print("The value of a is {} and b is {}".format(a, b))    #default
```

```
The value of a is 10 and b is 20
```

The above statement prints the string with the values of ‘a’ and ‘b’ in place of those braces in the same order as they are present in **format()** method.

```
a = 10; b = 20  #multiple statements in single line

print("The value of b is {1} and a is {0}".format(a, b)) #specify position of arguments
```

```
The value of b is 20 and a is 10
```

In this statement, we are displaying the same result, but here instead we are using the indexes of the values in **format()** method. The index of 'a' is 1 and the index of 'b' is 0. Values associated with those variables present in those indexes will be displayed in place of these braces.

```
#we can use keyword arguments to format the string
print("Hello {name}, {greeting}".format(name="satish", greeting="Good Morning"))
```

```
Hello satish, Good Morning
```

In this statement, as we are passing the keyword arguments in **format()** method, we have to pass the keys into these braces. In our example, the keys are 'name' and 'greeting'.

```
#we can combine positional arguments with keyword arguments
print('The story of {0}, {1}, and {other}'.format('Bill', 'Manfred',
                                                  other='Georg'))
```

```
The story of Bill, Manfred, and Georg
```

In this statement, we are passing positional as well as keyword arguments to the **format()** function. One thing to be remembered seriously here is that the keyword arguments should always come after the positional arguments in **format()** method.

Python Input

We have the **input()** function in Python which is used to take the input from the user.
The below example was discussed at the timestamp 5:25 in the video.

```
num = input("Enter a number: ")
print num
```

```
Enter a number: 10
10
```

4.8 Operators

- ➢ Operators are the special symbols in Python used for performing mathematical and logical computations.
- ➢ The numbers on which the operations are performed are called **operands**

Types of Operators

1) Arithmetic Operators
2) Relational Operators (Comparison Operators)
3) Logical Operators (Boolean Operators)
4) Bitwise Operators
5) Assignment Operators
6) Special Operators

Arithmetic Operators

Arithmetic Operators are used to perform operations like addition, subtraction, multiplication, division, floor division, modulo division, exponent, etc.

Below example begins at the timestamp 0:40.

```
]: x, y = 10, 20

   #addition
   print(x + y)

   #subtraction(-)

   #multiplication(*)

   #division(/)

   #modulo division (%)

   #Floor Division (//)

   #Exponent (**)
```

```
30
```

Example

x, y = 10 ,20
print(x + y) # Prints the sum.
print(x - y) # Prints the difference.
print(x * y) # Prints the product.
print(x / y) # Prints the division.
print(x % y) # Prints the modulo division.
print(x // y) # Prints the floor division.
print(x ** y) # Prints the exponent value.

Comparison Operators (Relational Operators)

- ➢ Comparison/Relational operators are used to compare two given values. They return only Boolean values. (i.e., either **True** or **False**)
- ➢ >, <, ==, !=, >=, <= are comparison operators.

Below example begins at the timestamp 4:35 in the video.

```
: a, b = 10, 20

  print(a < b)  #check a is less than b

  #check a is greater than b

  #check a is equal to b

  #check a is not equal to b (!=)

  #check a greater than or equal to b

  #check a less than or equal to b

  True
```

Example

a, b = 10, 20
print(a<b) # Prints True if a is less than b. Otherwise it prints False.
print(a>b) # Prints True if a is greater than b. Otherwise it prints False.
print(a==b) # Prints True if a is equal to b. Otherwise it prints False.
print(a!=b) # Prints True if a is not equal to b. Otherwise it prints False.

print(a>=b) # Prints True if a is greater than or equal to b. Otherwise it prints False.
print(a<=b) # Prints True if a is less than or equal to b. Otherwise it prints False.

Logical Operators

Logical Operators are **AND**, **OR** and **NOT** operators. Even these operators give only Boolean values as the result.

The below examples begin at timestamp 5:15.

```
: a, b = True, False

#print a and b
print(a and b)

#print a or b

#print not b
```

```
False
```

Example

a, b = True, False
print(a and b)
print(a or b)
print(not b)

Bitwise Operators

- Bitwise Operators are those operators which act on the operands as if the operands are the strings of binary digits.
- These operators operate bit by bit.
- The bitwise operators are &, |, ~, ^, >>, <<

Types of Bitwise Operations

1) Bitwise AND (&)
2) Bitwise OR (|)
3) Bitwise NOT (~)
4) Bitwise XOR (^)
5) Bitwise Right Shift (>>)
6) Bitwise Left Shift (<<)

```
a, b = 10, 4

#Bitwise AND
print(a & b)

#Bitwise OR


#Bitwise NOT


#Bitwise XOR


#Bitwise rightshift


#Bitwise Leftshift
```

```
0
```

The above example begins at the timestamp 6:10

Example

```
a,b = 10,4
print(a&b)  # Bitwise AND
print(a|b)  # Bitwise OR
print(~b)   # Bitwise NOT
print(a^b)  # Bitwise XOR
print(a>>2) # Bitwise RIGHT SHIFT
print(a<<3) # Bitwise LEFT SHIFT
```

Assignment Operators

Assignment operators in python are used to assign values to variables.

Types of Assignment Operations

1) Simple Assignment (=)
2) Addition AND (+=)
3) Subtraction AND (-=)
4) Multiplication AND (*=)
5) Division AND (/=)
6) Modulo Division AND (%=)
7) Exponent AND (**=)

Below example was discussed at the timestamp 8:45 in the video.

```
: a = 10

a += 10          #add AND
print(a)

#subtract AND (-=)

#Multiply AND (*=)

#Divide AND (/=)

#Modulus AND (%=)

#Floor Division (//=)

#Exponent AND (**=)
```

```
20
```

Special Operators

A. Identity Operators

Identity operators are used to check if the given two operands are pointing to the same location.

There are only two identity operators in Python. They are **is** and **is not**.

Below examples have been discussed starting from the timestamp 9:55 in the video.

```
a = 5
b = 5
print(a is b)    #5 is object created once both a and b points to same object

#check is not
```

```
True
```

In the above example, the '**is**' operator is checking if both 'a' and 'b' are pointing to the same object. As they both are pointing to the same object, it is giving the result as **True**.

```
l1 = [1, 2, 3]
l2 = [1, 2, 3]
print(l1 is l2)
```

```
False
```

If two lists have the same elements in the same order, it doesn't mean they both are pointing to the same object. Hence it is giving the result here as **False**.

```
s1 = "Satish"
s2 = "Satish"
print(s1 is not s2)
```

```
False
```

When two strings have the same value and if there are no whitespaces in the strings, then they both point to the same object. Otherwise, they both point to two different objects.

B. Membership Operators

- Membership operators are used to check if a given object is a member of a given data structure or not.
- The two membership operators in Python are **in** and **not in**.

```
lst = [1, 2, 3, 4]
print(1 in lst)       #check 1 is present in a given list or not

#check 5 is present in a given list
```

```
True
```

In the above example, it is checking if the number 1 is present in the list 'lst'. As it is present, the result became **True**.

In case of lists, tuples, sets it checks if the given element is present in those data structures or not. Whereas in case of dictionaries, it checks if the given element/value is present as the key in it. If yes, then it returns **True**. Otherwise it returns **False**.

```
d = {1: "a", 2: "b"}
print(1 in d)
```

```
True
```

In this example, as the number 1 is present as one of the keys in the dictionary 'd', it returns **True**.

4.9 Control Flow: if else

Python if-else statement

The **if-elif-else** statements are used in python for decision making.

Syntax

if <test expression>:
statements(s)

If the given test expression gives the result as **True**, then only the statement(s) associated with this block are executed.

If the given test expression gives the result as **False**, then the statement(s) associated with this block are not executed.

If the test expression gives any non boolean value as the result, then the values **0** and **None** are interpreted as **False** and all other non zero and non boolean values are interpreted as **True.**

Below example was discussed from the timestamp 3:25 in the video.

```
: num = 10
if num > 0:
    print("Positive number")
else:
    print("Negative Number")
```

```
Positive number
```

In the above example, we are checking if the given number is greater than 0 or not. If yes, then it prints the statement in the '**if**' block. Otherwise, it prints the statement in the **'else'** block.

if-elif-else statement

Syntax

if <test-expression-1>:
    statement(s)

elif <test-expression-2>:
    statement(s)

else:
    statement(s)

Below is an example discussed at the timestamp 4:15 in the video.

```
num = 0

if num > 0:
    print("Positive number")
elif num == 0:
    print("ZERO")
else:
    print("Negative Number")
```

```
ZERO
```

In the above example, if the given 'num' is greater than 0, then the statement in the '**if**' block gets executed. If 'num' is equal to 0, then the statement in the **'elif'** block gets executed. If both the conditions fail, then the statement in the **'else'** block gets executed.

```
num = 10.5

if num >= 0:
    if num == 0:
        print("Zero")
    else:
        print("Positive number")
else:
    print("Negative Number")
```

```
Positive number
```

```
num1 = 10
num2 = 50
num3 = 15

if (num1 >= num2) and (num1 >= num3):           #logical operator   and
    largest = num1
elif (num2 >= num1) and (num2 >= num3):
    largest = num2
else:
    largest = num3
print("Largest element among three numbers is: {}".format(largest))
```

```
Largest element among three numbers is: 50
```

In the above program, we are finding out the largest number among the given 3 numbers.
Here we are initializing num1 = 10, num2 = 50, num3 = 15.
If ‘num1’ is greater than or equal to ‘num2’ and if ‘num1’ is greater than or equal to ‘num3’, then the statement in the ‘**if**’ block gets executed.
If ‘num2’ is greater than or equal to ‘num1’ and if ‘num2’ is greater than or equal to ‘num3’, then the statement in the ‘**else**’ block gets executed. Otherwise the ‘**else**’ block statement will get executed.

4.10 Control Flow: while

The while loop in Python is used to iterate over a block of code as long as the test expression is true.

Syntax

while test-expression:
Body of while

The 'Body of while' get executed only if the test-expression is evaluated as True. After one iteration, the test-expression is again checked and if it again returns **True**, then again the same 'Body of while' will get executed. This looping continues until the test-expression returns **False**.

Below is an example that is shown at timestamp 1:36 in the video.

```
#Find product of all numbers present in a list

lst = [10, 20, 30, 40, 60]

product = 1
index = 0

while index < len(lst):
    product *= lst[index]
    index += 1

print("Product is: {}".format(product))
```

```
Product is: 14400000
```

In this example, we want to compute the product of all the values in the given list. We are traversing through the list with the help of indexes.

So we are starting the index from 0 and are initializing the variable 'product' to 1 which is used to store the result and are checking if the index is less than the length of the list. If the condition is satisfied, we are the element at that index position is multiplied to the variable 'product' and we are incrementing the index by 1.

This process keeps repeating until the test condition fails and finally we are printing the result which is the product of all the numbers in the given list.

while loop with else

We can have an optional '**else'** block for a '**while'** loop. This '**else'** block gets executed only if the test condition in the while loop returns **False**.

We also can break the while loop using '**break'** statement in the body of the loop, but when this '**break'** statement gets executed, then the control comes out of the loop ignoring this '**else'** block.

So the '**else'** block of a '**while'** loop runs only if the test expression is returning **False** and if there is no '**break'** statement.

Below is an example shown at the timestamp 6:30 in the given video

```
numbers = [1, 2, 3,4,5]

#iterating over the list
index = 0
while index < len(numbers):
    print(numbers[index])
    index += 1

else:
    print("no item left in the list")
```

In the above example, we are incrementing the 'index' variable after every iteration. The length of the given list is 5 and once if the value of the 'index' variable becomes 5, then the condition index<len(numbers) fails and the control comes out of the loop and then executes the 'else' block.

Note: If we forget to increment the 'index' variable in each iteration, then it runs into infinite looping.

Below is an example of checking whether a given number is a prime. This is explained at timestamp 9:50 in the given video.

```
num = int(input("Enter a number: "))        #convert string to int


isDivisible = False;

i=2;
while i < num:
    if num % i == 0:
        isDivisible = True;
        print ("{} is divisible by {}".format(num,i) )
    i += 1;

if isDivisible:
    print("{} is NOT a Prime number".format(num))
else:
    print("{} is a Prime number".format(num))
```

```
Enter a number: 33
33 is divisible by 3
33 is divisible by 11
33 is NOT a Prime number
```

Procedure:

In the above example, for a number to be a prime, it should not be divisible by any other number except 1 and itself. So if we assume the given number is 'N', then in the range 1 to N, 1 is the first value and 'N' is the last value. We have to check if any value in between these two values (i.e., >1 and <N), can divide 'N' leaving a remainder 0. If we are able to find any such value, then we have to declare the number 'N' as a non-prime, otherwise, we can declare it as a prime.

We are using an indicator variable '**isDivisible'** which is boolean to indicate whether the number is a prime or not.

First we are initializing the '**isDivisible'** variable to **False** before beginning the iteration process. In the iteration process, if we find any value (say 'i') where N%i==0, then immediately we have to change the value of '**isDivisible'** variable to **True**.

After the completion of the iteration process, if the value of '**isDivisible'** remains **False**, then it means there is no value in the range that could divide the given number leaving a remainder of 0 and hence we can declare the number as a prime. Otherwise we have to declare it as a non-prime.

4.11 Control Flow: for

- ➢ The for loop in Python is used to iterate over sequences (like lists, tuples, etc) or other iterable objects.
- ➢ Iterating over a sequence is called a **Traversal**.

Syntax

for element in sequence:

Body of for

Here, element is the variable that takes the value of the item inside the sequence on each iteration. Loop continues until we reach the last item in the sequence.

Below example's explanation begins at the timestamp 0:55 in the video.

```
#Find product of all numbers present in a list

lst = [10, 20, 30, 40, 50]

product = 1
#iterating over the list
for ele in lst:
    print(type(ele))
    product *= ele

print("Product is: {}".format(product))
```

```
<class 'int'>
<class 'int'>
<class 'int'>
<class 'int'>
<class 'int'>
Product is: 12000000
```

In the above example, the variable 'ele' denotes each element in the iterable 'lst' in every iteration, till the end of looping process .

range() function

- ➢ We can generate a sequence of numbers using the range() function.
- ➢ For example, range(10) will generate numbers from 0 to 9 (10 numbers).
- ➢ We can also define the start, stop and step size as range(start,stop,step size). step size defaults to 1 if not provided.

- ➢ This function does not store all the values in memory, it would be inefficient. So it remembers the start, stop, step size and generates the next number on the go.

Below examples are discussed in the video starting from the timestamp 3:50 onwards.

```
#print range of 10
for i in range(10):
    print(i)
```

```
0
1
2
3
4
5
6
7
8
9
```

This above code snippet is an example for iterating over a sequence of elements without storing them. Here as we haven't mentioned any 'step-size', the default step size would be 1.

```
#print range of numbers from 1 to 20 with step size of 2
for i in range(0, 20, 5):
    print(i)
```

```
0
5
10
15
```

This code snippet is an example of iterating over a sequence generated using range() function with a step size of 5.

For loop with else

- ➢ A **for** loop can have an optional **else** block as well.
- ➢ The else part is executed if the items in the sequence used in for loop exhausts.

➢ **break** statement can be used to stop a for loop. In such case, the **else** part is ignored.Hence, a for loop's else part runs if no break occurs.

The below example was discussed from the timestamp 9:50.

```
: numbers = [1, 2, 3]

  #iterating over the list
  for item in numbers:
      print(item)
  else:
      print("no item left in the list")
```

```
1
2
3
no item left in the list
```

```
: for item in numbers:
      print(item)
      if item % 2 == 0:
          break
  else:
      print("no item left in the list")
```

```
1
2
```

The first code snippet is an example of the scenario where there is no **break** statement and the iteration over the entire list is over and hence the **else** block got executed.
The second code snippet is an example of the scenario where we encounter a **break** statement and the iteration process gets terminated without completion. Hence the **else** block doesn't get executed.

Below is an example of the program to check if a given number is a prime using **for** loops only. The timestamp for the explanation of this code is 10:35.

```
: index1 = 20
  index2 = 50

  print("Prime numbers between {0} and {1} are :".format(index1, index2))

  for num in range(index1, index2+1):      #default step size is 1
      if num > 1:
          isDivisible = False;
          for index in range(2, num):
              if num % index == 0:
                  isDivisible = True;
          if not isDivisible:
              print(num);
```

```
Prime numbers between 20 and 50 are :
23
29
31
37
41
43
47
```

4.12 Control Flow: break and continue

In Python, the **break** and **continue** statements can alter the flow of any loop.

Loops iterate over a block of code until the test expression is false, but sometimes we wish to terminate the current iteration or even the whole loop without checking the test expression. The **break** and **continue** statements are used in these cases.

The below example was explained in the video starting from the timestamp 1:28

```
numbers = [1, 2, 3, 4]
for num in numbers:          #iterating over list
    if num == 4:
        break
    print(num)
else:
    print("in the else-block")
print("Outside of for loop")
```

```
1
2
3
Outside of for loop
```

In this code snippet, we see the loop running until the value of 'num' becomes equal to 4. Once after it reaches 4, then the condition **num==4** returns **True** and thereby the break statement gets executed and the control comes out of the loop and executes the statements outside this block.

Example for usage of break statement in Prime number program

```
num = int(input("Enter a number: "))      #convert string to int


isDivisible = False;

i=2;
while i < num:
    if num % i == 0:
        isDivisible = True;
        print ("{} is divisible by {}".format(num,i) )
        break; # this line is the only addition.
    i += 1;

if isDivisible:
    print("{} is NOT a Prime number".format(num))
else:
    print("{} is a Prime number".format(num))
```

```
Enter a number: 16
16 is divisible by 2
16 is NOT a Prime number
```

continue statement

The continue statement deals with not performing any operation, just by allowing the next iteration to happen.

Generally while performing looping, for certain values/indexes if we do not want to perform any operation and just to allow the next iteration to take place, we use the **continue** statement.

```
#print odd numbers present in a list
numbers = [1, 2, 3, 4, 5]

for num in numbers:
    if num % 2 == 0:
        continue
    print(num)
else:
    print("else-block")
```

```
1
3
5
else-block
```

In the above code snippet, we want to print only the odd numbers. So while looping over an iterator object(here it is a list), if the element is odd, then we are printing it. Otherwise we are just going forward for the next iteration without performing any additional task using the **continue** statement.