UNIT 10 DATA STRUCTURES AND CONTROL STATEMENTS IN PYTHON

Structure

10.1 Introduction
10.2 Objectives
10.3 Identifiers and Keywords
10.4 Statements and Expressions
10.4 Variables
10.5 Operators
10.6 Data Types
10.7 Data Structures
10.8 Control Flow Statements
10.9 Summary

10.1 INTRODUCTION

A **Python**is a general-purpose, high level programming language which is widely used by programmers. It was created by Guido Van Rossum in 1991 and further developed by the Python Software Foundation. It was designed with an emphasis on code readability, and its syntax allows programmers to express their concepts in fewer lines of code. To learn how to code in any language, it is, therefore, very important to learn its basic components. This is what is the objective of this chapter. In this chapter, we will learn about the basic constituents of Python starting from its identifiers, variables, operators andalso about how to combine them to form expressions and statements. We will also study about the data types available in Python along with the control flow statements.

10.2 OBJECTIVES

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

- Understand the basic building blocks of Python programming Language
- Understand various Data Structures
- Understand usage of control flow statements

10.3 IDENTIFIERS AND KEYWORDS

An **identifier** is a name given to a variable, function, class or module. Identifiers may be one or more characters in the following format:

- Identifiers can be a combination of letters in lowercase (a to z) or uppercase (A to Z) or digits (0 to 9) or an underscore (_). Names like myIndia, other_1 and mca_course, all are valid examples. A Python identifier can begin with an uppercase or a lowercase alphabet or (_).
- An identifier cannot start with a digit but is allowed everywhere else. 1plus is invalid, but plus1 is perfectly fine.
- Keywords (listed in TABLE1) cannot be used as identifiers.
- One cannot use spaces and special symbols like !, @, #, $, % etc. as identifiers.
- Identifier can be of any length. (However, it is preferrd to keep it short and meaningful).

Examples of valid identifiers are:marksPython, Course, MCA301 etc.

Keywords are a list of reserved words that have predefined meaning to the Python interpreter.These are special vocabulary and cannot be usd by programmers as identifiers for variables, functions, constants or with any identifier name. Attempting to use a keyword as an identifier name will cause an error. As Python is case sensitive, keywords must be written exactly as given in TABLE1.

TABLE 1 :List of keywords in Python

and	async	not
assert	finally	or
break	for	pass
class	from	nonlocal
continue	global	raise
def	import	try
elif	in	while
else	is	with
except	lambda	yield
False	True	None
del	if	Return
As	await	

Check your progress - 1:

Q1. Is Python case sensitive when dealing with Identifiers?

a) Yes
b) No

Q2. What is the maximum possible length of an identifier?

a) 32 characters
b) 64 characters
c) 76 characters
d) None of the above

Q3. All keywords in Python are in _____
a) lower case
b) UPPER CASE
c) Capitalized
d) None of the above

10.4 STATEMENTS AND EXPRESSIONS

A **statement** is a unit of code that the Python interpreter can execute. We have seen two kinds of statements: print being an expression statement and assignment.When you type a statement in interactive mode, the interpreter executes it and displays the result, if there is one.A script usually contains a sequence of statements. If there is more than one statement, the results appear one at a time as the statement executes.
For example, the script

```
Print(1)
x = 2
print(x)
```

produces the output

```
1
2
```

The assignment statement produces no output.

An **expression** is an arrangement of values and operators which are evaluated to make a new value. Expressions are statements as well. A value is the representation of some entity like a letter or a number that can be manipulated by a program. A single value **25** or a single variable **x** or a combination of a variable, operator and value **z + 25** are all examples of expressions. An expression, when used in interactive mode is evaluated by the interpreter and result is displayed instantly. For example,

```
In[]: 7 + 3
Out[]: 10
```

But in script, an expression all by itself doesn't show any output altogether. You need to explicitly print the result.

Check your progress-2 :

Q4. What is the output of the following Python expression?: 36 / 4
a) 9.0
b) 9

Q5. What is the output of following statement? : y = 3.14

a) 3.14
b) y
c) None of the above

Q6. An expression can contain:

a) Values
b) Variables
c) Operators
d) All of the above
e) None of the above

Q7. In thePython statement x = a + 5 – b:
a and b are _____
a + 5 – b is _____

a) Operands, an equation
b) Operands, an expression
c) Terms, a group
d) Operators, a statement

10.5 VARIABLES

A**variable** is a named placeholder to hold any type of data which the program can use to assign and modify during the course of execution. Python is a Dynamically Typed Language. There is no need to declare a variable explicitly by specifying whether the variable is an integer or a float or any other type. To define a new variable in Python, we simply assign a value to a name.

Variable names can be arbitrarily long. They can contain both letters and numbers, but they cannot start with a number. It is legal to use uppercase letter; but it is a good idea to begin variable names with a lowercase letter. Variable names are case-sensitive. E.g., IGNOU and Ignou are different variables.

The underscore character can appear in a name. it is often used in names with multiple words, such as my_name or marks_in_maths. Variable names can start with an underscore character, but we generally avoid doing this unless we are writing library code for others to use.

The general format for assigning values to variables is as follows:

variable_name = expression

The equal sign (=) also known as simple assignment operator is used to assign values to variables. In the general format, the operand to the left of the = operator is the name of the variable and the operand to the right of the = operator is the expression which can be a value or any code snippet that results in a value. That value is stored in the variable on the execution of the assignment statement. Assignment operator should not be confused with the = used in algebra to denote equality. E.g.,

In[]: **number = 100** #integer type value is assigned to a variable *number*

In[]: **name = "Python"** #string type value is assigned to variable *name*

In Python, not only the value of a variable may change during program execution but also the type of data that is assigned. You can assign an integer value to a variable, use it as an integer for a while and then assign a string to the variable. A new assignment overrides any previous assignments.

In[]: **year = 2020** #an integer value is assigned to *year* variable

In[]: **year**

Out[]: **2020**

In[]:**year ="twenty twenty"** #then an string value is assigned to *year* variable

In[]: **year**

Out[]: **'twenty twenty'**

Python allows you to assign a single value to several variables simultaneously. E.g.

In[]: **x = y = z =5** #an integer value is assigned to variables x, y, z simultaneously.

Check your progress -3 :

Q8. Which of the following is a valid variable name in Python?

a) do it
b) do+1
c) 1do
d) All of the above
e) None of the above

Q9. What is the value of 'a' here?

a = 25
b = 35
a = b

a) 25
b) 35

c) It will print error
d) None of the above

Q10. Which of the following is true for variable names in Python?
a) unlimited length
b) variable names are not case sensitive
c) underscore and ampersand are the only two special characters allowed
d) none of the mentioned

Q11. What error occurs when you excecute the following Python Code snippet?

```
apple = mango
```

a) SyntaxError
b) NameError
c) ValueError
d) TypeError

10.6 OPERATORS

Operators are special symbols that represent computations like addition and multiplication. The values the operator is applied to are called operands. Python language supports a wide range of operators. They are:

1. Arithmetic Operators
2. Assignment Operators
3. Relational Operators
4. Logical Operators
5. Bitwise Operators

TABLE 2 : List of Arithmetic Operators

Operator	**Operator Name**	**Description**	**Example (try in lab) n1 = 5, n2 = 6**
+	Addition operator	Adds two operands, producing their sum	In[]: **n1 + n2** Out[]: **11**
-	Subtraction operator	Subtracts the two operands, producing their difference	In[]: **n1 - n2** Out[]: **-1**
*	Multiplication operator	Produces the product of the operands	In[]: **n1 * n2** Out[]: **30**
/	Division operator	Produces the quotient of its operands where the left operand is the dividend and the right operand is the divisor	In[]: **n1 / n2** Out[]: **0.833333**

%	Modulus operator	Divides left hand operand by the right hand operand and returns a remainder	In[]: **n1 % n2** Out[]: **5**
**	Exponent operator	Performs exponential (power) calculation on operators	In[]: **n1 ** n2** Out[]: **15625**
//	Floor division operator	Returns the integral part of the quotient	In[]: **n1 // n2** Out[]: **0**

TABLE 3 : List of Assignment Operators

Operator	**Operator Name**	**Description**	**Example (try in lab) ('~' stand for is equivalent to)**
=	Assignment	Assigns values from right side operands to left side operand	m = n1 + n2 In[]: **m = n1 + n2** In[]: **m** Out[]: **11**
+=	Addition Assignment	Adds the value of right operand to the left operand and assigns the result to left operand	m += n1 ~ m = m + n1 In[]: **m += n1** In[]: **m** Out[]: **16**
-=	Subtraction Assignment	Subtracts the value of right operand from the left operand and assigns the result to left operand	m -= n1 ~ m = m – n1 In[]: **m -= n1** In[]: **m** Out[]: **11**
*=	Multiplication Assignment	Multiplies the value of right operand with the left operand and assigns the result to left operand	m *= n1 ~ m = m* n In[]: **m *= n1** In[]: **m** Out[]: **55**
/=	Division Assignment	Divides the value of right operand with the left operand and assigns the result to left operand	m /= n1 ~ m = m / n1 In[]: **m /= n1** In[]: **m** Out[]: **11.0**
**=	Exponentiation Assignment	Evaluates to the result of raising the first operand to the power of the second operand	m ** n1 ~ m = m ** n1 In[]: **m **= n1** In[]: **m** Out[]: **161051.0**
//=	Floor Division	Produces the integral part	m //= n1 ~ m =

	Assignment	of the quotient of its operands where the left operand is the dividend and the right operand is the divisor	m // n1 In[]: **m //= n1** In[]: **m** Out[]: **32210.0**
%=	Remainder Assignment	Computes the remainder after division and assigns the value to the left operand.	m %= n1 ~ m = m % n1 In[]: **m %= n1** In[]: **m** Out[]: **0.0**

TABLE 4 : List of Relational Operators

Operator	**Operation**	**Description**	**Example (Try in Lab)** **n1 = 10, n2 = 0, s1 = "Hello", s2="World"**
==	Equals to	If values of two operands are equal, then the condition is True, otherwise it is False.	In[]: **n1 ==n2** Out[]: **False** In[] : **s1 ==s2** Out[]: **False**
!=	Not equal to	If values of two operands are not equal, then condition is True, otherwise it is False.	In[] : **n1 != n2** Out[]: **True** In[]: **s1 != s2** Out[]: **False**
>	Greaer than	If the value of the left operand is greater than the value of the right operand, then the condition is True, otherwise it is False.	In[]: **n1 > n2** Out[]: **True** In[]: **s1 > s2** Out[]: **True**
<	Less than	If the value of the left operand is less than the value of the right operand, the condition is True otherwise it is False.	In[]: **n1 < n2** Out[]: **False**
>=	Greater than or equal to	If the value of the left operand is greater than or equal to the right operand, the condition is True otherwise it is False.	In[]: **n1 >= n2** Out[]: **True**
<=	Less than or equal to	If the value of the left operand is less than or equal to the right operand, the condition is True otherwise it is False.	In []: **n1 <= n2** Out[]: **False**

Note :For strings, the characters in both the strings are compared one by one based on their Unicode value. The character with the lower Unicode value is considered to be smaller.

TABLE 5 : List of Logical Operator

Operator	**Operation**	**Description**	**Example (Try in Lab) n1 = 10, n2 = -20**
And	Logical AND	If both operands are True, then condition becomes True.	In[]: **n1 == 10 and n2 == -20** Out[]: **True**
Or	Logical OR	If any of the two operands are True, then condition becomes True.	In[]: **n1 >= 10 or n2 <-20** Out[]: **True**
Not	Logical NOT	Used to reverse the logical state of its operand.	In[]: **not(n1 ==20)** Out[]: **True**

TABLE 6 : Boolean Logic Truth Table

n1	**n2**	**n1 and n2**	**n1 or n2**	**not n1**
True	True	True	True	False
True	False	False	True	
False	True	False	True	True
False	False	False	False	

TABLE 7 : List of Bitwise Operators

Operator	**Operator Name**	**Description**	**Example (try in Lab) n1 = 60, n2 = 13**
&	Binary AND	Result is one in each bit position for which the corresponding bits of both operands as 1s.	In[]: **n1 & n2** Out[]: **12** (means 0000 1100)
|	Binary OR	Result is one in each bit position for which the corresponding bits of either or both operands are 1s.	In[]: **n1 | n2** Out[]: **61** (means 0011 1101)
^	Binary XOR	Result is one in each bit position for which the corresponding bits of either but not both operands are 1s.	In[]: **n1 ^ n2** Out[]: **49** (means 0011 0001)

~	Binary Ones Complement	Inverts the bits of its operand.	In[]: **~n1** Out[]: **-61** (means 1100 0011 in 2's complement form due to a signed binary number)
<<	Binary Left Shift	The left operands value is moved left by the number of bits specified by the right operand.	In[]: **n1 << 2** Out[]: **240** (means 1111 0000)
>>	Binary Right Shift	The left operand value is moved right by the number of bits specified by the right operand.	In[]: **n1 >> 2** Out[]: **15** (means 0000 1111)

TABLE 8 : Bitwise Truth Table

n1	n2	n1 & n2	n1 | n2	n1 ^ n2	~n1
0	0	0	0	0	1
0	1	0	1	1	
1	0	0	1	1	0
1	1	1	1	0	

TABLE 9 : Operator Precedence in Python

Operators	Meaning
()	Parentheses
**	Exponent
+x, -x, ~x	Unary plus, Unary minus, Bitwise NOT
*, /, //, %	Multiplication, Division, Floor Division, Modulus
+, -	Addition, Subtraction
<<,>>	Bitwise shift operators
&	Bitwise AND
^	Bitwise XOR
|	Bitwise OR
==, !=, >, >=, <, <=	Comparisons
Not	Logical NOT
And	Logical AND
Or	Logical OR

Check your progress – 4 :

Q12. 4 is 100 in binary and 11 is 1011. What is the output of the following bitwise operators?

a = 4

b = 11

print (a | b)
print (a >> 2)

a) 15, 1
b) 14, 1
c) 15, 2
d) 14, 2

Q13. What is the output of print(2 ** 3 ** 2)?

a) 64
b) 128
c) 256
d) 512

Q14. What is the output of the following assignment operator?

y = 10
x = y += 2
print(x)

a) 12
b) 10
c) SyntaxError

Q15. What is the output of following code?

a = 100
b = 200
print (a and b)

a) 100
b) 200
c) True
d) False

Q16. Which of the following operators has the lowest precedence?

a) %
b) +
c) **
d) not
e) and

10.7 DATA TYPES

Data types specify the type of data like numbers and characters to be stored and manipulated within a program. Basic data types of Python are:

10.7.1 Number
10.7.2 String
10.7.3 None

10.7.4 List
10.7.5 Tuple
10.7.6 Dictionary
10.7.7 Set

10.7.1 Number

In Numbers or Numerical Data Types, we mainly have numbers. These numbers are also of four types : Integer, Float, Complex, Boolean.

TABLE 10 : Data Types in Python

Type / Class	Description	Examples
Int	Integer numbers	-12, -3, 0, 123, 2
float	Floating point numbers	-2.04, 4.0, 14.23
complex	Complex numbers	3+4j, 2-2j
bool	Boolean numbers	True, false

In Interactive mode:

Note :type() function can be used to check the data type of the variables

Examples:

In[]: **i = 23** #variable i assigned with integer value

In[]: **type(i)**

Out[]: **int**

In[]: **c = 2+5j** #variable c assigned with a complex number

In[]: **type(c)**

Out[]: **complex**

In[]: **b = 10>7**

In[]: **type(b)** #to display the data type of variable b

Out[]: **bool**

In[]: **b** #to display the value of b

Out[]: **True**

10.7.2 String

A **string** is an ordered sequence of characters. These characters may be alphabets, digits or special characters including spaces. String values are enclosed in single quotation marks (e.g. “hello’) or in double quotation marks (e.g., “python course”). The quotes are not a part of the string, they are used to mark the beginning and end of the string for the interpreter. In Python, there is no character data type, a character is a string of length one. It is represented by **str** class. Few of the characteristics of strings are:

- Numerical operations can not be performed on strings, even when the string contains a numeric value like str2 defined below.
- A string has a length. Get the length with the len() built-in function.
- A string is indexable. Get a single character at a position in a string with the square bracket operator, for example str1[4]. Indexing always start with 0.
- One can retrieve a slice (sub-string) of a string with a slice operation, for example str1[4:8].
- Strings are immutable, which means you can’t change an existing string.

In[]: **str1 = ‘Hello Friend’** #variable str1 of string type is declared

In[]: **str2 = “452”** #variable str2 of string type is declared

In[]: **len(str1)** #length of string str1

Out[]: **12**

In[]: **str1[4]** #to access 5th character in the string

Out[]: **‘o’**

In[]: **str1[-1]** #negative indices can be used, which count backward from the end of string. [-1] yields the last letter.

Out[]: **‘d’**

In []: **str[4:8]** # Operator returns the part of the string from the first index to the second index, including the first but excluding the last. So, sub-string starting from 5 character to 8 character is extracted.

Out[]: **‘o Fr’**

If one omits the first index (before the colon), the slice starts at the beginning of the string and if second index is omitted, the slice goes to the end of the string.

In[]: **str1[:3]**

Out[]: **'hel'**

In[]: **str1[3:]**

Out[]: **'lo Friend'**

In []: **str1[2] = 'a'**

TypeError: 'str' object does not support item assignment.

In[]: **str2 = ' Narender'**

In[]: **str1 + str2** #for concatenating two strings, "+" operator is used

Out[]: **'hello Friend Narender'**

In[]: **str1 * 3** #for creating multiple concatenated copies of a string, "*" operator is used

Out[]: **'hello FriendhelloFriendhello Friend'**

10.7.3 None

A **none** is another special data type in Python. A none is frequently used to represent the absence of a value. For example,

In[]: **money = None** #None value is assigned to variable money

Some literatures on Python consider additional data types like List, Tuple, Set &Dictionary whereas some others consider them as the built-in data structures. We will put them in the category of data structures and discuss considering them in same but considering them as data types is also not wrong.

10.7.4 List

A **list** is a sequence of items separated by commas and items enclosed in square brackets [ ]. These are similar to arrays available with other programmimg languages but unlike arrays items in the list may be of different data types. Lists are mutable, and hence, they can be altered even after their creation. Lists in Python are ordered and have a definite sequence and the indexing of a list is done with 0 being the first index. Each element in the list has its definite place in the list, which allows duplicating of elements in the list, with each element having its own distinct place and creditability. It is represented by **list** class.

In[]: **list1 = [5, 3.4, 'IGNOU', 'Part 3', 45]** #to create a list

In[]: **list1** #to print the elements of the list list1

Out[]: **[5, 3.4, 'IGNOU', 'Part 3', 45]**

In[]: **list1[0]** #accessing first element of the list

Out[]: **5**

In[]: **list1[3]** #accessing 4th element of the list

Out[]: **Part 3**

In[]:**list1[-1]** #accessing last element of the list

Out[]: **45**

In[]: **list1[-3]** #accessing last 3rdelement of the list

Out[]: **'IGNOU'**

10.7.5 Tuple

A **tuple** is similar to the list in many ways. Like lists, tuples also contain the collection of the items of different data types. The items of the tuple are separated with a comma and enclosed in parentheses.

A tuple is a read-only data structure as we can't modify the size and value of the items of a tuple and it is immutable.

In[]: **t = ('hello','world',2)** #creating tuple t

In[]: **t** # printing tuple t

Out[]: **('hello', 'world', 2)**

In[]: **t[1]** #accessing specific element of the tuple say second element

Out[]: **'world'**

10.7.6 Dictionary

A **dictionary** in Python holds data items in key-value pairs of items. The items in the dictionary are separated with the comma and enclosed in the curly brackets{}. Dictionaries permit faster access to data. Every key is separated from its value using a colon(:) sign. The key value pairs of a dictionary can be accessed using the key. Keys are usually of string type and their values can be of any data type. In order to access any value in the dictionary, we have to specify its key in square brackets[].

#Creating dictionary dict1

In[]: **dict1 = {'Fruit':'Apple','Climate':'Cold','Price(Kg)':120}**

In[]: **dict1** #printing the contents of dictionary dict1

Out[]: **{'Fruit': 'Apple', 'Climate': 'Cold', 'Price(Kg)': 120}**

In[]: **dict1['Climate']** #Getting value by specifying the key

Out[]:**Cold**

In[]: **dict1.keys()** #printing all the Keys in the dictionary

Out[]: **dict_keys(['Fruit', 'Climate', 'Price(Kg)'])**

In[]: **dict1.values()** #printing all the values in the dictionary

Out[]: **dict_values(['Apple', 'Cold', 120])**

10.7.7 Set

In Python, a**set** is an orderd collection of data type that is iterable, mutable and has no duplicate elements. The order of elements in a set is undefined though it may consist of various elements. The major advantage of using a set, as opposed to a list, is that it has a highly optimized method for checking whether a specific element is contained in the set.

Sets can be created by using the built-in set() function with an iterable object or a sequence by placing the sequence inside curly braces, separated by 'comma'. A set contains only unique elements but at the time of set creation, multiple duplicate values can also be passed. The order of elements in a set is undefined and is unchangeable. Type of elements in a set need not be the same, various mixed-up data type values can also be passed to the set.

In[]: **set1 = set("IGNOU MCA BATCH")** #creating set set1

In[]: **set1**
#displaying contents of set1

Out[]: **{' ', 'A', 'B', 'C', 'G', 'H', 'I', 'M', 'N', 'O', 'T', 'U'}**

In[]: **set2 = set([2, 3, 2, 'MCA', 'IGNOU', 1, 'IGNOU'])** #set with the use of mixed values

In[]: **set2**

Out[]: **{1, 2, 3, 'IGNOU', 'MCA'}**

Check your progress - 5:

Q17. Which of these is not a core data type?

a) Lists
b) Dictionary
c) Tuples
d) Class

Q18. What data type is the object below?

L = [1, 23, 'hello', 1]

a) List
b) Dictionary
c) Tuple
d) Array

Q19. What will be the output of the following Python code?

In[]:str="hello"
In[]: str[:2]

a) he
b) lo
c) olleh
d) hello

Q20. In python we do not specify types, it is directly interpreted by the compiler.So consider the following operation to be performed:

x = 13 ? 2

objective is to make sure x has a integer value, select all that apply

a) x = 13 // 2
b) x = int(13 / 2)
c) x = 13 % 2
d) all of the above

Q21. What is the result of print(type({}) is set)

a) True
b) False

Q22. What is the data type of print (type(10))?

a) float
b) integer
c) int

Q23. If we convert one data type to another, then it is called

a) Type Conversion
b) Type Casting
c) Both of the above

d) None of the above

Q24. In which data type, indexing is not valid?
a) list
b) string
c) distionary
d) None of the above

Q25. In which of the following data type duplicate items are not allowed?
a) list
b) set
c) dictionary
d) None of the above

Q26. Which statement is correct?
a) List is immutable && Tuple is mutable
b) List is mutable && Tuple is immutable
c) Both are mutable
d) Both are immutable

10.8 DATA STRUCTURES

Data Structures are used to organize, store and manage data for efficient access and modification. Some literatures on Python categorize List, Tuple, Dictionary and Set as data types and some categorize these as primitive data structures. We had already introduced the above mentioned structures in the data type. Lets discuss them in detail here:

10.8.1 List

A **list** a container data type that acts as a dynamic aray. That is to say, a list is a sequence that can be indexed into and that can grow and shrink. A few characteristics of lists are:

- A list has a (current) length – Get the length of a lsit with len() function.
- A list has an order – the items in a list are ordered, order going from left to right.
- A list is heterogeneous –different types of objects can be inserted into the same list.
- Lists are mutable and one can extend, delete, insert or modify items in the list.

The syntax for creating list is,

list_name = [item_1, item_2, item_3, ……., item_n]

Any item can be stored in a list like string, number, object, another variable and even another list. In a list, we can have a mix of different item types and these item types need not have to be homogeneous. For example, we can have a list which is a mix of type numbers, strings and another list itself. The contents of the list variable are displayed by executing the list variable name.

Examples :

In[]: **list_name = [ ]** #empty list without any items

#when each item in the list is string

In[]: **stringlist = [“mahesh”, “ramesh”, “suresh”, “vishnu”, “ganesh”]**

#when each item in the list is number

In[]: **numlist = [4, 6, 7, 10, 15, 13]**

#when list contains mixed items

In[]: **mixed_list = [‘cat’, 87.23, 65, [9, 8, 1]]**

10.8.1.1 Indexing and Slicing in Lists

As an ordered sequence of elements, each item in a list can be individually, through indexing. The expression inside the bracket is called the index. Lists use square brackets [ ] to access individual items, with the first item at index 0, the second item at index 1 and so on. The index provided within the square brackets indicates the value being accessed.

The syntax for accessing an item in a list is,

list_name [index]

where index should always be an integer value and indicates the item to be selected.

In[]: **stringlist[0]** #to access first item in the stringlist

Out[]: **‘mahesh’**

In[]: **numlist[4]** #to access fifth item in the numlist

Out[]: **15**

In[]: **numlist[6]** #if index value is more than the number of items in the list, it will raise an error

Output would be :

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

numlist[6]

IndexError : List index out of range

Here, in numlist index numbers range from 0 to 5 as it contains 6 items.

In addition to positive index numbers, one can also access items from the list with a negative index number, by counting backwards from the end of the list, starting at -1. It is useful of the list is long and we want to locate an item towards the end of a list.

In[]: **stringlist[-2]** #to access second list item from the list

Out[]: **'vishnu'**

The slice operator also works on lists:

In[]: **numlist[1:3]**

Out[]: **[6, 7]**

In[]: **mixed_list[:3]**

Out[]: **['cat', 87.23, 65]**

In[]: **mixed_list[2:]**

Out[]: **[65, [9, 8, 1]]**

In[]: **numlist[:]**

Out[]: **[4, 6, 7, 10, 15, 13]**

If the first index is omitted, the slice starts at the beginning and if the second index is omitted, the slice goes to the end. So, if both indexes are omitted, the slice is a copy of the whole list.

10.8.1.2 List Operations

The + operator concatenates lists:

In[]: **a = [1, 2, 3]**

In[]: **b = [4, 5, 6]**

In[]: **a + b**

Out[]: **[1, 2, 3, 4, 5, 6]**

Similarly, the * operator repeats a list given a number of times:

In[]: **c = [‘pass’]**

In[]: **c * 3**

Out[]: **[‘pass’, ‘pass’, ‘pass’]**

== operator is used to compare the two lists.

In[]: **a == b**

Out[]: **False**

in and **not in** membership operator are used to check for the presence of an item in the list.

In[]: **2 in a**

Out[]: **True**

In[]: **4 not in [a]**

Out[]: **True**

10.8.1.3 The list() function

The built-in-list() function is used to create a list. The syntax for list() function is,

list([sequence])

where the sequence can be a string, tuple or list itself. If the optional sequence is not specified then an empty list is created.

In[]: **greet = “How are you doing?”** #string declaration

In[]: **greet**

Out[]: **‘How are you doing?’**

In[]: **str_to_list = list(greet)** #string converted to list using list()

In[]: **str_to_list**

Out[]: **[‘H’,’o’,’w’,’ ‘,’a’,’r’,’e’,’ ‘,’y’,’o’,’u’,’ ‘,’d’,’o’,’i’,’n’,’g’,’?’]**

In[]: **friend = “nidhi”** #string declaration

In[]: **str_to_list + friend** #list concatenated with string

TypeError: can only concatenate list (not “str”) to list

In[]: **str_to_list + list(friend)** #list concatenated with list

Out[]:**['H','o','w',' ','a','r','e',' ','y','o','u',' ','d','o','i','n','g','?','n','i','d','h','i']**

10.8.1.4 Modifying items in Lists

Lists are mutable in nature as the list items can be modified after a list has been created. Also, a list can be modified by replacing the older item with a newer item in its place and without assigning the list to a completely new variable.

In[]: **stringlist[1] = "umesh"**

In[]: **stringlist**

Out[]: **['mahesh', 'umesh', 'suresh', 'vishnu', 'ganesh']**

When an existing list variable is assigned to a new variable, an assignment (=) on lists does not make a new copy. Instead, assignment makes both the variables names point to the same list in memory. That's why any change in one list will reflect in other list also.

In[]: **listofstrings = stringlist**

In[]: **listofstrings[1] = "ramesh"**

In[]: **listofstrings**

Out[]:**['mahesh', 'ramesh', 'suresh', 'vishnu', 'ganesh']**

In[]: **stringlist**

Out[]:**['mahesh', 'ramesh', 'suresh', 'vishnu', 'ganesh']**

10.8.1.5 Traversing a list

The most common way to traverse the elements of a list is with a for loop.

In[]: **for names in stringlist :**

print(names)

Out[]: **mahesh**

ramesh

suresh

vishnu

ganesh

This works well if one needs to read the elements of the list. But if one wants to write or update the elements, one needs the indices. For this one needs to combine the functions range and len:

In[]: **for i in range(len(numlist)) :**

numlist[i] = numlist[i] * 2

In[]: **numlist**

Out[]: **[8, 12, 14, 20, 30, 26]**

This loop traverses the list and updates each element. The command len returns the number of elements in the list.The command range returns a list of indices from 0 to n-1, where n is the length of the list. Each time through the loop, the variable i gets the index of the next element. The assignment statement in the body uses i to read the old value of the element and to assign the new value.

10.8.1.6 List Methods

Python provides methods that operate on lists. For example, **append** adds a new element to the end of a list:

In[]: **numlist.append(50)**

In[]: **numlist**

Out[]: **[8, 12, 14, 20, 30, 26, 50]**

extend takes a list as an argument and appends all of the elements:

In[]: **nlist = [25, 35, 45]**

In[]: **numlist.extend(nlist)**

In[]:**numlist**

Out[]: **[8, 12, 14, 20, 30, 26, 50, 25, 35, 45]**

sort arranges the elements of the list from low to high:

In[]: **numlist.sort()**

In[]: **numlist**

Out[]: **[8, 12, 14, 20, 25, 26, 30, 35, 45, 50]**

pop is used to delete elements from a list if the index of the element is known:

In[]: **numlist.pop(5)**

Out[]: **26**

In[]: **numlist**

Out[]: [8, 12, 14, 20, 25, 30, 35, 45, 50]

pop modifies the list and returns the element that was removed. If the removed element is not required then one can use the **del** operator:

In[]: **del numlist[0]**

In[]: **numlist**

Out[]: **[12, 14, 20, 25, 30, 35, 45, 50]**

To remove more than one element, **del** can be used with a slice index

In[]: **del numlist[0:2]**

In[]: **numlist**

Out[]: **[20, 25, 30, 35, 45, 50]**

If element from the list is known which needs to be removed and not the index, **remove** can be used:

In[]: **numlist.remove(30)**

In[]: **numlist**

Out[]: **[20, 25, 35, 45, 50]**

The return value for remove is none.

To get a list of all the methods associated with the list, pass the list function to **dir()**

In[]: **dir(list)**

TABLE 11 : Various List Methods

List Methods	**Syntax**	**Description**
append()	list.append(item)	The append() method adds a single item to the end of the list. This method does not return new list and it just modifies the original.
count()	list.count(item)	The count() method counts the number of times the item has occurred in the list and returns it.
insert()	list.insert(index, item)	The insert() method inserts the item at the given index, shifting items to the right.
extend()	list.extend(list2)	The extend() method adds the items in list2 to the end of the list.

index()	list.index(item)	The index() method searches for the given item from the start of the list and returns its index. If the value appears more than once, you will get the index of the first one. If the item is not present in the list then ValueError is thrown by this method.
remove()	list.remove(item)	The remove() method searches for the first instance of the given item in the list and removes it. If the item is not present in the list then ValueError is thrown by this method.
sort()	list.sort()	The sort() method sorts the items in place in the list. This method modifies the original list and it does not return a new list.
reverse()	list.reverse()	The reverse() method reverses the items in place in the list. This method modifies the original list and it does not return a new list.
pop()	list.pop([index])	The pop() method removes and returns the item at the given index. This method returns the rightmost item if the index is omitted.

Note: Replace the word"list" mentioned in the syntax with your actual list name in your code.

10.8.1.7Built-In Functions Used on Lists

There are many built-in functions for which a list can be passed as an argument.

TABLE 12 : Built – In Functions Used on Lists

Built-In Functions	**Description**
len()	The len() function returns the numbers of items in a list.
sum()	The sum() function returns the sum of numbers in the list.
any()	The any() function returns True if any of the Boolean values in the list is True.
all()	The all() function returns True if all the Boolean values

	in the list are True, else returns False.
sorted()	The sorted() function returns a modified copy of the list while leaving the original list untouched.

In[]: **len(stringlist)**

Out[]: **5**

In[]: **sum(numlist)**

Out[]: **175**

In[]: **max(numlist)**

Out[]: **50**

In[]: **min(numlist)**

Out[]: **20**

In[]: **any([1, 1, 0, 0, 1, 0])**

Out[]: **True**

In[]: **all([1, 1, 1, 1])**

Out[]: **True**

In[]: **sorted_stringlist = sorted(stringlist)**

In[]: **sorted_stringlist**

Out[]: **[‘ganesh’, ‘mahesh’, ‘ramesh’, ‘suresh’, ‘vishnu’]**

10.8.1.8 Nested List

A list inside another list is called a **nested list** and you can get the behavior of nested lists in Python by storing lists within the elements of another list. The syntax for nested lists is:

nested_list_name = [[item_1, item_2, item_3],

[item_4, item_5, item_6],

[item_7, item_8, item_9]]

Each list inside another list is separated by a comma. One can traverse through the items of nested lists using the for loop.

In[]: asia = **[[“India”, “Japan”, “Korea”],**

[“Srilanka”, “Myanmar”, “Thailand”],

[“Cambodia”, “Vietnam”, “Israel”]]

In[]: **asia[0]**

Out[]: **[‘India’, ‘Japan’, ‘Korea’]**

In[]: **asia[0][1]**

Out[]: **‘Japan’**

In[]: **asia[1][2]**

Out[]: **‘Thailand’**

Program 10.1 : Write a Program to Find the Transpose of a Matrix

```
matrix = [[10, 20],
          [30, 40],
          [50, 60]]
matrix_transpose = [[0, 0, 0],
                    [0, 0, 0]]
for rows in range(len(matrix)):
    for columns in range(len(matrix[0])):
matrix_transpose[columns][rows] = matrix[rows][columns]
print("Transposed Matrix is")
for items in matrix_transpose:
    print(items)
```

Fig 1 : Screen Shot of execution of Program 10.1

```
[2] matrix = [[10, 20],
              [30, 40],
              [50, 60]]
    matrix_transpose = [[0, 0, 0],
                        [0, 0, 0]]
    for rows in range(len(matrix)):
      for columns in range(len(matrix[0])):
        matrix_transpose[columns][rows] = matrix[rows][columns]
    print("Transposed Matrix is")
    for items in matrix_transpose:
      print(items)

    Transposed Matrix is
    [10, 30, 50]
    [20, 40, 60]
```

Check your progress – 6 :

Q27. Empty list in python is made by?

a) l = []

b) l = list()

c) Both of the above
d) None of the above

Q28. If we try to access the item outside the list index, then what type of error it may give?

a) List is not defined
b) List index out of range
c) List index out of bound
d) No error

Q29. l = [1,2,3,4], then l[-2] is?

a) 1
b) 2
c) 3
d) 4

Q30. The marks of a student on 6 subjects are stored in a list, list1 = [80, 66, 94, 87, 99, 95]. How can the students average marks be calculated?

a) print(avg(list1))
b) print(sum(list1)/len(list1))
c) print(sum(list1)/sizeof(list1))
d) print(total(list1)/len(list1))

Q31. What will be the output of the following Python code?

```
list1=[“Python”, “Java”, “c”, “C”, “C++”]
print(min(list1))
```

a) c
b) C++
c) C
d) Min function can’t be used on string elements

Q32. The elements of a list are arranged in descending order:
Which of the following two will give same outputs?

a) print(list_name.sort())
b) print(max(list_name))
c) print(list_name.reverse())
d) print(list_name[-1])

i, ii
i, iii
ii, iii
iii, iv

Q33. What will be the output of below python code?

```
list1=[“tom”, “mary”, “simon”]
list1.insert(5,8)
print(list1)
```

a) [“tom”, “mary”, “simon”, 5]
b) [“tom”, “mary”, “simon”, 8]
c) [8, “tom”, “marry”, “simon”]
d) Error

Q34. What will be the output of below python code?
list1=[1,3,5,2,4,6,2]
list1.remove(2)
print(sum(list1))
a) 18
b) 19
c) 21
d) 22

Q35. What will be the output of below python code?
list1=[3,2,5,7,3,6]
list1.pop(3)
print(list1)
a) [3,2,5,3,6]
b) [2,5,7,3,6]
c) [2,5,7,6]
d) [3,2,5,7,3,6]

10.8.2 Tuple

In mathematics, a tuple is a finite ordered list (sequence) of elements. A **tuple** is defined as a data structure that comprises an ordered, finite sequence of immutable, heterogeneous elements that are of fixed sizes. Often, we may want to return more than one value from a function. Tuples solve this problem. They can also be used to pass multiple values through one function parameter.

10.8.2.1 Creating Tuples

A tuple is a finite ordered list of values of possibly different types which is used to bundle related values together without having to create a specific type to hold them. Tuples are immutable. Once a tuple is created, one cannot change its values. A tuple is defined by putting a comma-separated list of values inside parantheses(). Each value inside a tuple is called an item. The syntax for creating tuples is,

tuple_name = (item_1, item_2, item_3, …….., item_n)

Syntactically, a tuple is a comma-separated list of values:

In[]: **t = ‘a’, ‘b’, ‘c’, ‘d’, ‘e’**

Although it is not necessary, it is common to enclose tuples in parantheses to help us quickly identify tuples when Python code is looked at:

In[]: **t = (‘a’, ‘b’, ‘c’, ‘d’, ‘e’)**

It is actually the comma that forms a tuple making the commas significant and not the parentheses.

To create a tuple with a single element, one must include the final comma:

In[]: **t1 = (‘a’,)**

In[]: **type(t1)**

Out[]: **tuple**

In Python, the tuple type is tuple. Without the comma Python treats (‘a’) as an expression with a string in parentheses that evaluates to a string:

In[]: **t2 = (‘a’)**

In[]: **type(t2)**

Out[]: **str**

One can create an empty tuple without any values. The syntax is,

tuple_name = ()

In[]: **empty_tuple = ()**

In[]: **empty_tuple**

Out[]: **()**

One can store any type of type string, number, object, another variable, and even another tuple itself. One can have a mix of different types of items in tuples, and they need not be homogeneous.

In[]: **socis = (‘mca’,’6 sem’,’3-6 yrs’,54000)**

In[]: **ignou = (‘soe’,’soh’,socis,’soms’)**

In[]: **ignou**

Out[]: **(‘soe’, ‘soh’, (‘mca’, ‘6 sem’, ‘3 – 6 yrs’, 54000), ‘soms’)**

10.8.2.2 Basic Tuple Operations

Most list operators also work on tuples. The bracket operator indexes an element:

In[]: **t[0]**

Out[]: **‘a’**

And the slice operator selects a range of elements.

In[]: **t[1:3]**

Out[]: **('b', 'c')**

But if you try to modify one of the elemnts of the tuples, one will get an error:

In[]: **t[0] = 'A'**

TypeError: 'tuple' object does not support item assignment

We can't modify the elements of a tuple, but can replace one tuple with another:

In[]: **t = ('A',) + t[1:]**

In[]: **t**

Out[]: **('A', 'b', 'c', 'd', 'e')**

Here, + operator is used to concatenate two tuples together. Similarly, * operator can be used to repeat a sequence of tuple items.

In[]: **t * 2**

Out[]: **('A', 'b', 'c', 'd', 'e', 'A', 'b', 'c', 'd', 'e')**

in and **not in** membership operator are used to check for the presence of an item in a tuple.

Comparison operators like <, <=, >, >=, == and != are used to compare tuples.

Python starts by comparing the first element from each sequence. If they are equal, it goes on to the next element, and so on, until it finds elements that differ. Subsequent elements are not considered even if they are really big.

In[]: **(0, 1, 2) < (0, 3, 4)**

Out[]: **True**

In[]: **(0, 1, 500000) < (0, 3, 4)**

Out[]: **True**

10.8.2.3 The tuple() Function

The built-in tuple() function is used to create a tuple. The syntax for the tuple() function is:

tuple([sequence])

where sequence can be a number, string or tuple itself. If the optional sequence is not specified, then an empty tuple is created.

In[]: **t3 = tuple()**

In[]: **t3**

Out[]: **()**

If the argument is a sequence (string, list or tuple), the result of the call to tuple is a tuple with the elements of the sequence:

In[]: **t3 = tuple('IGNOU')**

In[]: **t3**

Out[]: **('I', 'G', 'N', 'O', 'U')**

In[]: **t4 = (1, 2, 3, 4)**

In[]: **nested_t = (t3, t4)**

In[]: **nested_t**

Out[]: **(('I', 'G', 'N', 'O', 'U'), (1, 2, 3, 4))**

10.8.2.4Built-In Functions Used on Tuples

There are many built-in functions as listed in TABLE for which a tuple can be passed as an argument.

TABLE 13: Built-In functions Used on Tuples

Built-In Functions	Description
len()	The len() function returns the number of items in a tuple.
sum()	The sum() function returns the sum of numbers in the tuple.
sorted()	The sorted() function returns a sorted copy of the tuple as a list while leaving the original tuple untouched.

In[]: **len(t3)**

Out[]: **5**

In[]: **sum(t4)**

Out[]: **10**

In[]: **t5 = sorted(t3)**

In[]: **t5**

Out[]: **[‘G’, ‘I’, ‘N’, ‘O’, ‘U’]**

10.8.2.5 Tuple assignment

One of the unique syntactic features of the Python language is the ability to have a tuple on the left side of an assignment statement. This allows one to assign more than one variable at a time when the left side is a sequence.

In this example we have a two-element list (which is a sequence) and assign the first and second elements of the sequence to the variables x and y in a single statement.

In[]: **m = [‘good’, ‘luck’]**

In[]: **x, y = m**

In[]: **x**

Out[]: **‘good’**

In[]: **y**

Out[]: **‘luck’**

Python roughly translates the tuple assignment syntax to be the following:

In[]: **m = [‘good’, ‘luck’]**

In[]: **x = m[0]**

In[]: **y = m[1]**

Stylistically, when we use a tuple on the left side of the assignment statement, we omit the parentheses, but the following is an equally valid syntax:

In[]: **(x, y) = m**

10.8.2.6 Relation between Tuples and Lists

Though tuples may seem similar to lists, they are often used in different situations and for different purposes. Tuples are immutable, and usuallycontain a heterogeneous sequence of elements that are accessed via unpacking or indexing. Lists are mutable, and their items are accessed via indexing. Items cannot be added, removed or replaced in a tuple.

If an item within a tuple is mutable, then you can change it. Consider the presence of a list as an item in a tuple, then any changes to the list get reflected on the overall items in the tuple.

In[]: **ignou = [‘soe’, ‘soms’, ‘socis’]**

In[]: **univ = (‘du’, ‘ggsipu’, ‘jnu’, ignou)**

In[]: **univ**

Out[]: **('du', 'ggsipu', 'jnu', ['soe', 'soms', 'socis'])**

In[]: **univ[3].append('soss')**

In[]: **univ**

Out[]: **('du', 'ggsipu', 'jnu', ['soe', 'soms', 'socis', 'soss'])**

10.8.2.7 Tuple Methods

To get a list of all the methods associated with the tuple, pass the tuple function to dir().

Various methods associated with tuple are listed in the TABLE.

TABLE 14: Various Tuple Methods

Tuple Methods	Syntax	Description
count()	tuple_name.count(item)	The count() method counts the number of times the item has occurred in the tuple and returns it.
index()	tuple-name.index(item)	The index() method searches for the given item from the start of the tuple and returns its index. If the value appears more than once, you will get the index of the first one. If the item is not present in the tuple, then ValueError is thrown by this method.

Note: Replace the word "tuple_name" mentioned in the syntax with your actual tuple name in the code.

In[]: **channels = ("dd", "star", "sony", "zee", "dd", "sab")**

In[]: **channels.count("dd")**

Out[]: **2**

In[]: **channels.index("dd")**

Out[]: **0**

In[]: **channels.index("zee")**

Out[]: **3**

In[]: **channels.count("sab")**

Out[]: **1**

PROGRM 10.2 : Program to Populate with User-Entered Items

```
1. tuple_items = ()
2. total_items = int(input("Enter the total number of items: "))
3. for i in range(total_items):
4. user_input = int(input("Enter a number: "))
5. tuple_items += (user_input,)
6. print(f"Items added to tuple are {tuple_items}")
7. list_items = []
8. total_items = int(input("Enter the total number of items: "))
9. for i in range(total_items):
10.  item = input("Enter an item to add: ")
11. list_items.append(item)
12. items_of_tuple = tuple(list_items)
13. print(f"Tuple items are {items_of_tuple}")
```

Fig 2 : Screen Shot of execution of Program 10.2

```
tuple_items = ()
total_items = int(input("Enter the total number of items: "))
for i in range(total_items):
     user_input = int(input("Enter a number: "))
     tuple_items += (user_input,)
print(f"Items added to tuple are {tuple_items}")
list_items = []
total_items = int(input("Enter the total number of items: "))
for i in range(total_items):
     item = input("Enter an item to add: ")
     list_items.append(item)
items_of_tuple = tuple(list_items)
print(f"Tuple items are {items_of_tuple}")
```

```
Enter the total number of items: 5
Enter a number: 8
Enter a number: 2
Enter a number: 9
Enter a number: 1
Enter a number: 6
Items added to tuple are (8, 2, 9, 1, 6)
Enter the total number of items: 4
Enter an item to add: 2
Enter an item to add: 4
Enter an item to add: 6
Enter an item to add: 8
Tuple items are ('2', '4', '6', '8')
```

Items are inserted into the tuple using two methods: using continuous concatenation += operator and by converting list items to tuple items. In the code, tuple_itemsis of tuple type. In both the methods, the total number of items are specified which will be inserted to the tuple beforehand. Based on this number, the for loop is iterated using the range() function. In the first method, the user entered items are continuously concatenated to the tuple using += operator. Tuples are immutable and are not supposed to be changed. During each iteration, each original_tuple is replaced by original_tuple + (new_element), thus creating a new tuple. Notice a comma after new_element. In the second method, a list is created. For each iteration, the user entered value is appended to the list_variable. This list is then converted to tuple type using tuple() function.

Program 10.3 : Program to Swap Two Numbers without Using Intermediate/Temporary Variable. Prompt the User for Input.

1. a = int(input("Enter a value for the first number "))
2. b = int(input("Enter a value for the second number "))
3. b, a = a, b
4. print("After Swapping")
5. print(f"Value for first number {a}")
6. print(f"Value for second number {b}")

Fig. 3 : Screen Shot of Program 10.3

```
a = int(input("Enter a value for the first number "))
b = int(input("Enter a value for the second number "))
b, a = a, b
print("After Swapping")
print(f"Value for first number {a}")
print(f"Value for second number {b}")
```

```
Enter a value for the first number 5
Enter a value for the second number 9
After Swapping
Value for first number 9
Value for second number 5
```

The contents of variables a and b are reversed. The tuple variables are on the left side of the assignment operator and, on the right side, are the tuple values. The number of variables on the left and the number of values on the right has to be the same. Each value is assigned to its respective variable.

Check your progress – 7 :

Q36. A python tuple can also be created without using parentheses

a) False
b) True

Q37. What is the output of the following?
aTuple = "Yellow", 20, "Red"
a, b, c = aTuple
print(a)

a) ('Yellow', 20, 'Red')
b) TypeError
c) Yellow

Q38. Choose the correct way to access value 20 from the following tuple
aTuple = ("Orange", [10, 20, 30], (5, 15, 25))

a) aTuple[1:2][1]
b) aTuple[1:2](1)
c) aTuple[1:2][1]
d) aTuple[1][1]

Q39. Select which is true for python tuple

a) A tuple maintains the order of items
b) A tuple is unordered
c) we cannot change the tuple once created
d) we can change the tuple once created

Q40. What will be the output of below python code:

```
tuple1=(2, 4, 3)
tuple2=tuple1*2
print(tuple2)
```

a) (4, 8, 6)
b) (2, 4, 3, 2, 4, 3)
c) (2, 2, 4, 4, 3, 3)
d) Error

Q41. What will be the output of below python code:

```
tupl = (“annie”, “hena”, “sid”)
print(tupl[-3:0])
```

a) (“annie”)
b) ()
c) None
d) Error as slicing is not possible in tuple

Q42. Which of the following options will not result in an error when performed on tuples in python where tupl = (5, 2, 7, 0, 3)?

a) tupl[1] = 2
b) tupl.append(2)
c) tupl1 = tupl + tupl
d) tupl.sort()

10.8.3 Dictionaries

In the real world, you have seen your Contact-list in your phone. It is practically impossible to memorize the mobile number of everyone you come across. In the Contact-list, you store the name of the person as well as his number. This allows you to identify the mobile number based on a person’s name. One can think of a person’s name as the key that retrives his mobile number, which is the value associated with the key. So, dictionary can be thought of :

- an un-ordered collection of key-value pairs, with the requirement that the keys are unique within a dictionary.
- as a mapping between a set of indices (which are called keys) and a set of values. Each key maps to a value. The association of a key and a value is called a key-value pair or sometimes an item.
- has a length, specifically the number of key-value pairs.
- provides fast look up by key.

The keys of the dictionary must be immutable object types and are case sensitive. Keys can be either a string or a number. But lists can not be used as keys. A value in the dictionary can be of any data type including string, number, list or dictionary itself.

Dictionaries are constructed using curly braces {}, wherein a list of key:value pairs get separated by commas. There is a colon(:) separating each of these keys and value pairs, where the words to the left of the colon operator are the keys and the words to the right of the colon operator are the values.

The syntax for creating a dictionary is :

dictionary_name = {key_1:value_1, key_2:value_2, key_3:value_3, …….. , key_n : value_n}

In[]: **eng2sp = {'one' : 'uno', 'two' : 'dos', 'three' : 'tres'}**

In[]: **eng2sp**

Out[]:**{'one' : 'uno', 'two' : 'dos', 'three' : 'tres'}**

With Python 3.6 version, the output of dictionary statements is ordered key:value pairs. Here, ordered means "insertion ordered", i.e, dictionaries remember the order in which the key:value pairs were inserted. The elements of a dictionary are never indexed with integer indices. Instead, the keys are used to look up the corresponding values:

In[]: **eng2sp['two']**

Out[]: **'dos'**

The key 'two' always maps to the value 'dos' so the order of the items doesn't matter. If the key isn't in the dictionary, one get an exception:

In[]: **eng2sp['four']**

Out[]: **KeyError: 'four'**

Slicing in dictionaries is not allowed since they are not ordered like lists.

10.8.3.1 Built –In Functions Used on Dictionaries

There are many built-in functions for which a dictionary can be passed as an argument. The main operations on a dictionary are storing a value with some key and extracting the value for a given key.

TABLE 15: Built-In Functions Used on Dictionaries

Built – in Functions	**Description**
len()	The len() function returns the number of items (key:value pairs) in a dictionary.
all()	The all() function returns Boolean True value if all the keys in the dictionary are True else retuens False.
any()	The any() function returns Boolean True value if any

	of the key in the dictionary is True else returns False.
sorted()	The sorted() function by default returns a list of items, which are sorted based on dictionary keys.

In[]: **len(eng2sp)**

Out[]: **3**

len() function can be used to find the number of key:value pairs in the dictionary eng2sp. In Python, any non-zero integer value is True, and zero is interpreted as False. With all() function, if all the keys are Boolean True values, then the output is True else it is False.

In[]: **dict_func = {0 : True, 1 : False, 4 : True}**

In[]: **all(dict_func)**

Out[]: **False**

For **any()** function, if any one of the keys is True then it results in a True Boolean value else False Boolean value.

In[]: **any(dict_func)**

Out[]: **True**

The **sorted()** function returns the sorted list of keys by default in ascending order without modifying the original key:value pairs. For list of keys sorted in descending order by passing the second argument as reverse = True.

In[]: **sorted(eng2sp)**

Out[]: **[‘one’, ‘three’, ‘two’]**

In[]: **sorted(eng2sp, reverse=True)**

Out[]: **[‘two’, ‘three’, ‘one’]**

The **in** operator works on dictionaries; it tells whether something appears as a key in the dictionary (appearing as a value is not good enough).

In[]: **‘one’ in eng2sp**

Out[]: **True**

In[]: **‘uno’ in eng2sp**

Out[]: **False**

The **in** operator uses different algorithms for lists and dictionaries. For lists, it uses a linear search algorithm. As a list gets longer, the search time gets longer in direct proportion to the length of the list. For dictionaries, Python

uses an algorithm called a hash table, so, the in operaor takes about the same amount of time no matter how many items there are in a dictionary.

10.8.3.2 Dictionary Methods

Various methods associated with dictionary are listed below:

TABLE 16: Various Dictionary Methods

Dictionary Methods	Syntax	Description
clear()	dictionary_name.clear()	The clear() method removes all the key:value pairs from the dictionary.
fromkeys()	dictionary_name.fromkeys (seq[, value])	The fromkeys() method creates a new dictionary from the given sequence of elements with a value provided by the user.
get()	dictionary_name.get(key[, default])	The get() method returns the value associated with the specified key in the dictionary. If the key is not present then it returns the default value. If default is not given, it defaults to None, so that this method never raises a KeyError.
items()	dictionary_name.items()	The items() method returns a new view of dictionary's key and value pairs as tuples.
keys()	dictionary_name.keys()	The keys() method returns a new view consisting of aall the keys in the dictionary.
pop()	dictionary_name.pop(key[, default])	The pop() method removes the key from the dictionary and returns its value. If the key is not present, then it returns the default value. If the default is not given and the key is not in the dictionary, then it results in KeyError.
popitem()	dictionary_name.popitem( )	The popitem() methos removes and returns an arbitrary (key, value) tuple pair from the dictionary. If the dictionary is empty, then calling popitem() results in KeyError.
setdefault()	dictionary_name.setdefault (key[, default])	The setdefault() method returns a value for the key present in the dictionary. If the key is not present, then insert the key into the dictionary with a default value and return the default value. If key is present, default defaults to None, so that this methos never raises a KeyError.
update()	dictionary_name.update([o	The update() method updates the dictionary with the key:value pairs from other dictionary object and it

	ther])	returns None.
values()	dictionary_name.values()	The values() method returns a new view cons all the values in the dictionary.

Note: Replace the word “dictionary_name” mentioned in the syntax with the actual dictionary name.

In[]: **million_dollar = {“sanju” : 2018, “tiger zindahai” : 2017, “baahubali 2” : 2017, “dangal” : 2016, “bajrangibhaijaan” : 2015}**

In[]: **bolwd_million_dollar = million_dollar.fromkeys(million_dollar, “1,00,00,000”)**

In[]: **bolwd_million_dollar**

Out[]: **{'sanju': '1,00,00,000',**

'tiger zindahai': '1,00,00,000',

'baahubali 2': '1,00,00,000',

'dangal': '1,00,00,000',

'bajrangibhaijaan': '1,00,00,000'}

In[]: **million_dollar.get("bahubali 1")**

In[]: **million_dollar.get("bahubali 1", 2015)**

Out[]: **2015**

In[]: **million_dollar.keys()**

Out[]: **dict_keys(['sanju', 'tiger zindahai', 'baahubali 2', 'dangal', 'bajrangibhaijaan'])**

In[]: **million_dollar.values()**

Out[]: **dict_values([2018, 2017, 2017, 2016, 2015])**

In[]: **million_dollar.items()**

Out[]: **dict_items([('sanju', 2018), ('tiger zindahai', 2017), ('baahubali 2', 2017), ('dangal', 2016), ('bajrangibhaijaan', 2015)])**

In[]: **million_dollar.update({"bahubali 1" : 2015})**

In[]: **million_dollar**

Out[15]:

{'sanju': 2018,

'tiger zindahai': 2017,

'baahubali 2': 2017,

'dangal': 2016,

'bajrangibhaijaan': 2015,

'bahubali 1': 2015}

One of the common ways of populating dictionaries is to start with an empty dictionary{}, then use the **update()** method to assign a value to the key using assignment operator. If the key doesnot exist, then the key:value pairs will be created automatically and added to the dictionary.

In[]: **states ={}**

In[]: **states.update({"Haryana":"Chandigarh"})**

In[]: **states.update({"Bihar":"Patna"})**

In[]: **states.update({"west bengal" : "kolkata"})**

In[]: **states**

Out[23]: **{'Haryana': 'Chandigarh', 'Bihar': 'Patna', 'west bengal': 'kolkata'}**

If a dictionary is used as the sequence in a for statement, it traverses the keys of the dictionary. This loop prints each key and the corresponding value:

In[] : **for key in states :**

print(key, states[key])

Out[]:

Haryana Chandigarh

Bihar Patna

west bengalkolkata

Program 10.4 : Write a Program that Accepts a Sentence and Calculate the Number of Digits, Uppercase and Lowercase Letters

```
1. sentence = input("Enter a sentence : ")
2. construct_dictionary = {"digits":0, "lowercase":0, "uppercase":0}
3. for each_character in sentence :
4.     if each_character.isdigit() :
5. construct_dictionary["digits"] += 1
6. elifeach_character.isupper() :
7. construct_dictionary["uppercase"] += 1
8. elifeach_character.islower() :
9. construct_dictionary["lowercase"] += 1
10. print("The number of digits, lowercase and uppercase letters are")
```

11. print(construct_dictionary)

Fig. 4: Screen Shot of execution of Program 10.4

```
sentence = input("Enter a sentence : ")
construct_dictionary = {"digits":0, "lowercase":0, "uppercase":0}
for each_character in sentence :
  if each_character.isdigit() :
    construct_dictionary["digits"] += 1
  elif each_character.isupper() :
    construct_dictionary["uppercase"] += 1
  elif each_character.islower() :
    construct_dictionary["lowercase"] += 1
print("The number of digits, lowercase and uppercase letters are")
print(construct_dictionary)
```

```
Enter a sentence : Ask not what your country can do for you; ask what you can do for your country, John Kennedy, 1961
The number of digits, lowercase and uppercase letters are
{'digits': 4, 'lowercase': 69, 'uppercase': 3}
```

To delete the key:value pair, use the **del** statement followed by the name of the dictionary along with the key you want to delete.

In[]: **del states[“west bengal”]**

In[]: **states**

Out[]: **{‘Haryana’: ‘Chandigarh’, ‘Bihar’ : ‘Patna’}**

10.8.3.3 Relation between Tuples and Dictionaries

Tuples can be used as key:value pairs to build dictionaries.

In[]:**pm_year = (('jln',1947),('lbs',1964),('ig',1966),('rg',1984),('pvn',1991),('abv',1998),('nm',2014))**

In[]: **pm_year**

Out[]:
(('jln', 1947),
('lbs', 1964),
('ig', 1966),
('rg', 1984),
('pvn', 1991),
('abv', 1998),
('nm', 2014))

In[]: **pm_year_dict = dict(pm_year)**

In[]: **pm_year_dict**

Out[]:

{'jln': 1947,
'lbs': 1964,
'ig': 1966,
'rg': 1984,
'pvn': 1991,
'abv': 1998,
'nm': 2014}

The tuples can be converted to dictionaries by passing the tuple name to the **dict()** function. This is achieved by nesting tuples within tuples, wherein each nested tuple item should have two items in it. The first item becomes the key and the second item as its value when the tuple gets converted to a dictionary.

Check your progress - 8:

Q43. In Python, Dictionaries and its keys both are immutable.

a) a)True, True
b) True, False
c) False, True
d) False, False

Q44. Select correct ways to create an empty dictionary:

a) sampleDict = {}
b) sampleDict = dict()
c) sampleDict = dict{}

Q45. Items are accessed by their position in a dictionary and all the keys in a dictionary must be of the same type.

a) True
b) False

Q46. To obtain the number of entries in dictionary, d which command do we use?

a) d.size()
b) len(d)
c) size(d)
d) d.len()

Q47. Which one of the following is correct?

a) In python, a dictionary can have two same keys with different values.
b) In python, a dictionary can have two same values with different keys.
c) In python, a dictionary can have two same keys or same values but cannot have two same key-value pair.
d) In python, a dictionary can neither have two same keys nor two same

values.

Q48. What will be the output of above Python code?

```
d1={"abc":5,"def":6,"ghi":7}
print(d1[0])
```

a) abc
b) 5
c) {"abc":5}
d) Error

Q49. What will the above Python code do?

```
dict={"Phy":94,"Che":70,"Bio":82,"Eng":95}
dict.update({"Che":72,"Bio":80})
```

a) It will create new dictionary as dict={"Che":72,"Bio":80} and old dict will be deleted.
b) It will throw an error as dictionary cannot be updated.
c) It will simply update the dictionary as dict={"Phy":94,"Che":72,"Bio":80,"Eng":95}
d) It will not throw any error but it will not do any changes in dict

Q50. Select all correct ways to remove the key 'marks' from a dictionary

```
student = {
    "name" : "Emma",
    "class" : 9,
    "marks" : 75
}
```

a) student.pop("marks")
b) del student["marks"]
c) student.popitem()
d) dict1.remove("key2")

Q51. Select all correct ways to copy a dictionary in Python

a) dict2 = dict1.copy()
b) dict2 = dict(dict1.items())
c) dict2 = dict(dict1)
d) dict2 = dict1

10.8.4Sets

A **set** is an unordered collection with no duplicate items. Primary uses of sets include membership testing and eliminating duplicate entries. Sets also support mathematical operations, such as union, intersection, difference, and symmetric difference.

Curly braces{} or the set() function can be used to create sets with a comma-separated list of items inside curly brackets{}. Note: to create an empty set you have to use set() and not{} as the latter creates an empty dictionary.

Sets are mutable. Indexing is not possible in sets, since set items are unordered. One cannot access or change an item of the set using indexing or slicing.

In[]: **basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}**

In[]: **basket**

Out[]: **{'apple', 'banana', 'orange', 'pear'}**

A set is a collection of unique items. Duplicate items are removed from the set basket. Here, the set will contain only one item of 'orange' and 'apple'.

In[]: **'orange' in basket**

Out[]: **True**

In[]: **'grapes' in basket**

Out[]: **False**

One can test for the presence of an item in a set using **in** and not in membership operators.

In[]: **len(basket)**

Out[]: **4**

In[]: **sorted(basket)**

Out[]: **['apple', 'banana', 'orange', 'pear']**

Total number of items in the set basket is found using the len() function. The sorted() function returns a new sorted list from items in the set.

In[]: **a = set('abrakadabra')**

In[]: **a**

Out[]: **{'a', 'b', 'd', 'k', 'r'}**

In[]: **b = set('alexander')**

In[]: **b**

Out[]: **{'a', 'd', 'e', 'l', 'n', 'r', 'x'}**

In[]: **a-b** #letters present in set a, but not in set b

Out[]: **{'b', 'k'}**

In[]: **a | b** #Letters present in set a, set b, or both

Out[]: **{'a', 'b', 'd', 'e', 'k', 'l', 'n', 'r', 'x'}**

In[]: **a & b** #letters present in both set a and set b

Out[]: **{'a', 'd', 'r'}**

In[]: **a ^ b** #letters present in set a or set b, buth not both

Out[]: **{'b', 'e', 'k', 'l', 'n', 'x'}**

10.8.4.1 Set Methods

A list of all the methods associated with the set can be obtained by passing the set function to dir().

Various methods associated with set are listed in the TABLE.

TABLE 17 : Various Set Methods

Set Methods	Syntax	Description
add()	set_name.add(item)	The add() method adds an item to the set set_name.
clear()	set_name.clear()	The clear() method removes all the items from the set set_name.
difference()	set_name.difference(*others)	The difference() method returns a new set with items in the set set_name that are not in the others sets.
discard()	set_name.discard(item)	The discard() method removes an item from the set set_name if it is present.
intersection()	set_name.intersection(*others)	The intersection() method returns a new set with items common to the set set_name and all other sets.
isdisjoint()	set_name.isdisjoint(other)	The isdisjoint() method returns True if the set set_name has no items in common with other set. Sets are disjoint if and only if their intersection is the empty set.
issubset()	set_name.issubset(other)	The issubset() method returns True if every item in the set set_name is in the other set.
issuperset()	set_name.issuperset(other)	The issuperset() method returns True if every element in other set is in the set set_name.
pop()	set_name.pop()	The method pop() removes and returns an arbitrary item from the set set_name. It raises KeyError if the set is empty.
remove()	set_name.remove(item)	The method remove() removes an item from the set set_name. It raises KeyError if

		the item is not contained in the set.
symmetric_difference()	set_name.symmetric_difference(other)	The method symmetric_difference() returns a new set with items from the set set_name and all other sets.
union()	set_name.union(*others)	The method union() returns a new set with items from the set set_name and all others sets.
update()	set_name.update(*others)	Update the set set_name by adding items from all others sets

Note: Replace the words "set_name", "other" and "others" mentioned in the syntax with your actual set names in your code.

In[]: **flowers1 = {'sunflower', 'roses', 'lavender', 'tulips', 'goldcrest'}**

In[]: **flowers2 = {'roses', 'tulips', 'lilies', 'daisies'}**

In[]: **flowers2.add('orchids')**

In[]: **flowers2.difference(flowers1)**

Out[]: **{'daisies', 'lilies', 'orchids'}**

In[]: **flowers2.intersection(flowers1)**

Out[]: **{'roses', 'tulips'}**

In[]: **flowers2.isdisjoint(flowers1)**

Out[]: **False**

In[]: **flowers2.issuperset(flowers1)**

Out[]: **False**

In[]: **flowers2.issubset(flowers1)**

Out[]: **False**

In[]: **flowers2.symmetric_difference(flowers1)**

Out[]: **{'daisies', 'goldcrest', 'lavender', 'lilies', 'orchids', 'sunflower'}**

In[]: **flowers2.union(flowers1)**

Out[]:

{'daisies',
'goldcrest',
'lavender',

```
 'lilies',
 'orchids',
 'roses',
 'sunflower',
 'tulips'}
```

In[]: **flowers2.update(flowers1)**

Out[]: **flowers2**

Out[]:

```
{'daisies',
 'goldcrest',
 'lavender',
 'lilies',
 'orchids',
 'roses',
 'sunflower',
 'tulips'}
```

In[]: **flowers2.discard("roses")**

In[]: **flowers2**

Out[]:

```
{'daisies',
 'goldcrest',
 'lavender',
 'lilies',
 'orchids',
 'sunflower',
 'tulips'}
```

In[]: **flowers1.pop()**

Out[]: **'roses'**

In[]: **flowers2.clear()**

In[]: **flowers2**

Out[]: **set()**

10.8.4.2Traversing of Sets

One can iterate through each item in a set using a for loop.

In[]: **for flower in flowers1 :**

print(f"{flower} is a flower")

Out[]: **lavender is a flower**

tulips is a flower

goldcrest is a flower

sunflower is a flower

10.8.4.3Frozenset

A **frozenset** is basically the same as a set, except that it is immutable. Once a frozenset is created, its items cannot be changed. Since they are immutable, they can be used as members in other sets and as dictionary keys. The frozensets have the same functions as normal sets, except none of the functions that change the contents (update, remove, pop, etc.) are available.

In[]: **fs = frozenset(["g","o","o","d"])**
#creating/declaring frozenset

In[]: **fs** #displaying the contents of frozenset

Out[]: **frozenset({'d', 'g', 'o'})**

In[]: **pets = set([fs, "horse", "cow"])** #Frozenset type is used within a set

In[]: **pets**

Out[]: **{'cow', frozenset({'d', 'g', 'o'}), 'horse'}**

In[]: **lang_used = {"english":59, "french":29, "spanish":21}**

In[]: **frozenset(lang_used)** #keys in a dictionary are returned when a dictionary is passed as an argument to frozenset() function.

Out[]: **frozenset({'english', 'french', 'spanish'})**

In[]: **frs = frozenset(["german"])**

#Frozenset is used as a key in dictionary

In[]: **lang_used = {"english":59, "french":29, "spanish":21, frs:6}**

In[]: **lang_used**

Out[]: **{'english': 59, 'french': 29, 'spanish': 21, frozenset({'german'}): 6}**

Check your progress - 9:

Q52. Select which is true for Python set:

a) Sets are unordered.
b) Set doesn't allow duplicate
c) sets are written with curly brackets {}
d) set Allows duplicate
e) set is immutable
f) a set object does support indexing

Q53. What is the output of the following union operation?:

```
set1 = {10, 20, 30, 40}
set2 = {50, 20, "10", 60}
set3 = set1.union(set2)
print(set3)
```

a) {40, 10, 50, 20, 60, 30}
b) {40, '10', 50, 20, 60, 30}
c) {40, 10, '10', 50, 20, 60, 30}
d) SynatxError: Different types cannot be used with sets

Q54. What is the output of the following set operation?:

```
set1 = {"Yellow", "Orange", "Black"}
set2 = {"Orange", "Blue", "Pink"}
set3 = set2.difference(set1)
print(set3)
```

a) {'Yellow', 'Black', 'Pink', 'Blue'}
b) {'Pink', 'Blue'}
c) {'Yellow', 'Black'}

Q55. What is the output of the following set operation?:

```
set1 = {"Yellow", "Orange", "Black"}
set1.discard("Blue")
print(set1)
```

a) {'Yellow', 'Orange', 'Black'}
b) KeyError: 'Blue'

Q56. The isdisjoint() method returns True if none of the items are present in both sets, otherwise, it returns False.

a) True
b) False

Q57. Select all the correct ways to copy two sets

a) set2 = set1.copy()
b) set2 = set1
c) set2 = set(set1)
d) set2.update(set1)

Q58. The symmetric_difference() method returns a set that contains all items from both sets, but not the items that are present in both sets.

a) False
b) True

10.9 CONTROL FLOW STATEMENTS

Python supports a set of control flow statements that you can integrate into your program. The statements inside your Python program are generally executed sequentially from top to bottom in the order that they appear. Apart from sequential control flow statements, you can employ decision making and looping control flow statements to break up the flow of execution thus enabling your program to conditionally execute particular block of code. The term control flow details the direction the program takes.

The control flow statements in Python Programming Language are:

10.9.1 Sequential Control Flow Statements : This refers to the line by line execution, in which the statements are executed sequentially, in the same order in which they appear in the program.

10.9.2 Decision Control Flow Statements : Depending on whether a condition is True or False, the decision structure may skip the execution of an entire block of statements or even execute one block of statements instead of other (if, if…else and if…elif…else).

10.9.3 Loop Control Flow Statements : This is a control structure that allows the execution of a block of statements multiple times until a loop termination condition is met (for loop and while loop). Loop Control Flow statements are also called Repetition statements ot Iteration Statements.

10.9.2.1 The if Decision Control Flow Statement

In order to write useful programs, we almost need the ability to check conditions and change the behavior of the program accordingly. The Decision Control Flow Statements give us this ability. The simplest form is the if statement.

The syntax for if statement is:

```
if  Boolean_Expression :
        statement (s)
```

The Boolean expression after the if statement is called the condition. We end the if statement with a colon character (:) and the line(s) after the if statement are indented. The if statement decides whether to run statement (s) or not depending upon the value of the Boolean expression. If the Boolean expression evaluates to True then indented statements in the if block will be executed,otherwise if the result is False then none of the statements are executed. E.g.,

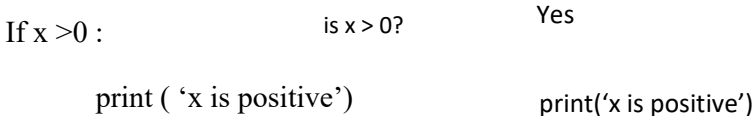

Fig. 5: If Logic

Here, depending on the value of x, print statement will be executed i.e. only if x > 0.

There is no limit on the number of statements that can appear in the body, but there must be at least one. In Python, the if block statements are determined through indentation and the first unindented statement marks the end. You don't need to use the == operator explicitly to check if the variable's value evaluates to True as the variable name can itself be used as a condition.

Program 10.5:Program reads your age and prints a message whether you are eligible to vote or not???

```
1. age = int(input("Enter your age : "))
2. if age >= 18 :
3.       print ("Congratulations!!! you can vote")
4.       print ("!!! Thanks !!!")
5. print ("Voting is your birth right. Use judicially")
```

Fig. 6: Screen Shot of execution of Program 10.5

```
age = int(input("Enter your age : "))
if age >= 18 :
  print ("Congratulations!!! you can vote")
  print ("!!! Thanks !!!")
print ("Voting is your birth right. Use judicially")


Enter your age : 22
Congratulations!!! you can vote
!!! Thanks !!!
Voting is your birth right. Use judicially
```

Here, condition or Boolean expression is true, indented block would be executed

```
Enter your age : 17
Voting is your birth right. Use judicially
```

Here, condition is False, so indented block would not be executed and only the unindented statement after the if block would be executed.

10.9.2.2 The if…else Decision Control Flow Statement

An if statement can also be followed by an else statement which is optional. An else statement does not have any condition. Statements in the if block are

executed if the Boolean_Expression is True. Use the optional else block to execute statements if the Boolean_Expression is False. The if…else statements allow for a two – way decision.

The syntax for if…else statement is:

```
if Boolean_Expression :
        statement_blk_1
else
        statement_blk_2
```

If the Boolean_Expression evaluates to True, then statement_blk_1 (single or multiple statements) is executed, otherwise it is evaluated to False then statement_blk_2 (single or multiple statements) is executed. Indentation is used to separate the blocks. After the execution of either statement_blk_1 or statement_blk_2, the control is transferred to the next statement after the if statement. Also, if and else keywords should be aligned at the same column position.

Program 10.6 : Program to find if a given number is odd or even

```
1. num = int(input("Enter a number : ")
2. if num % 2 == 0 :
3.         print (num," is even number")
4. Else:
5.         print (num,"  is odd number")
```

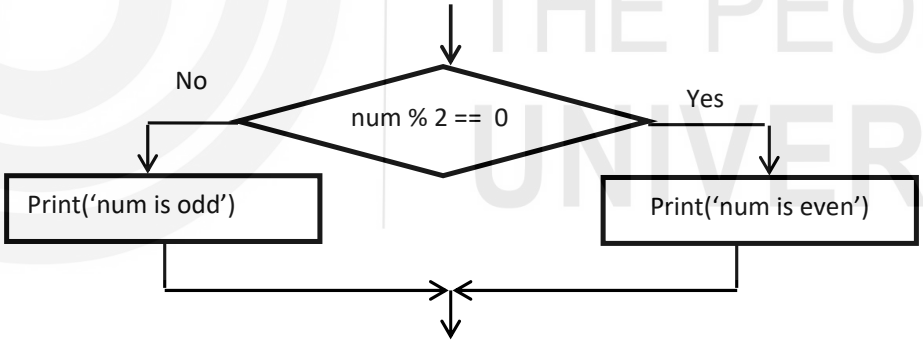

Fig. 6: IF – ELSE LOGIC

A number is read and stored in the variable named num. The num is checked using modulus operator to determine whether the num is perfectly divisible by 2 or not. Num entered is 15 which makes the evaluated expression False, so the else statement is executed and num is odd.

Fig. 7: Screen Shot of execution of Program 10.6

```
num = int(input("Enter a number : "))
if num % 2 == 0:
  print (num," is even number")
else:
  print (num,"  is odd number")
```

```
Enter a number : 15
15   is odd number
```

10.9.2.3 The if…elif…else Decision Control Statement

The if…elif…else is also called as multi-way decision control statement. When you need to choose from several possible alternatives, an elif statement is used along with an if statement. The keyword 'elif' is short for 'else if' and is useful to avoid excessive indentation. The else statement must always come last, and will again act as the default action.

The syntax for if…elif…else statement is,

```
if Boolean_Expression_1 :
        statement_blk_1
elif Boolean_Expression_2 :
        statement_blk_2
elif Boolean_Expression_3 :
        statement_blk_3
        :
        :
        :
else :
        statement_blk_last
```

This if…elif…else decision control statement is executed as follows:

- In the case of multiple Boolean expression, only the first logical Boolean expression which evaluates to True will be executed.
- If Boolean_Expression_1 is True, then statement_blk_1 is executed.
- If Boolean_Expression_1 is False and Boolean_Expression_2 is True, then statement_blk_2 is executed.
- If Boolean_Expression_1 and Boolean_Expression_2 is False and Boolean_Expression_3 is True, then statement_blk_3 is executed and so on.
- If none of the Boolean_Expression is True, then statement_blk_last is executed.

Program 10.7 : Write a program to prompt for a score between 0.0 and 1.0. If the score is out of range, print an error. If the score is between 0.0 and 1.0, print a grade using the following table

Score	Grade
>= 0.9	A
>= 0.8	B
>= 0.7	C
>= 0.6	D
< 0.6	F

```
1. score = float(input("Enter your score : "))
2. if score < 0 or score >1 :
3. print('Wrong Input')
4. elif score >= 0.9 :
5. print('Your Grade is "A"')
6. elif score >= 0.8 :
7. print('Your Grade is "B"')
8. elif score >= 0.7 :
9. print('Your Grade is "C"')
10. elif score >= 0.6 :
11. print('Your Grade is "D"')
12. else :
13. print('Your Grade is "F"')
```

if score < 0 or score >1 — Yes — Wrong Input'

score >= 0.9 — Yes — Your Grade is "A"

score >= 0.8 — Yes — Your Grade is "B"

>= 0.7 — Yes — Your Grade is "C"

>= 0.6 — Yes — Your Grade is "D"

le is "F"

Fig. 9: Screen Shot of execution of Program 10.7

```
score = float(input("Enter your score : "))
if score < 0 or score > 1 :
  print('Wrong Input')
elif score >= 0.9 :
  print('Your Grade is "A"')
elif score >= 0.8 :
  print('Your Grade is "B"')
elif score >= 0.7 :
  print('Your Grade is "C"')
elif score >= 0.6 :
  print('Your Grade is "D"')
else :
  print('Your Grade is "F"')
```

```
Enter your score : 0.82
Your Grade is "B"
```

Fig.8 : IF – ELIF - ELSE LOGIC

10.9.2.4 Nested if Statement

In some situations, you have to place an if statement inside another statement. An if statement that contains another if statement either in its if block or else block is called a Nested if statement.

The syntax of the nested if statement is,

```
if Boolean_Expression_1 :
        if Boolean_Expression_2 :
                statement_blk_1
        else :
                statement_blk_2
else :
        statement_blk_3
```

If the Boolean_Expression_1 is evaluated to True, then the control shifts to Boolean_Expression_2 and if the expression is evaluated to True, then statement_blk_1 is executed.If the Boolean_Expression_2 is evaluated to False then the statement_blk_2 is executed. If the Boolean_Expression_1 is evaluated to False, then the statement_blk_3 is executed.

A three – branch example could be written as :

```
if x == y :
        print('x and y are equal')
else :
        if x <y :
                print('x is less than y')
        else :
                print('x is greater than y')
```

The outer conditional contains two branches. The first branch contains a simple statement. The second branch contains another if statement has two branches of its own. Those two branches are both simple statements, although they could have been conditional statements as well.

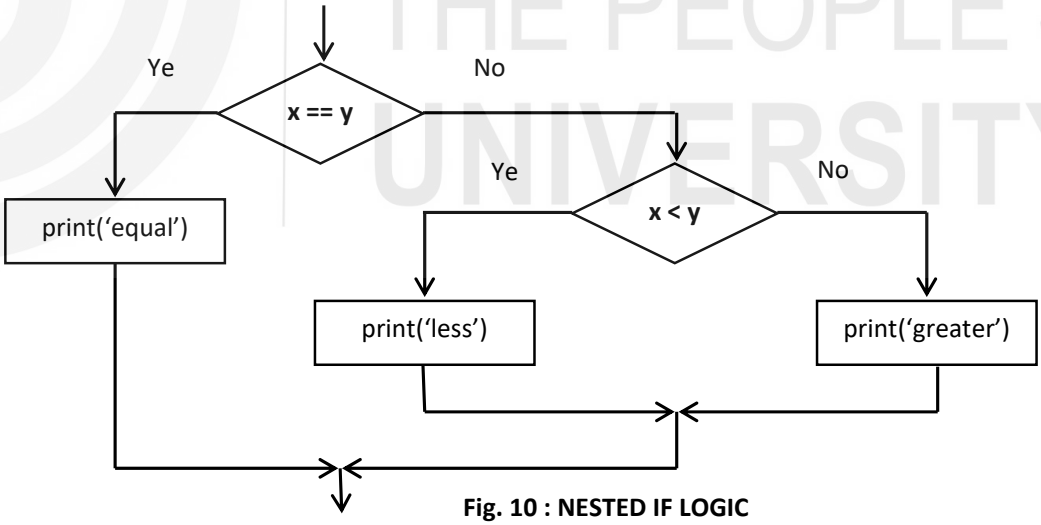

Fig. 10 : NESTED IF LOGIC

PROGRAM 10.8 : Program to check if a given year is a leap year

```
1. year = int(input('Enter a year : '))
2. if year % 4 == 0 :
3.         if year % 100 == 0 :
4.                 if year % 400 == 0 :
5.                         print(f'{year} is a Leap Year')
6. else :
7. printf(f'{year} is not a Leap Year')
```

```
8.  else :
9.                  print(f'{year} is a Leap Year')
10. else :
11.         print(f'{year} is not a Leap Year')
```

Fig. 11: Screen Shot of execution of Program 10.8

```
year = int(input('Enter a year : '))
if year % 4 == 0:
  if year % 100 == 0 :
    if year % 400 == 0 :
      print(f'{year} is a Leap Year')
    else :
      printf(f'{year} is not a Leap Year')
  else :
    print(f'{year} is a Leap Year')
else :
  print(f'{year} is not a Leap Year')
```

```
Enter a year : 2014
2014 is not a Leap Year
```

All years which are perfectly divisible by 4 are leap years except for century years (years ending with 00) which is a leap year only it is perfectly divisible by 400. For example, years like 2012, 2004, 1968 are leap years but 1971, 2006 are not leap years. Similarly, 1200, 1600, 2000, 2400 are leap years but 1700, 1800, 1900 are not.

Although the indentation of the statements makes the structure apparent, nested conditional becomes difficult to read very quickly. In general, it is a good idea to avoid them when you can.

10.9.3.1 The while loop

Computers are often used to automate repetitive tasks. Repeating identical or similar tasks without making error is something that computers do well and people do poorly. Because iteration is so common, Python provides several language features to make it easier.

Orogrne form of iteration in Python is the while statement. The syntax for while loop is :

while Boolean_Expression :

statement(s)

Program 10.9 : Program that counts down from five and then says "Blastoff!"

```
1. n = 5
2. while n >0 :
```

```
3.          print(n)
4.          n = n – 1
5.  print('Blastoff!')
```

Fig. 12: Screen Shot of execution of Program 10.9

```
n = 5
while n > 0 :
  print(n)
  n = n - 1
print('Blastoff!')
```

```
5
4
3
2
1
Blastoff!
```

You can almost read the while statement as if it were English. It means, "While n is greater than 0, display the value of n and then reduce the value of n by 1. When you get to 0, exit the while statement and display the word Blastoff!"

More formally, the flow of execution for a while statement is :

1. Evaluate the condition or the Boolean_Expression, yielding True or False.
2. If the Boolean_Expression is false,exit the while statement and then continue execution at the next statement.
3. If the Boolean_Expression is true, execute the body and then go back.

This type of flow is called a loop because the third step loops back around to the top. We call each time we execute the body of the loop an iteration. for the above loop, we would say, "It had five iterations", which means that the body of the loop was executed five times.

The body of the loop should change the value of one or more variables so that eventually the condition or the Boolean_Expression becomes false and the loop terminates. We call the variable that changes each time the loop executes and controls when the loop finishes the iteration variable. If there is no iteration variable, the loop will repeat forever, resulting in an infinite loop.

PROGRAM 10.10 : Program to display the Fibonacci Sequences up to nth term where n is provided by the user

```
1.  nterms = int(input('How many terms?'))
2.  current = 0
3.  previous = 1
```

```
4.  next_term = 0
5.  count = 0
6.  if nterms<= 0 :
7.  print('Please enter a positive number')
8.  elifnterms == 1 :
9.  print('Fibonacci Sequence')
10. print('0')
11. else :
12. print("Fibonacci Sequence")
13.         while count <nterms :
14.                 print(next_term)
15.                 current = next_term
16. next_term = previous + current
17.         previous = current
18.                 count += 1
```

Fig. 13: Screen Shot of execution of Program 10.10

```
nterms = int(input('How many terms? '))
current = 0
previous = 1
next_term = 0
count = 0
if nterms <= 0 :
  print('Please enter a positive number')
elif nterms == 1 :
  print('Fibonacci Sequence')
  print('0')
else :
  print("Fibonacci Sequence")
  while count < nterms :
    print(next_term)
    current = next_term
    next_term = previous + current
    previous = current
    count += 1
```

```
How many terms? 5
Fibonacci Sequence
0
1
1
2
3
```

In a Fibonacci sequence, the next number is obtained by adding the previos two numbers. The first two numbers of the Fibonacci sequence are 0 and 1. The next number is obtained by adding 1 and 1 which is 1. Again, the next number is obtained by adding 1 and 1 which is 2 and so on. Get a number from user up to which you want to generate Fibonacci sequence. Assign

values to variables current, previous, next_term and count. The variable count keeps track of number of times the while block is executed. User is required to enter a posititve number to generate a single number in the sequence, then print zero. The next_term is obtained by adding the previous and current variables and the statements in the while block are repeated until while block conditional expression becomes False.

10.9.3.2 The for loop

Sometimes we want to loop through a set of things such as a list of words, the lines in a file, or a list of numbers. When we have a list of things to loop through, we can construct a definite loop using a for statement. We call the while statement an indefinite loop because it simply loops until some condition becomes False, whereas the for loop is looping through a known set of items so it runs through as many iterations as there are items in the set.

The syntax for the for loop is :

for iteration_variable in sequence :

statement(s)

The for loop starts with for keyword and ends with a colon. The first item in the sequence gets assigned to the iteration variableiteration_variable. Here, iteration_variable can be any valid variable name. Then the statement block is executed. This process of assigning items from the sequence to the iteration_variable and then executing the statement continues until all the items in the sequence are completed.

The for loop is incomplete without the use of **range()** function which is a built-in function. It is very useful in demonstrating for loop. The range() function generates a sequence of numbers which can be iterated through using for loop. the syntax for range() function is,

range([start ,] stop [, step])

Both start and step arguments are optional and is represented with the help of square brackets and the range argument value should always be an iteger.

start → value indicates the beginning of the sequence. If the start argument is not specified, then the sequence of numbers start from zero by default.

stop → generates numbers up to this value but not including the number itself.

step → indicates the difference between every two consecutive numbers in the sequence. The step value can be both negative and positive but not zero.

PROGRAM 10.11 : Program to find the sum of all odd and even numbers up to a number specified by the user.

1. number = int(input("Enter a number"))
2. even = 0

```
3. odd = 0
4. for i in range(number) :
5.         if i % 2 == 0 :
6.                 even = even + i
7. else :
8.                 odd = odd + i
9. print(f"Sum of Even numbers are {even} and Odd numbers are
   {odd}")
```

Fig. 14: Screen Shot of execution of Program 10.11

```
number = int(input("Enter a number"))
even = 0
odd = 0
for i in range(number) :
  if i % 2 == 0 :
    even = even + i
  else :
    odd = odd + i
print(f"Sum of Even numbers are {even} and Odd numbers are {odd}")


Enter a number10
Sum of Even numbers are 20 and Odd numbers are 25
```

A range of numbers are generated using range() function. As only stop value is indicated in the range() function, so numbers from 0 to 9 are generated. Then the generated numbers are segregated as odd or even by using the modulus operator in the for loop. All the even numbers are added up and assigned to even variable and odd numbers are added up and assigned to odd variable and print the result. The for loop will be executed / iterated number of times entered by user which is 10 in this case.

PROGRAM 10.12 : Program to find the factorial of a number.

```
1. number = int(input('Enter a number'))
2. factorial = 1
3. if number <0 :
4. print("Factorial doesn't exist for negative numbers")
5. elif number == 0 :
6. print("The factorial of 0 is 1")
7. else :
8.         for i in range(1, number + 1) :
9.                 factorial = factorial * i
10. print(f"The factorial of number {number} is {factorial}")
```

Fig. 15: Screen Shot of execution of Program 10.12

```
number = int(input('Enter a number'))
factorial = 1
if number < 0 :
  print("Factorial doesn't exist for negative numbers")
elif number == 0 :
  print("The factorial of 0 is 1")
else :
  for i in range(1, number + 1) :
    factorial = factorial * i
print(f"The factorial of number {number} is {factorial}")
```

```
Enter a number5
The factorial of number 5 is 120
```

Read the number from user. A value of 1 is assigned to variable factorial. To find the factorial of a number it has to be checked for a non – negative integer. If the user entered number is zero then the factorial is 1. To generate numbers from 1 to the user entered number range() function is used. Every number is multiplied with the factorial variable and is assigned to the factorial variable inside the for loop. The for loop block is repeated for all the numbers starting from 1 up to the user entered number. Finally, the factorial value is printed.

10.9.3.3The continue and break Statements

The break and continue statements provide greater control over the execution of code in a loop.Whenever the break statement is encountered, the execution control immediately jumps to the first instruction following the loop. To pass control to the next iteration without exiting the loop, use the continue statement. Both continue and break statements can be used in while and for loops.

PROGRAM 10.13 : Program that prints integers from zero to 5.

```
1. count = 0
2. while True :
3.         count += 1
4.         if count >5 :
5.         break
6.         print (count)
```

Fig. 16: Screen Shot of execution of Program 10.13

```
count = 0
while True :
  count += 1
  if count > 5 :
    break
  print (count)
```

```
1
2
3
4
5
```

In the while loop, as soon as count value reaches 6, because of break statement print statement will not be executed and control will come out of the while loop.

PROGRAM 10.14 : Program that processes only odd integers from 0 to 10.

```
1. count = 0
2. while count <10 :
3.         count += 1
4.         if count  % 2 == 0 :
5.                 continue
6.         print (count)
```

Fig. 17: Screen Shot of execution of Program 10.14

```
count = 0
while count < 10 :
  count += 1
  if count  % 2 == 0 :
    continue
  print (count)
```

```
1
3
5
7
9
```

The continue statement in the while loop will not let the print statement to execute if the count value is even and control will go for next iteration. Here, loop will go through all the iterations and the continue statement will affect only the statements in the loop to be executed or not occurring after the continue statement. Whereas, the break statement will not let the loop to complete its iterations depending on the condition specified and control is transferred to the statement outside the loop.

PROGRAM 10.15 : Program to check whether a number is prime or not

```
1. number = int(input('Enter a number: '))
2. prime = True
3. for i in range(2, number) :
4.         if number % i == 0 :
5.                 prime = False
6.                 break
7. if prime :
8.         print(f"{number} is a prime number")
9. else :
10.        print(f"{number} is not a prime number")
```

Fig. 18: Screen Shot of Program 10.15

```
number = int(input('Enter a number: '))
prime = True
for i in range(2, number) :
  if number % i == 0 :
    prime = False
  break
if prime :
  print(f"{number} is a prime number")
else :
  print(f"{number} is not a prime number")


Enter a number: 9
9 is a prime number
```

Check your progress-10 :

Q59. In a python program, a control structure:

a) Defines program-specific data structures
b) Directs the order of ececution of the statements in the program
c) Dictates what happens before the program starts and after it terminates
d) None of the above

Q60. Which of the following is False regarding loops in python?

a) Loops are used to perform certain tasks repeatedly
b) While loop is used when multiple statements are to executed repeatedly until the given condition becomes False
c) While loop is used when multiple statemenst are to executed repeatedly until the given condition becomes True.
d) for loop can be used to iterate through the elements of lists.

Q61. We can write if/else into one line in python.

a) True
b) False

Q62. Given the nested if-else below, what will be the value x when the code executed successfully?

x = 0

```
a = 5
b = 5
if a > 0:
    if b < 0:
        x = x + 5
elif a > 5:
        x = x + 4
    else:
        x = x + 3
else:
    x = x + 2
print(x)
```

a) 0
b) 4
c) 2
d) 3

Q63. if -3 will evaluate to true

a) True
b) False

Q64. What is the output of the following nested loop

```
numbers = [10, 20]
items = [“Chair”, “Table”]
for x in numbers:
    for y in items:
print(x, y)
```

a) 10 Chair
10 Table
20 Chair
20 Table

b) 10 Chair
20 Chair
10 Table
20 Table

Q65. What is the value of x?:

```
X = 0
While (x < 100):
    x+=2
print(x)
```

a) 101
b) 99
c) None of the above, this is an infinite loop
d) 100

Q66. What is the value of the var after the for loop completes its execution?

```
var = 10
for i in range(10):
    for j in range(2, 10, 1):
        if var % 2 == 0:
            continue
            var += 1
    var+=1
else:
    var+=1
print(var)
```

a) 20
b) 21
c) 10
d) 30

Q67. Find the output of the following program:

```
a = {i :i * i for i in range(6)}
print (a)
```

Dictionary comprehension doesn't exist

a) {0:0, 1:1, 2:4, 3:9, 4:16, 5:25, 6:36}
b) {0:0, 1:1, 4:4, 9:9, 16:16, 25:25}
c) {0:0, 1:1, 2:4, 3:9, 4:16, 5:25}

10.10 SUMMARY

After the completion of this chapter, I am sure, you would be able to learn and understand the basics of Python language. Using the contents explained in this chapter, you would be able to turn your logic into codes where identifiers and variables would help you to form statements and expressions using operators, lists, tuples, sets and dictionries which are the back bone of Python language which makes it unique and easy to program with. All the basic structures of Python can be bound together with the control flow statements which ultimately form a Python program.However, this is not all; there is still a long way to go. Python has many more unique features which make it a programmer's language. So, Happy Programming!

SOLUTIONS TO CHECK YOUR PROGRESS

Answers to Check your Progress 1 to 10 :					
1. a	2. d	3. d	4. a	5. c	6. d
7. b	8. e	9. b	10. a	11. b	12. a
13. d	14. c	15. b	16. e	17. d	18. a
19. a	20. d	21. b	22. c	23. c	24. c
25. b	26. b	27. b	28. b	29. c	30. b
31. c	32. b	33. b	34. c	35. a	36. b
37. c	38. d	39. a,c	40. b	41. b	42. c
43. c	44. a,b	45. b	46. b	47. b	48. d
49. c	50. a,b,c	51. a,b,c	52. a,b,c	53. c	54. b
55. a	56. a,c,d	57. a,c,d	58. b	59. b	60. b
61. a	62. d	63. a	64. a	65. d	66. b
67. d					