F.Y.B.Sc.
(Computer Science)
SEMESTER - I (CBCS)

PROGRAMMING WITH PYTHON - I

SUBJECT CODE : USCS102

Prof. Suhas Pednekar
Vice-Chancellor,
University of Mumbai,

Prof. Ravindra D. Kulkarni   Prof. Prakash Mahanwar
Pro Vice-Chancellor,   Director,
University of Mumbai,   IDOL, University of Mumbai,

Programme Co-ordinator : **Shri Mandar Bhanushe**
Head, Faculty of Science and Technology,
IDOL, University of Mumbai, Mumbai

Course Co-ordinator   : **Mr. Sumedh Shejole**
Asst. Professor,
IDOL, University of Mumbai, Mumbai

Course Writers   : **Mlind Thorat**
Asst. Professor,
KJSIEIT, Sion, ,Mumbai

: **Mr. Santosh Patil Shirke**
Asst. Professor,
SJRS College, Kalwa

December 2021, Print - I

Published by   : Director
Institute of Distance and Open Learning ,
University of Mumbai,
Vidyanagari, Mumbai - 400 098.

DTP Composed and   :   Mumbai University Press
Printed by   Vidyanagari, Santacruz (E), Mumbai - 400098

CONTENTS

❖ ❖ ❖ ❖

Syllabus
F.Y.B.Sc. (CS) Programming with Python - I
Semester I (CBCS)

Objectives

The objective of this paper is to introduce various concepts of programming to the students using Python.

Expected learning outcomes
1) Students should be able to understand the concepts of programming before actually starting to write programs.
2) Students should be able to develop logic for Problem Solving.
3) Students should be made familiar about the basic constructs of programming such as data, operations, conditions, loops, functions etc.
4) Students should be able to apply the problem solving skills using syntactically simple language i.e. Python (version: 3.X or higher)

Unit I :

Reason for Python as the learner's first programming language. Introduction to the IDLE interpreter (shell) and its documentation. Expression evaluation: similarities and differences compared to a calculator; expressions and operators of types int, float, boolean. Built-in function type. Operator precedence.

Enumeration of simple and compound statements. The expression statement. The assert statement, whose operand is a boolean expression (values true or false). The assignment statement, dynamic binding of names to values, (type is associated with data and not with names); automatic and implicit declaration of variable names with the assignment statement; assigning the valueNone to a name. The del (delete) statement. Input/output with print and input functions. A statement list (semicolon separated list of simple statements on a single line) as a single interpreter command. The import statement for already-defined functions and constants. The augmented assignment statement. The built-inhelp() function.

Interactive and script modes of IDLE, running a script, restarting the shell. The compound statement def to define functions; the role of indentation for delimiting the body of a compound statement; calling a previously defined function. Compound data types str, tuple and list (enclosed in quotes, parentheses and brackets, respectively). Indexing individual elements within these types. Strings and tuples are immutable, lists are mutable. Built-in functions min, max, sum. Interactive solution of model problems, (e.g., finding the square root of a number or zero of a function), by repeatedly executing the body of a loop (where the body is a statement list).

Unit II

Advantages of functions, function parameters, formal parameters, actual parameters, global and local variables.

The range function, the iterative for statement. The conditional statements if, if-else, if-elif-else. The iterative statements while, while-else, for-else. The continue statement to skip over one iteration of a loop, the break statement to exit the loop. Nested compound statements. Dictionaries: concept of key-value pairs, techniques to create, update and delete dictionary items. Problem-solving using compound types and statements.

Unit III
Anonymous functions. List comprehensions. Gentle introduction to object-oriented programming; using the built-in dir() function, enumerate the methods of strings, tuples, lists, dictionaries. Using these methods for problem-solving with compound types.

Text books:
1. Magnus Lie Hetland, Beginning Python: From Novice to Professional, Apress
2. Paul Gries, et al., Practical Programming: An Introduction to Computer Science Using Python 3, Pragmatic Bookshelf, 2/E 2014

Additional References:
1. Charles Dierbach, Introduction to Computer Science using Python, Wiley, 2013
2. Paul Gries, Jennifer Campbell, Jason Montojo, Practical Programming: An Introduction to Computer Science Using Python 3, Pragmatic Bookshelf, 2/E 2014
3. Adesh Pandey, Programming Languages – Principles and Paradigms, Narosa, 2008

❖❖❖❖

1

PROGRAMMING WITH PYTHON- I

Unit Structure

1.1   Objective

1.2   Reasons for Python as the learner's first programming language.

1.3   Introduction to the IDLE interpreter (shell) and its documentation. Expression evaluation: similarities and differences compared to a calculator; expressions and operators of types int, float,boolean.

1.4   Built-in function type.

1.5   Operator precedence

1.6   Summary

1.7   Reference for further reading

1.8   Unit End Exercises

1.1 OBJECTIVE

1. Understand the fundamentals of writing Python scripts
2. Learn core Python scripting elements such as variables, expression, and operators
3. Use Python to input and output function
4. Understand the Built-in Functions

1.2 REASONS FOR PYTHON AS THE LEARNER'S FIRST PROGRAMMING LANGUAGE

1. Reasons for Python are the learner's first programming language are:

1. Presence of Third Party Modules:

The Python Package Index (PyPI) contains numerous third-party modules that make Python capable of interacting with most of the other languages and platforms.

2. Extensive Support Libraries:

Python provides a large standard library that includes areas like internet protocols, string operations, web service tools, and operating system interfaces. Many highly used programming tasks have already been scripted into the standard libraries which reduces the length of code to be written significantly.

3. Open Source and Community Development:

Python language is developed under an OSI-approved open source license, which makes it free to use and distribute, for commercial purposes also.

4. Learning Ease and Support Available:

Python offers excellent readability and uncluttered simple-to-learn syntax which helps beginners to easily use this programming language.

5. User-friendly Data Structures:

Python has built-in list and dictionary data structures that can be used to construct fast runtime data structures. Further, Python also provides an option of dynamic high-level data typing which reduces the length of support code.

6. Productivity and Speed:

Python supportsthe object-oriented design, provides enhanced process control capabilities,and possesses strong integration and text processing capabilities and,a unit testing framework all of which contribute to increasing its speed and productivity.

Applications of Python
- GUI based desktop applications
  - Image processing and graphic design applications
  - Scientific and computational applications
  - Games
- Web frameworks and web applications
- Enterprise and business applications
- Operating systems
- Language development
- Prototyping

1.3 INTRODUCTION TO THE IDLE INTERPRETER (SHELL) AND ITS DOCUMENTATION.

Introduction to the IDLE interpreter (shell) and its documentation

IDLE has the following features:

a. coded in 100% pure Python, using the Tkinter GUI toolkit

b. cross-platform: works mostly same on Windows, Unix, and macOS

c. Python shell window (interactive interpreter) with colorizing code input, output, and error messages

d. multi-window text editor with multiple undo, Python colorizing, smart indent, call tips, auto-completion, and other features

e. search within any window, replace within editor windows and search through multiple files (grep)

f. debugger with persistent breakpoints, stepping, and viewing of global and local namespaces configuration, browsers, and other dialogs

Menus

IDLE has two main window types, the Shell window, and the Editor window. It is possible to have multiple editor windows simultaneously. On Windows and Linux, each has its top menu. Each menu documented below indicates which window type it is associated with.

Output windows, such as used for Edit => Find in Files, are a subtype of editor window. They currently have the same top menu but a different default title and context menu.

On macOS, there is one application menu. It dynamically changes according to the window currently selected. It has an IDLE menu, and some entries described below are moved around to conform to Apple guidelines.

File menu
New File
Create a new file editing window.
Open…
Open an existing file with an Open dialog.
Recent Files
Open a list of recent files. Click one to open it.
Open Module…
Open an existing module (searches sys.path).
Class Browser
Show functions, classes, and methods in the current Editor file in a tree structure. In the shell, open a module first.
Path Browser
Show sys.path directories, modules, functions, classes and methods in a tree structure.
Save
Save the current window to the associated file, if there is one.
Close
Close the current window (ask to save if unsaved).
Exit
Close all windows and quit IDLE (ask to save unsaved windows).

Edit menu (Shell and Editor)
Undo
Undo the last change to the current window. A maximum of 1000 changes may be undone.
Redo

Redo the last undone change to the current window.
Cut
Copy selection into the system-wide clipboard; then delete the selection.
Copy
Copy selection into the system-wide clipboard.

Format menu (Editor window only)
Indent Region
Shift selected lines right by the indent width (default 4 spaces).
Dedent Region
Shift selected lines left by the indent width (default 4 spaces).
Comment Out Region
Insert ## in front of selected lines.
Uncomment Region
Remove leading # or ## from selected lines.

Run menu (Editor window only)

Run Module

Do Check Module. If no error, restart the shell to clean the environment, then execute the module. Output is displayed in the Shell window. Note that output requires use of print or write. When execution is complete, the Shell retains focus and displays a prompt. At this point, one may interactively explore the result of execution. This is similar to executing a file with python -i file at a command line.

Run… Customized

Same as Run Module, but run the module with customized settings. Command Line Arguments extend sys.argv as if passed on a command line. The module can be run in the Shell without restarting.

Check Module

Check the syntax of the module currently open in the Editor window. If the module has not been saved IDLE will either prompt the user to save or autosave, as selected in the General tab of the Idle Settings dialog. If there is a syntax error, the approximate location is indicated in the Editor window.
Python Shell
Open or wake up the Python Shell window.

2 Expression evaluation: similarities and differences compared to a calculator; expressions and operators of types int, float, boolean.

3. Expression evaluation:

Python's eval() allows to evaluate arbitrary Python expressions from a string-based or a compiled code object input . This function can be

used to dynamically evaluate Python expressions from any input that comes as a string or a compiled code object.

The eval() is defined as :

eval(expression[, globals[, locals]])

The function takes three arguments. The function takes a first argument, called expression, which holds the expression that you need to evaluate. eval() also takes two optional arguments:

1. globals

2. locals

The First Argument: expression

The first argument to eval() is called expression. It's a required argument that holds the string-based or compiled-code-based input to the function. When eval() function call , the content of expression is evaluated as a Python expression.

```
>>>eval("90 + 10")
100

>>>eval("sum([10, 50, 25])")
85

>>> x = 50

>>>eval("x * 2")
100
```

To evaluate a string-based expression, Python's eval() runs the following steps:

1.   Parse expression
2.   Compile it to bytecode
3.   Evaluate it as a Python expression
4.   Return the result of the evaluation

The Second Argument: globals

The second argument to eval() is called globals. It's optional and holds a dictionary that provides a global namespace to eval(). With globals, you can tell eval() which global names to use while evaluating expression. Global names are all those names that are available in your current global scope or namespace. All the names passed to globals in a dictionary will be available to eval() at execution time.

```
Python
>>>x1 = 100  # A global variable
>>>eval("x1 + 100", {"x1": x1})
200
>>> y1 = 200  # Another global variable
```

```python
>>>eval("x1 + y1", {"x1": x1})
```

Traceback (most recent call last):

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

NameError: name 'y1' is not defined

If you supply a custom dictionary to the globals argument of eval(), then eval() will take only those names as globals. Any global names defined outside of this custom dictionary won't be accessible from inside eval(). That's why Python raises a NameError when you try to access y1 in the above code: The dictionary passed to globals doesn't include y1.

You can insert names into globals by listing them in your dictionary, and then those names will be available during the evaluation process. For example, if you insert y1 into globals, then the evaluation of "x1 + y1" in the above example will work as expected:

```python
Python
>>>eval("x1 + y1", {"x1": x1, "y1": y1})
300
```

Since you add y1 to your custom globals dictionary, the evaluation of "x1 + y1" is successful and you get the expected return value of 300.

The following examples show that even though you supply an empty dictionary to globals, the call to eval() will still have access to Python's built-in names:

```python
Python
>>>eval("sum([13, 12, 25])", {})
50
>>>eval("min([19, 55, 12])", {})
12
>>>eval("pow(10, 2)", {})
100
```

In the above code, you supply an empty dictionary ({}) to globals. Since that dictionary doesn't contain a key called "__builtins__", Python automatically inserts one with a reference to the names in builtins. This way, eval() has full access to all of Python's built-in names when it parses expression.

If eval() is call without passing a custom dictionary to globals, then the argument will default to the dictionary returned by globals() in the environment where eval() is called:

```python
Python
>>>x1 = 10
>>> y1 = 20
>>>eval("x1 + y1")  # Access global variables
30
```

Wheneval() is called without supplying a globals argument, the function evaluates expression using the dictionary returned by globals() as its global namespace. So, in the above example, you can freely access x1 and y1 because they're global variables included in your current global scope.

The Third Argument: locals

Python's eval() takes a third argument called locals. This is another optional argument that holds a dictionary. In this case, the dictionary contains the variables that eval() uses as local names when evaluating expression.

Local names are those names (variables, functions, classes, and so on) that you define inside a given function. Local names are visible only from inside the enclosing function. You define these kinds of names when you're writing a function.

Since eval() is already written, you can't add local names to its code . However, you can pass a dictionary to locals, and eval() will treat those names as local names:
```python
>>>eval("x1 + 100", { }, {"x1": 100})
200
```

Evaluating Expressions With Python's eval()

Python's eval() can be used to evaluate Boolean, math, and general-purpose Python expressions.

Boolean Expressions

Boolean expressions are Python expressions that return a True or False when the interpreter evaluates them. They're commonly used in if statements to check if some condition is true or false.

```python
Python
>>>x1 = 100

>>> y1 = 100

>>>eval("x1 != y1")
False
```

```
>>>eval("x1< 300 and y1> 100")
False

>>>eval("x1 is y1")
True

>>>eval("x1 in {155, 100, 180, 190}")
True
```

eval()can be used with Boolean expressions that use any of the following Python operators:

- Value comparison operators: <, >, <=, >=, ==, !=
- Logical (Boolean) operators: and, or, not
- Membership test operators: in, not in
- Identity operators: is, is not

Math Expressions
One common use case of Python's eval() is to evaluate math expressions from a string-based input.
The following examples show how you can use eval() along with math to perform math operations:

```
>>> import math
>>> # Area of a circle
>>>eval("math.pi * pow(2, 2)")
12.56
```

General-Purpose Expressions

eval() can be used with more complex Python expressions that incorporate function calls, object creation, attribute access, comprehensions, and so on.

For example, you can call a built-in function or one that you've imported with a standard or third-party module:

```
>>> import subprocess
>>>eval("subprocess.getoutput('echo Hi, Friends')")
'Hi, Friends'
```

Operators
Operators are special symbols in Python that carry out arithmetic or logical computation. The value that the operator operates on is called the operand.

For example:
```
>>>12+93
115
```

Here, + is the operator that performs addition. 12 and 93 are the operands and 115 is the output of the operation.

Arithmetic operators:

Arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication, etc.

Operator	Meaning	Example
+	Add two operands or unary plus	x + y+ 2
-	Subtract right operand from the left or unary minus	x - y- 2
*	Multiply two operands	x * y
/	Divide left operand by the right one (always results into float)	x / y
%	Modulus - remainder of the division of left operand by the right	x % y (remainder of x/y)
//	Floor division - division that results into whole number adjusted to the left in the number line	x // y
**	Exponent - left operand raised to the power of right	x**y (x to the power y)

Example 1: Arithmetic operators in Python

```
x = 15
y = 4

print('x + y =',x+y)
# Output: x + y = 19

print('x - y =',x-y)
# Output: x - y = 11

print('x * y =',x*y)
# Output: x * y = 60

print('x / y =',x/y)
# Output: x / y = 3.75

print('x // y =',x//y)
# Output: x // y = 3
```

```
print('x ** y =',x**y)
# Output: x ** y = 50625
```

Comparison operators:

Comparison operators are used to compare values. The comparison operators are also known as relational operator. It returns either True or False according to the condition.

Operator	Meaning	Syntax
>	Greater than:- True if the left operand is greater than the right	a>b
<	Less than:- True if the left operand is less than the right	a < b
==	Equal to:- True if both operands are equal	a == b
!=	Not equal to:- True if operands are not equal	a != b
>=	Greater than or equal to:- True if the left operand is greater than or equal to the right	a >= b
<=	Less than or equal to:- True if the left operand is less than or equal to the right	a <= b

```
a = 100
b = 112

print('a> b is',a>b)
Output: a > b is False

print('a<bis',a<b)
Output: a < b is True

print('a == b is',a==b)
Output: a == b is False

print('a != b is',a!=b)
Output: a != b is True

print('a >= b is',a>=b)
Output: a>= b is False

print('a <= b is',a<=b)
Output: a<= b is True
```

Logical operators

Logical operators are and, or, not operators.

Operator   Meaning

and	if both the operands are true then condition become true.
or	if either of the operands is true then condition become true.
not	Used to reverse the logical state of its operand.

a = True
b = False

print('a and bis',a and b)

print('a or bis',x or y)

print('not ais',nota)
Output

a and b is False
a or b is True
nota is False

Bitwise operators

Bitwise operator works on bits and performs bit by bit operation.

&	Binary AND Operator	This Operator is used to copy a bit to the result if it exists in both operands.
|	Binary OR Operator	It copies a bit if it exists in any operand.
^	Binary XOR Operator	It copies a bit if it is set in one operand but not both.
~	Binary ones Complement Operator	It is unary and will flip the bits, so 1 will become 0, and 0 will become 1.
<<	Binary Left Shift Operator	It shifts the value of the left operand to left by the number of bits given by the right operand.
>>	Binary Right Shift Operator	It shifts the value of the left operand to right by the number of bits given by the right operand.

Assignment operators

Assignment operators are used to assigning values to variables.

=	These operators are used to assign the values for the right side operands to the left side operand.	a = 25
+=	These operators are used to add the right operand to the left operand and the result is aassign to the left operand.	a += 10 a = a + 10
-=	The right operand gets subtracted from the left operand and the result is assigned to the left operand.	a -= 10 a = a - 10
*=	The right operand gets multiplied with the left operand and the result is assigned to the left operand.	a *= 10 a = a * 10
/=	The left operand is divided by the right operand and the result is assigned to the left operand.	a /= 10 a = a / 10
%=	It performs the modulus operation and the result is assigned to the left operand.	a %= 10 a = a % 10

a = 25 is a simple assignment operator that assigns the value 25 on the right to the variable a on the left.

```
a=25
a += 10
print(" a is : ",a)
#output:a is : 35

a -= 10
print(" a is : ",a)
#output:a is : 15

a *= 2
print(" a is : ",a)
#output:a is : 50

a /= 5
print(" a is : ",a)
#Output:a is : 5

a %= 4
print(" a is : ",a)
#output:a is : 1
```

Special operators

Python language offers some special types of operators like the identity operator or the membership operator.

Identity operators

Identity operators are used to compare the memory locations of two objects.

is	True - If the variable on either side of the operator point to the same object False - If the variable on either side of the operator does not point to the same object
is not	False - if the variable on either side of the operator point to the same object. True - if the variable on either side of the operator point to the different object.

Example:

```
p=50
q=50

print('p =',p,':',id(p))
#output:-p=50 : 80983211
print('q =',p,':',id(q))
#output:-q=50 : 80983211

if(p is q) :
print('p and q have same identity')
else :
print('p and q do not have same identity')

#output:-p and q have same identity

if(id(p) = = id(q)) :
print('p and q have same identity')
else :
print('p and q do not have same identity')

#output:-p and q have same identity

if(p is not q) :
print('p and q do not have same identity')
else :
print('p and q have same identity')

#output:p and q do not have same identity
```

Membership operators

There are two membership operators in Python (in and not in). They are used to test for membership in a sequence(string, list, tuple and dictionary).

in	Evaluates to true if it finds a variable in the sequence and return false if it does not find a variable in the sequence
not in	Evaluates to false if it finds a variable in the sequence and return true if it does not find a variable in the sequence

```
p=60
q=30
list1=[10,30,50,66]
example 1:
if(p in list1):
print('p is in the list')
else :
print('p is not in the list')
#output:-p is not in the list
example 2:
if(p not in list1):
print('p is not in the list')
else :
print('p is in the list')
#output:-p is not in the list

example 3:
if(q in list1):
print('q is in the list')
else:
print('q is not in the list')
#output:-q is in the list
```

3 Built-in function type.

type() function in python returns the data type of the object which is passes as an argument in this function. It is basically used for debugging process.

syntax
```
type(object)
type(name, bases, dict)
```
a. object – for which type needs to be returned
b. name – name of the class
c. bases – a tuple of classes
d. dict – dictionary that holds namespace for the class

```
>>>print(type([1,2,3]))
#output :- <type 'list'>

>>>print(type((1,2,3)))
#output :- <type 'tuple'>

>>>print(type({1:'a',2:'b',3:'c'}))
#output :- <type 'dict'>

>>>print(type("Good"))
#output :- <type 'str'>

>>>print(type(1))
#output :- <type 'int'>

>>>print(type(3.14))
#output :- <type 'float'>

>>>print(type(True))
#output :- <type 'bool'>
```

4   Operator precedence.

The order of the operation means the operator precedence to evaluate an expression. The operator will help to evaluate the expression. Following table shows all operatorsthat has highest precedence to lowest.

Operator	Description
( )	Parentheses
**	Exponent
+a, -a, ~a	Unary plus, Unary minus, Bitwise NOT
*, /, //, %	Multiplication, Division, Floor division, Modulus
+, -	Addition, Subtraction
<<, >>	Bitwise shift operator
&	Bitwise and
^	Bitwise XOR
|	Bitwise or
==, !=, >, >=, <, <=	Comparisons operator
Not	Logical NOT
And	Logical AND
Or	Logical OR

Operators with the highest precedence appear at the top of the table

Example:
```
>>> 5*2+10
20
```

In this example the first precedence is given to the multiplication then addition operator.

```
>>>5*(2+10)
60
```

In this example the first precedence is given to the parenthesis then multiplication operator.

```
>>>10/5+10
12
```

In this example the first precedence is given to the division then addition operator.

1.6 SUMMARY

a. We learnt Reasons for Python as the learner's first programming language.

b. We have studied IDLE interpreter (shell) and its documentation.

c. We learnt syntax of expression evaluation: similarities and differences compared to a Calculator

d. We used different types of expressions and operators.

e. We also used Built-in function type & operator precedence in python programs.

1.7 REFERENCE FOR FURTHER READING

a. Charles Dierbach, Introduction to Computer Science using Python, Wiley, 2013

b. Paul Gries , Jennifer Campbell, Jason Montojo, Practical Programming: An Introduction to Computer Science Using Python 3, Pragmatic Bookshelf, 2/E 2014

1.8 UNIT END EXERCISES

1. What are the features of the python programming language?
2. Explain Expression evaluation in python.
3. Explain various operators used in python.
4. Write a python program using Math Expressions.
5. Explain the use of Built-in type function with an example.
6. Explain operator precedence with an example.

❖❖❖❖

2

PROGRAMMING WITH PYTHON- II

Unit Structure

2.1   OBJECTIVE

1. Understand the simple statements
2. Understand the compound statements
3. Use Python to delete the statement
4. Understand print and input functions in python.
5. Understand the built-inhelpFunctions

2.2 ENUMERATION OF SIMPLE AND COMPOUND STATEMENTS.

A Simple statement in python:

As the name suggests, simple statements are comprised in one single line.These types of statements will never affect the flow of a program.

There are many simple statements like assert, break, continue, pass, del, import, return, etc.
The syntax for simple statement is:
simple_stmt ::= expression_stmt
        | assert_stmt
        | assignment_stmt
        | augmented_assignment_stmt
        | annotated_assignment_stmt
        | pass_stmt
        | del_stmt
        | return_stmt
        | yield_stmt
        | raise_stmt
        | break_stmt
        | continue_stmt
        | import_stmt
        | future_stmt
        | global_stmt
        | nonlocal_stmt

Compound statements in python:

A compound statement is defined as a group of statements that affect or control the execution of other statements in some way. Python programming language provides different ways to construct or write compound statements. These are mainly defined as:

If Statement:
This statement is used for conditional execution.
The syntax of the statement is as follows:-
if expression: suite
elif expression: suite
else: suite

while Statement:
While statement is used to execute a block of statements repeatedly until a given condition is fulfilled. Syntax of the statement is as follows
while expression:
statement(s)

for statement:
for statement is used to traverse a list or string or array in a sequential manner for example

```
print("List Iteration")
li1 = ["python", "for", "programmers"]
fori in li1:
print(i)
```

Try statement:

The try statement specifies exception handlers and/or cleanup code for a group of statements.

With statement:

The with statement is used to wrap the execution of a block with methods defined by a context manager where a context manager is an object that defines the runtime context to be established.

2.3 THE EXPRESSION STATEMENT, ASSIGNMENT STATEMENT, THE ASSERT STATEMENT,

2.3.1 The expression statement.

These types of statements are formed using operators, variables or values which create one mathematical expression.
Example:
>>>i=5
>>> v=i-2
>>>v
3
>>> z=i**2
>>>z
25

2.3.2 Assignment Statement:

In this type of statement, we assign value to a variable. This statement has one equal (=) sign. We cannot set keywords as a variable name.

Example:

>>>i=5
>>> a=10
>>> #keyword cannot be used in assignment
>>>for=5
SyntaxError: invalid syntax

2.3.2   The assert statement
Assert statements:
Assert statements are a convenient way to insert debugging assertions into a program:

assert_stmt ::= "assert" expression ["," expression]
The simple form, assert expression, is equivalent to

if __debug__:
if not expression: raise AssertionError
The extended form, assert expression1, expression2, is equivalent to

19

if __debug__:

if not expression1: raise AssertionError(expression2)

These equivalences assume that __debug__ and AssertionError refers to the built-in variables with those names. In the current implementation, the built-in variable __debug__ is True under normal circumstances, False when optimization is requested (command-line option -O). The current code generator emits no code for an assert statement when optimization is requested at compile time. Note that it is unnecessary to include the source code for the expression that failed in the error message; it will be displayed as part of the stack trace.

Assignments to __debug__ are illegal. The value for the built-in variable is determined when the interpreter starts.

```
>>> def avg(marks):
        assert len(marks) != 0,"List is empty."
        return sum(marks)/len(marks)

>>> mark2 = [55,88,78,90,79]
>>> print("Average of mark2:",avg(mark2))
Average of mark2: 78.0
>>> mark1 = []
>>> print("Average of mark1:",avg(mark1))
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    print("Average of mark1:",avg(mark1))
  File "<pyshell#14>", line 2, in avg
    assert len(marks) != 0,"List is empty."
AssertionError: List is empty.
```

2.4 BOOLEAN EXPRESSION (VALUES TRUE OR FALSE).

A boolean expression is an expression that yields just the two outcomes: true or false. When we work with multiple boolean expressions or perform some actions on them, we make use of the boolean operators. Since the boolean expression reveals true or false, the operations on these expressions also result in either "true" or "false". Consequently, there are three types of boolean operators:
1.   The AND operator (&& or "and")
2.   The OR operator (|| or "or")
3.   The NOT operator (not)

1.   AND Boolean Operator in Python
        The AND boolean operator is similar to the bitwise AND operator where the operator analyzes the expressions written on both sides and returns the output.

True and True = True
True and False = False
False and True = False
False and False = False

In python, you can directly use the word "and " instead of "&&" to denote the "and " boolean operator while for other languages, you need to put "&&" instead.

```
a = 30
b = 45
if(a > 30 and b == 45):
print("True")
else:
print("False")
```

Output:-False

2. OR Boolean Operator in Python
The OR operator is similar to the OR bitwise operator. In the bitwise OR, we were focusing on either of the bit being 1. Here, we take into account if either of the expression is true or not. If at least one expression is true, consequently, the result is true.
True or True = True
True or False = True
False or True = True
False or False = False

```
a = 25
b = 30
if(a > 30 or b < 45):
print("True")
else:
print("False")
```

Output:-True

3. NOT Boolean Operator in Python
The NOT operator reverses the result of the boolean expression that follows the operator. It is important to note that the NOT operator will only reverse the final result of the expression that immediately follows. Moreover, the NOT operator is denoted by the keyword "not".
```
a = 2
b = 2
if(not(a == b)):
print("If Executed")
else:
print("Else Executed")
```

Output:-Else Executed

2.5 DYNAMIC BINDING OF NAMES TO VALUES, (TYPE IS ASSOCIATED WITH DATA AND NOT WITH NAMES); AUTOMATIC AND IMPLICIT DECLARATION OF VARIABLE NAMES WITH THE ASSIGNMENT STATEMENT; ASSIGNING THE VALUENONE TO A NAME.

2.5.1 Python is a dynamically typed language. It doesn't know about the type of the variable until the code is executed. So variable declaration is of no use. What it does is store that value at some memory location and then bind that variable name to that memory container. And makes the contents of the container accessible through that variable name. So the data type does not matter. As it will get to know the type of the value at run-time.

Example:
#This will store 99 in the memory and bind the name x to it. After it executes, the type of x will be int.
x = 99
print(type(x))
# This will store 'India' at some location in the memory and binds name x to it. After it runs type of x will be str.
x = 'India'
print(type(x))

Output:
<class 'int'>
<class 'str'>

Variables
        As the name implies, a variable is something that can change. A variable is a way of referring to a memory location used by a computer program. A variable is a symbolic name for this physical location. This memory location contains values, like numbers, strings etc.

        A variable can be seen as a container to store values. While the program is running, variables are accessed and sometimes changed, i.e. a new value will be assigned to the variable.

        One of the main differences between Python and other programming languages is the way it deals with types. In strongly-typed languages, every variable must have a unique data type. E.g. if a variable is of type integer, solely integers can be saved in the variable. In Java or C, every variable has to be declared before it can be used.

        Declaration of variables is not required in Python. If there is a need for a variable, you type a name and start using it as a variable.

Another aspect of Python: Not only the value of a variable may change during program execution but the type as well. You can assign a float value to a variable, use it as a float for a while and then assign a string to the variable.

Example:
x = 122
>>>print x
122
>>> x ="Mumbai"
>>>print x
Mumbai

```
x = 82              # data type is integer
x = 82 + 0.11       # data type is changed to float
x = "Mumbai"        # and now it will be a string
```

Python automatically takes care of the physical representation for the different data types, i.e. an integer value will be stored in a different memory location than a float or a string.

Dynamic Typing vs Static Typing:

Dynamic Typing	Static Typing
Variable type checking will happen at runtime.	Variable type checking will happen at compile time.
As the typing checking happens in runtime the dynamically typed code can compile even it has some errors which might lead logical issues.	The statically typed code won't compile if it has any error. In order to run the code, we must fix the error.
Dynamically typed languages don't expect the variable declaration before using them. The variable declaration will happen automatically when we assign values to it.	Statically typed languages expect the variable declaration before assigning values to it.
Python, Perl, etc.	C, Java, etc.

Assigning the valueNone to a name:

Python None Keyword:

Python has a set of keywords that are reserved words that cannot be used as variable names, function names, or any other identifiers.

Keyword None: Represents a null value

Assign the value None to a variable:

```
a = None
print(a)
```

Definition and Usage
The None keyword is used to define a null value or no value at all.

None is not the same as 0 (Zero), False, or an empty string. None is a data type of its own and only None can be None.

2.6 THE DEL (DELETE) STATEMENT.

The del statement :

del 〈 object ,...〉

Each object is any kind of Python object. Usually, these are variables, but they can be functions, modules, classes.

The del statement works by unbinding the name, removing it from the set of names known to the Python interpreter. If this variable was the last remaining reference to an object, the object will be removed from memory. If, on the other hand, other variables still refer to this object, the object won't be deleted.

The del statement is typically used only in rare, specialized cases. Ordinary namespace management and garbage collection are generally sufficient for most purposes.

The del statement can be used to delete an item at a given index. Also, it can be used to remove slices from a list.

```
List1 = [1, 2, 3, 4, 5, 6, 7, 8, 9]
# deleting the third item
del list1[2]
print(list1)
# Output: [1, 2, 4, 5, 6, 7, 8, 9]
# deleting items from 2nd to 4th
del list1[1:4]
print(list1)
# Output: [1, 6, 7, 8, 9]
# deleting all elements
del list1[:]
print(list1)
# Output: []
```

2.7 INPUT/OUTPUT WITH PRINT AND INPUT FUNCTIONS.

Python provides numerous built-in functions that are readily available. Some of the functions like input() and print() are used for input and output operations respectively.

Python Input:
In Python, input() function is used totake input from the user.
The syntax for input() is:
input([prompt])

>>>na = input('Enter your name: ')
Enter your name :Amit
>>>na
'Amit'

>>>num = int(input('Enter a number: '))
Enter a number: 10
>>>num
10

>>>num = float(input('Enter a number: '))
Enter a number: 10
>>>num
10.0

print() function is used to output data to the output device.

print('Python programming')
Output:-Python programming

a = 56
print('a is', a)

Output:-a is 56

The syntax of the print() function :

print(*objects, sep=' ', end='\n', file=sys.stdout)

objects is the value(s) to be printed.

The sep separator is used between the values.

Once all values are printed, end is printed. It defaults into a new line.

The file is the object where the values are printed and its default value is sys.stdout.

print(11, 21, 31, 41)
print(11, 21, 31, 41, sep='$')
print(11, 21, 31, 41, sep='@', end='#')

Output
11 21 31 41
11$21$31$41
11@21@31@41#

2.8 A STATEMENT LIST (SEMICOLON SEPARATED LIST OF SIMPLE STATEMENTS ON A SINGLE LINE) AS A SINGLE INTERPRETER COMMAND.

A statement list (semicolon-separated list of simple statements on a single line) as a single interpreter command
Each statement is written on a separate physical line in the editor. However, statements in a block can be written in one line if they are separated bysemicolon.

Following is the code of three statements written in separate lines

a=10
b=20
c=a*b
print (c)

These statements can very well be written in one line by putting a semicolon in between.

a=10; b=20; c=a*b; print (c)

A new block of increased indent starts after : symbol as in case of if, else, while, for, try statements. However, using syntax, statements in a block can be written in one line by adding a semicolon.

Following is the example of a block of statements in a for loop

fori in range(4):
print ("Mumbai")
print ("i=",i)

This block can also be written in single line using :

fori in range(4): print ("Mumbai"); print ("i=",i)

2.9 THE IMPORT STATEMENT FOR ALREADY-DEFINED FUNCTIONS AND CONSTANTS.

The import statement for already-defined functions and constants
Any text file with the .py extension containing Python code is basically a module. Different Python objects such as functions, classes, variables, constants, etc., defined in one module can be made available to

an interpreter session or another Python script by using the import statement. Functions defined in built-in modules need to be imported before use. On similar lines, a custom module may have one or more user-defined Python objects in it. These objects can be imported into the interpreter session or another script.

If the programming algorithm requires defining a lot of functions and classes, they are logically organized in modules. One module stores classes, functions, and other resources of similar relevance. Such a modular structure of the code makes it easy to understand, use and maintain.

Creating a Module
The definition of sum() function. It is saved as cal.py.
cal.py

```
def sum(x, y):
return x + y
```

Importing a Module

now import this module and execute the sum() function in Python.

Example:

```
>>> import cal
>>>cal.sum(55, 44)
99
```

The import statement for constants
Constant:
A constant value is similar to a variable, with the exception that it cannot be changed once it is set. Constants have a variety of uses, from setting static values to writing more semantic code.
create two files, constants1.py and main.py to demonstrate.

```
# constants1.py
No1 = 20
```

```
# main.py
import constants1
No2 = 170
total = No2 + constants1.No1
print("Total is {total}")
```

Output:
Total is 190

2.10   THE AUGMENTED ASSIGNMENT STATEMENT.

The augmented assignment statement :
unlike normal assignment operators, augmented assignment operators are used to replace those statements where binary operator takes two operands says var11 and var12 and then assigns a final result back to one of operands i.e. var11 or var12.

For example: statement var11 = var11 + 5 is same as writing var11 += 5 in python and this is known as augmented assignment operator. Such type of operators are known as augmented because their functionality is extended or augmented to two operations at the same time.

List of Augmented Assignment Operators in Python

Addition & Assignment (+=): x+=y is equivalent to x=x+y

```
# Addition
a = 10
b = 5
a += b
print('Addition = %d' %(a))
```

OUTPUT :-Addition = 15

Subtraction & Assignment (-=): x-=y is equivalent to x=x-y

```
# Subtraction
a = 10
b = 5
a -= b
print('Subtraction = %d' %(a))
```

OUTPUT :-Subtraction = 5

Multiplication & Assignment (*=): x*=y is equivalent to x=x*y

```
# Multiplication
a = 10
b = 5
a *= b
print('Multiplication = %d' %(a))
```

OUTPUT :-Multiplication = 50

Division & Assignment (/=): x/=y is equivalent to x=x/y

```
# Division
a = 23
```

```
b = 3
a /= b
print('Division = %f' %(a))
```

OUTPUT :-Division = 7.666667

Remainder(or Modulo) & Assignment (%=): x%=y is equivalent tox=x%y

```
# Remainder or Modulo
a = 12
b = 5
a %= b
print('Remainder or Modulo = %d' %(a))
```

OUTPUT :-Remainder or Modulo = 2

2.11   THE BUILT-INHELP() FUNCTION.

The python help function is used to display the documentation of modules, functions, classes, keywords etc.

The help function has the following syntax:
        help(print)

It gives the following output:

Help on built-in function print in module built-ins:
print(...)
print(value, ..., sep=' ', end='\n', file=sys.stdout)
Prints the values to a stream, or to sys.stdout by default.
Optional keyword arguments:

        file: a file-like object (stream); defaults to the current sys.stdout.
        sep: string inserted between values, default a space.
        end: string appended after the last value, default a newline.

2.12   SUMMARY

a. In this chapter, we learn the use of Enumeration of simple and compound statements.

b. Study of expression statement and the assert statement.

c. We have studied Boolean expression (values true or false). The assignment statement, dynamic binding of names to values, (type is associated with data and not with names); automatic and implicit declaration of variable names with the assignment statement; assigning the value None to a name.

d. Understood the concept of the del (delete) statement in python.

2.13 REFERENCE FOR FURTHER READING

a. Charles Dierbach, Introduction to Computer Science using Python, Wiley, 2013

b. Paul Gries, Jennifer Campbell, Jason Montojo, Practical Programming: An Introduction to ComputerScience Using Python 3, Pragmatic Bookshelf, 2/E 2014

2.14 UNIT END EXERCISES

1 Explain Simple statements.
2 What is assigning the value None to a name?
3 Explain the augmented assignment statement.
4 Write a Python program for the import statement for already-defined functions.
5 Explain Compound statements.
6 Write the difference between Dynamic Typing and Static Typing
7 Explain Assert statement.

❖❖❖❖

3

PROGRAMMING WITH PYTHON- III

Unit Structure

3.1   Objective

3.2   Interactive and script modes of IDLE, running a script, restarting the shell.

3.3   The compound statement def to define functions; the role of indentation for delimiting the body of a compound statement; calling a previously defined function. Compound data types str, tuple and list (enclosed in quotes, parentheses and brackets, respectively). Indexing individual elements within these types. Strings and tuples are immutable, lists are mutable.

3.4   Built-in functions min, max, sum.

3.5   Interactive solution of model problems, (e.g., finding the square root of a number or zero of a function), by repeatedly executing the body of a loop (where the body is a statement list).

3.6   Summary

3.7   Reference for further reading

3.8   Unit End Exercises

3.1   OBJECTIVE

1. Understand the IDLE
2. Learn use of user define function and its application
3. Understand the List, Tuple
4. Understand the mutable

3.2   INTERACTIVE AND SCRIPT MODES OF IDLE, RUNNING A SCRIPT, RESTARTING THE SHELL.

• **Interactive Mode**

Python provides Interactive Shell to execute code immediately and produce instant output. To get into this shell, we have to write python code in the command prompt and start working on it.

- **Script Mode**

Using Script Mode, we can write our Python code in a separate file of any editor.

i) Click on Start button -> All Programs -> Python ->IDLE(Python GUI)

ii) Python Shell will be opened. Now click on File -> New Window.

- A new Editor will be opened. Write our Python code here.

To execute a file in IDLE, simply press the F5 key on your keyboard. You can also select Run → Run Module from the menu bar. Either option will restart the Python interpreter and then run the code that you've written with a fresh interpreter.

How to Use the Python IDLE Shell
The shell is the default mode of operation for Python IDLE. When you click on the icon to open the program, the shell is the first thing that you see:

You can restart the shell from this menu. If you select that option, then you'll clear the state of the shell. It will act as though you've started a fresh instance of Python IDLE. The shell will forget about everything from its previous state.

3.3 THE COMPOUND STATEMENT DEF TO DEFINE FUNCTIONS; THE ROLE OF INDENTATION FOR DELIMITING THE BODY OF A COMPOUND STATEMENT; CALLING A PREVIOUSLY DEFINED FUNCTION.

What is a function in Python?

In Python, a function is a group of related statements that performs a specific task.

Functions help break our program into smaller and modular chunks. As our program grows larger and larger, functions make it more organized and manageable.

Furthermore, it avoids repetition and makes the code reusable.

Syntax of Function

deffunction_name(parameters):
    """docstring"""
    statement(s)

Above shown is a function definition that consists of the following components.

Keyword def that marks the start of the function header.
A function_name to uniquely identify the function. Function naming follows the same rules of writing identifiers in Python.

Parameters (arguments) through which we pass values to a function.

A colon (:) to mark the end of the function header.

Optional documentation string (docstring) to describe what the function does.

One or more valid python statements that make up the function body. Statements must have the same indentation level.

An optional return statement to return a value from the function.

Example of a function
defdisp(name):
print("Hi, " + name + ". Good morning!")

How to call a function?

Once we defined a function, we can call it from another function, program. To call a function type the function name with appropriate parameters.

>>>disp('Raj')
Hi, Raj. Good morning!

Although optional, documentation is a good programming practice. In the above example, we have a docstring immediately below the function header. We can generally use triple quotes so that docstring can extend up to multiple lines. This string is available to us as the __doc__ attribute of the function.

For example:

>>>print(disp.__doc__)

  This function greets tothe person passed in as a parameter

Role of indentation

Indentation, one of the important features of python, refers to the spaces at the beginning of a code line.Python uses indentation to indicate a block of code. It conveys a better structure of a program to the readers. It explains relationship between control flow constructs such as conditions or loops, and code contained inside and outside of it.
Indentation makes our program :-
  – Easy to read
  – Easy to understand
  – Easy to modify
  – Easy to maintain
  – Easy for code debugging

The return statement
    The return statement is used to exit a function and go back to the place from where it was called.

Syntax of return
return [expression_list]

    This statement can contain an expression that gets evaluated and the value is returned. If there is no expression in the statement or the return statement itself is not present inside a function, then the function will return the None object.

    For example:

    >>>print(disp("Deep"))
    Hello, deep. Good morning!
    None

    Here, None is the returned value since disp() directly prints the name and no return statement is used.

    Example of return

    defabsolute_value(num):
        """This function returns the
    value of the entered number"""
    if number >= 0:

34

return number
else:
return -number

print(absolute_value(92))

print(absolute_value(-64))
Output
92
64

Types of Functions
Basically, we can divide functions into the following two types:

1) Built-in functions - Functions that are built into Python.
2) User-defined functions - Functions defined by the users themselves.

The Python Square Root Function:

Python's math module, in the standard library, this library can help you work on math-related problems in code. It contains useful functions, such as remainder() and factorial(). It also includes the Python square root function, sqrt().

>>> import math #importing math module
You can now use math.sqrt() to calculate square roots.

>>>math.sqrt(49)
7.0

>>>math.sqrt(70.5)
8.396427811873332

4   **Compound data types str, tuple and list (enclosed in quotes, parentheses and brackets, respectively). Indexing individual elements within these types. Strings and tuples are immutable, lists are mutable**

Compound data types in Python :

List

Dictionaries

Tuples

List:

A list is an array of objects. An object can be an integer,astring,float or other things. List is a container which holds the items or element. The elements in the list not of the same type. The element are separated by the comma in a square bracket.

The list are mutable data type. Mutable means you can do changes in the list elements after creating the list.

PYTHON LIST PROPERTIES

- Lists are ordered.
- Lists can contain any arbitrary objects.
- List elements can be accessed by index.
- Lists can be nested to arbitrary depth.
- Lists are mutable.
- Lists are dynamic.

Initializing Values of List:

>>> List1 = ["India", "Maharashtra", 81]
Empty List:

>>> List2=[]

List creation
>>>list=[13,66,88,45,76,77]

Indexing individual elements within list
The expression in brackets is called an **index.** The index indicates which character in the sequence you want to access. Index starts with zero.

Accessing first element of the list:-

>>>list[0]

output :- 13

>>>list[3]

Output :- 45

Negative indexing returns last element of the list.
>>>list[-1]
output :- 77

>>>list[-3]
output :- 45

Mutable Concept:
The list are the mutable data type. Mutable means you can do changes in the list elements after creating the list.

>>>print("value of second index position ",list[2])
88
>>list[2]=999
>>>print("value of second index position ",list[2])
999

Initialize two Lists in a single line:

>>> L1, L2=['mango','apple','banana'],['Raj','Deep']

>>L2
['Raj','Deep']

Delete element from list:
Del statement is use to delete the element from list.

>>>list=[10, 20, 30, 40, 50]

>>>del list[2]

>>print(list)
[10,20,40,50]

Built in function:

Min Function:
The min() function is used to find the minimum value from the list.

>>>list=[100, 20, 320, 140, 56]
>>min(list)
20

Max Function:
The max() function is used to find the maximum value from the list.

>>>list=[100, 20, 320, 140, 56]
>>max(list)
320

Sum Function:
Python provides an inbuilt function sum() which sums up the numbers in the list.
Syntax:
sum(iterable, start)

iterable :iterable can be anything list , tuples or dictionaries ,but most importantly it should be numbers.

start : this start is added to the sum of numbers in the iterable.

If start is not given in the syntax , it is assumed to be 0.

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

>>>print(sum(numbers))

List Operators:
There are two list operators '+'and '*' used in python. Operator '+' is used for concatenation and operator '*' is used for repetition.

Example:
>>>x1=[11,33,55]
>>>x2=[67,87,92]
>>>x3=x1+x2
>>>x3
[11,33,55,67,87,92]

>>> x=[11,33,55]
>>>x*2
[11,33,55,11,33,55]

In Operator:
Operator 'in' is used to checkwhether the given element is present in the list or not. If the given elementis present in the list it returns true.

Example:
>>>x=[11,33,55,66,78,98,211]
>>> 66 in x
True
>>>22 in x
False

Tuple:

Tuples are the immutable sequence data type. Immutable means once we create data types, the contents of it cannot be changed. Tuples are used to store the heterogeneous data.

Creating Tuple

>>> tuple1=('a','b','c','d')

>>>print(tuple1)
('a','b','c','d')

Accessing the elements in a tuple

>>> tuple1=('Deep','Raj','Sam','Geet')

>>>print("Name is: ",tuple1[0])
Name is : Deep

>>>print("Name is: ",tuple1[2])
Name is : Sam

Min method:

The min() function is used to find the minimum value in the tuple.
>>>tuple2=(12,34,5,67,55)
>>>print("Minimum value",min(tuple2))
5

Max method:

The max() function is used to find maximum value in the tuple.
>>>tuple2=(12,34,5,67,55)
>>>print("Maximum values",max(tuple2))
67

Len Method:

The len() function is used to find the total number of elements in the tuple.
>>>tuple2=(12,34,65,67,55)
>>>len(tuple2)
5

Tuple Operators:
There are two operators '+'and '*' used in python. Operator '+' is used for concatenation and operator '*' is used for repetition.

Operator '+'
>>>x1=(1,3,5)
>>>x2=(67,87,92)
>>x3=x1+x2
>>>x3
(1,3,5,67,87,92)

Operator '*'
>>> x=(1,3,5)
>>>x*2
(1,3,5,1,3,5)

In Operator:
Operator 'in' is used to checkwhether the given element is present in the tuple or not. If the given element is present, it returns true.

>>>x=(11,33,55,66,78,98,211)
>>> 66 in x
True
>>>22 in x
False

Dictionary:

Dictionaries are the mutable data types. Each key available in the dictionary is separated from its values by using the colon(:). The items of the dictionary are separated by using a comma(,) and the entire thing is enclosed in curly braces.

Dictionary creation
>>>mydict = {"name":"Deep", "surname":"patil", "age":29}

Accessing the values in a Dictionary:
>>print("Name is",mydict(["name"])
Name is Deep

>>print("Age is",mydict(["age"])
Age is 29

Deleting Dictionary Elements:
The del() statement is used to delete the dictionary element.
>>>del(mydict(["name"]))
>>>mydict
{"surname":"patil", "age":29}

Built-in Dictionary Methods
There are several built-in methods that can be invoked on dictionaries.

- d.clear() :- empties dictionary d with all key-value pairs:
  >>> d1 = {'a1': 10, 'a2': 20, 'a3': 30}
  >>> d1
  {'a1': 10, 'a2': 20, 'a3': 30}
  >>>d1.clear()
  >>> d1
  {}

- d.get(<key>[, <default>]) - Returns the value for a key if it exists in the dictionary.
  d1.get(<key>) searches dictionary d for <key> and returns the associated value if it is found. If <key> is not found, it returns None.

  >>> d1 = {'a1': 10, 'a2': 20, 'a3': 30}
  >>>print(d.get('a2'))
  20
  >>>print(d.get('a8'))
  None

- d.items() - Returns a list of key-value pairs in a dictionary.
  d1.items() returns a list of tuples containing the key-value pairs in d. The first item in each tuple is the key, and the second item is the key's value:

  >>> d1 = {'a1': 10, 'a2': 20, 'a3': 30}
  >>> d1
  {'a1': 10, 'a2': 20, 'a3': 30}
  >>>list(d1.items())

40

[('a1', 10), ('a2', 20), ('a3', 30)]
>>>list(d1.items())[1][1]
20

- d.keys() - Returns a list of keys in a dictionary.

>>> d1 = {'a1': 10, 'a2': 20, 'a3': 30}
>>> d1
{'a1': 10, 'a2': 20, 'a3': 30}

>>>list(d1.keys())
['a1', 'a2', 'a3']

3.4 BUILT-IN FUNCTIONS MIN, MAX, SUM. INTERACTIVE SOLUTION OF MODEL PROBLEMS, (E.G., FINDING THE SQUARE ROOT OF A NUMBER OR ZERO OF A FUNCTION), BY REPEATEDLY EXECUTING THE BODY OF A LOOP (WHERE THE BODY IS A STATEMENT LIST).

The Python Square Root Function:

Python's math module, in the standard library, can help you work on math-related problems in code. It contains many useful functions, such as remainder() and factorial(). It also includes the Python square root function, sqrt().
You'll begin by importing math:

>>> import math
You can now use math.sqrt() to calculate square roots.
The return value of sqrt() is the square root of x, as a floating point number.

>>>math.sqrt(49)
7.0

>>>math.sqrt(70.5)
8.396427811873332

3.5 SUMMARY

In this chapter we studied the use of Interactive and script modes of IDLE, running a script, restarting the shell.The compound statement def to define functions; the role of indentation for delimiting the body of a compound statement; calling a previously defined function.Compound data types str, tuple and list (enclosed in quotes, parentheses and brackets, respectively). Indexing individual elements within these types. Strings and

tuples are immutable, lists are mutable. Built-in functions min, max, sum. Interactive solution of model problems, (e.g., finding the square root of a number or zero of a function), by repeatedly executing the body of a loop (where the body is a statement list).

3.6 REFERENCE FOR FURTHER READING

1. Charles Dierbach, Introduction to Computer Science using Python, Wiley, 2013

2. Paul Gries , Jennifer Campbell, Jason Montojo, Practical Programming: An Introduction to ComputerScience Using Python 3, Pragmatic Bookshelf, 2/E 2014

3.7 UNIT END EXERCISES

1. What are two modes of working in IDLE?
2. What is list? Explain with example?
3. Explain the function of tuple with example.
4. What is difference between mutable and immutable?
5. What is difference between list and tuple?

❖❖❖❖

4

FUNCTIONS

Unit Structure

4.1.1 Definition

4.1.2 Advantages

4.1.3 Function types

4.1.4 Parameters and arguments

4.1.5 Return statement

4.1.6 Recursive function

4.1.7 Global and local variables

4.1.1 DEFINITION OF FUNCTION

A function is a block of organized, reusable code.That means a function created once and can be used multiple times.

4.1.2 ADVANTAGES OF USING FUNCTIONS

- Use of functions makes the code shorter
- Use of functions makes the code easier to understand
- It reduces duplication of code
- Code can be reused easily
- Implements information hiding
- Divides a larger problem into smaller parts
- Improves modularity of code

4.1.3 FUNCTION TYPES

Functions are divided into two categories :
- Built-in functions
- User-defined functions

Built-in functions

- A function that is already defined in a program or programming framework with a set of statements, which together performs a task is called a Built-in function. There is no need for a user to create these functions and these can be used directly in their program or application.

- The Python interpreter has a number of built-in functions.
- abs() - returns absolute value of a number
- bin() - converts integer to binary string
- dict() - Creates a Dictionary
- float() - returns floating point number from number, string
- help() - Invokes the built-in Help System
- input() - reads and returns a line of string
- int() - returns integer from a number or string
- iter() - returns an iterator
- len() - Returns Length of an Object
- list() - creates a list in Python
- max() - returns the largest item
- min() - returns the smallest value
- pow() - returns the power of a number
- print() - Prints the Given Object
- range() - return sequence of integers between start and stop
- round() - rounds a number to specified decimals
- sum() - Adds items of an Iterable
- tuple() - Returns a tuple
- type() - Returns the type of the object

User defined function

They are functions created by the user. They are defined once and called multiple times.

- Syntax:-

  deffunctionname(parameters):
    block of statements
    return [expression]
  function call
  function call

- Example:

  def display(name):
        print(name)
        return
  display()
  display()

4.1.4 PARAMETER AND ARGUMENT

A parameter is the variable listed inside the parentheses in the function definition.

An argument is a value that is sent to the function when it is called.
Types
- Actual parameters
    Here actual data is sent into a function
    e.g add(10,20)
- Formal parameters
    A variable containing a value is passed
    e.g add(x,y)

Actual parameters

Example:
Defadd(a,b):
        print(a+b)

add(20,30)
output: 50

Formal parameters
Example
defcube(**x**):
      return x*x*x

4.1.5 RETURN STATEMENT

- The Python return statement is a special statement that you can use inside a function or method to send the function's result back to the caller. A return statement consists of the return keyword followed by an optional return value. The return value of a Python function can be any Python object
- **Example:-**
- def add(a,b):
            return(a+b)
            print(("sum =",add(10,20))

output:
        sum= 30

4.1.6 GLOBAL AND LOCAL VARIABLES

Global Variables

In Python, a variable declared outside of the function or in global scope is known as a global variable. This means that a global variable can be accessed inside as well as outside of the function.

Example :

```
g = "global"

defdisplay():
print("Global variable inside the Function :", g)

display()
print("Global variable outside the Function:", g)
```

Local Variables

A variable declared inside the function's body or in the local scope is known as a local variable.

Example :

```
def display():
      l="Local"
      print(l)
display()
```

❖❖❖❖

5

CONDITIONAL STATEMENTS AND LOOPS

Unit Structure

5.2.1 Range function

5.2.2 Iterative for statement

5.2.3 Conditional statements if, if…else, if…elif…else and Nested if

5.2.4 Loops

5.2.1 RANGE FUNCTION

The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and stops before a specified number.

Syntax:

range(start,stop,step):

range() takes mainly three arguments.

- **start**: integer starting from which the sequence of integers is to be returned
- **stop:** integer before which the sequence of integers is to be returned.
  The range of integers end at stop – 1.
- **step:** integer value which determines the increment between each integer in the sequence

Some examples using the range function are given below:

Example 1:

fori in range(10):

   print(i, end =" ")

Output:

0 1 2 3 4 5 6 7 8 9

Example 2:

l = [10, 20, 30, 40]

fori in range(len(l)):

   print(l[i], end =" ")

Output:

10 20 30 40

Example 3:
fori in range(3, 6):
print(i, end ='' '')
Output:
3 4 5

Example 4:
fori in range(2, 10, 2):
print(i, end ='' '')
Output:
2 4 6 8

5.2.2 ITERATIVE FOR STATEMENT

A for loop is used for iterating over a sequence.

Example:

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  print(x)

forch in 'FYCS':
print(ch)
```

5.2.3 CONDITIONAL STATEMENTS

Conditional Statements in Python perform different computations or actions depending on whether a specific Boolean constraint evaluates to true or false. Conditional statements are handled by IF statements in Python.

If statement

Python if Statement is used for decision-making operations. It contains a body of code that runs only when the condition given in 'if' statement is true. If the condition is false, then the optional else statement runs which contains some code for the else condition.

Syntax:
```
if expression:
 Statement
else:
 Statement
```

Example:
a = 33
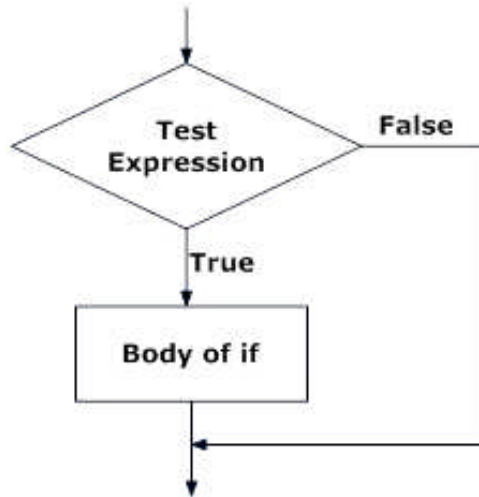b = 200
if b > a:
 print("b is greater than a")

Fig: Operation of if statement

Else
If the condition is false the block of else is executed. Else is optional
Syntax:

```
if test expression:
    Body of if
else:
    Body of else
```
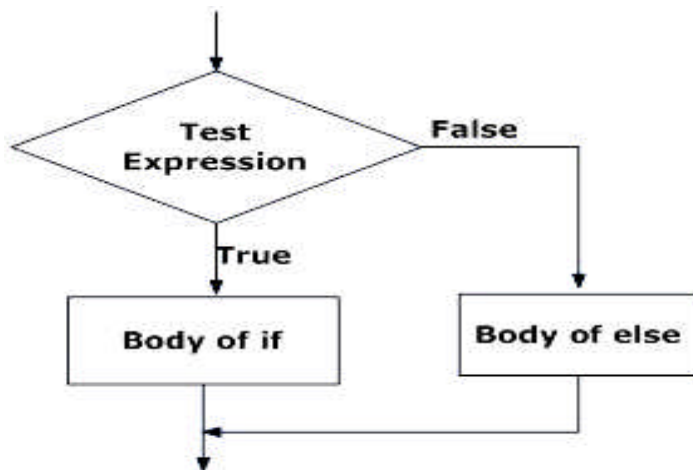

Fig: Operation of if...else statement

Elif

If condition for if is false the conditions in elif are checked if the conditions are true the blocks are executed. It allows us to check for multiple expressions.

Syntax:

```
if test expression:
    Body of if
elif test expression:
    Body of elif
elif test expression:
    Body of elif
else:
    Body of else
```

Example
```
n = int(input("Enter a number"))
f = 1
if n < 0 :
print("Factorial Does not exist")
elif n == 0 :
print("Factorial of 0 is 1")
else:
fori in range(1,n + 1):
    f = f * i
  print("factorial = ",f)
```
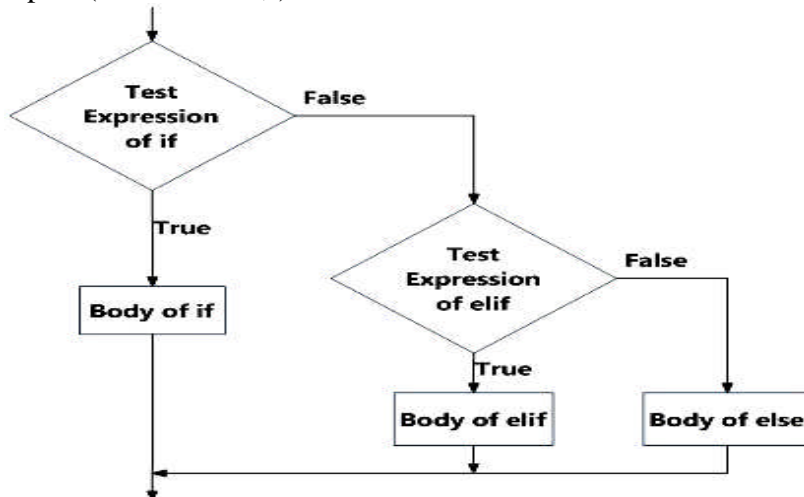

Fig: Operation of if...elif...else statement

Python Nested if statements

We can have a if statement inside another if statement. Such statements are called nested if statements.

Example:

```
x = 105
if x > 10:
  print("Above ten,")
 if x > 100:
   print("and also above 100")
 else:
   print("but not above 20.")
```

5.2.4 LOOPS

A loop statement allows us to execute a statement or group of statements multiple times
Python has two primitive loop commands:
- while loop
- for loop

The while Loop

The while loop is used to execute a set of statements as long as the condition is true.

Syntax:

```
while expression:
   Block of statements
```

Example:

```
i = 1
while i< 10:
 print(i)
 i += 1
```

While …else

While loops can also have an optional else block. The else part is executed if the condition in the while loop is evaluated as false.

```
counter = 0
while counter <5:
print("Inside loop")
counter = counter + 1
else:
print("Inside else")
```

The Continue Statement

The continue statement in Python returns the control to the beginning of the while loop. The continue statement rejects all the remaining statements in the current iteration of the loop and moves the control back to the top of the loop.

Syntax:
while expression:
   Block of statements
   If condition:
       continue

Example:
forch in 'Python' :
ifch == 'h':
continue
print("Current character : ", ch)

output :-
Currentcharacter : P
Currentcharacter :y
Currentcharacter :t
Currentcharacter :o
Currentcharacter :n

The break statement
      The break statement in Python terminates the current loop and resumes execution at the next statement.
Syntax:
while expression:
   Block of statements
   If condition:
       break

Example:
forch in 'Python' :
ifch == 'h':
break
print("Current character : ", ch)

output :-
Currentcharacter : P
Currentcharacter : y
Currentcharacter : t

❖❖❖❖

DICTIONARIES

Unit Structure

6.3.1 Dictionary

6.3.2Operations and functions of dictionary

A dictionary is an unordered collection of key value pairs. It is mutable and does not allow duplicates. Creating a dictionary is as simple as placing items inside curly braces {} separated by commas.

Key and values

An item has a key and a corresponding value that is expressed as a pair and can be referred to by using the key name.Keys are unique within a dictionary while values may not be.

Creating dictionary

Syntax:

Dictionaryname={"key":"value"}

Example:

```
studentinfo = {
 "Name": "John",
 "Class": "FYCS",
 "Rollno": 01
}
```

Accessing valuesin a dictionary

>>>print(studentinfo['Name'])

>>>print(studentinfo.get('Class'))

Adding an item to a dictionary

Adding an item to the dictionary is done by using a new index key and assigning a value to it

Syntax:

Dictionaryname["Key"]="value"

Example:

>>>studentinfo["DOB"] = "01/01/2020" **#new item added**

>>>print(studentinfo)

Updating Dictionary

Dictionaries are mutable. If the key is present then the corresponding value gets updated.

>>>studentinfo['Rollno'] = 5

>>>print(studentinfo)

Deleting an item from a dictionary

Individual elements can be removed from a dictionary

Syntax:

deldictionary_name[item]

Example:

>>>del studentinfo[Rollno]

>>>print(studentinfo)

Deleting all elements

All the elements can be deleted using clear()

Syntax:

Dictionary_name.clear();

Example:

>>>studentinfo.clear()

Deleting entire dictionary

An entire dictionary can be deleted by using del.

Syntax:

deldictionary_name;

Example:

delstudentinfo;

❖❖❖❖

7

ANONYMOUS FUNCTIONS

Unit Structure

7.1 INTRODUCTION TO ANONYMOUS FUNCTION

7.1.1 What are Anonymous functions in Python?

- In Python, an anonymous function is a function that is defined without a name. It is the same as a regular python function but can be defined without a name.

- While normal functions are defined using the **def** keyword in Python, anonymous functions are defined using the **lambda** keyword.

- Hence, anonymous functions are also called lambda functions

7.1.1 Use of Anonymous functions

- We use lambda functions when we require a nameless function for a short period of time.

- In Python, we generally use it as an argument to a higher-order function

- Lambda functions are used along with other built-in functions

7.1.2 How to use Anonymous Functions in a Python program?

Syntax: Lambda arguments: Expression
Example:
l = lambda a : a + 10
print(l(20))
output: 30

In the above example **lambda a : a + 10** is the lambda function, l is the argument and a+10 is the expression that gets evaluated and returned.

7.1.3 Using Anonymous functions with other built-in functions
Anonymous or Lambda functions can be used effectively with other functions

7.1.4 Using Anonymous function with filter()
The filter() function in Python takes in a function and a list as arguments. The function is called with all the items in the list and a new list is returned which contains items for which the function evaluates to True.

Example:
my_list = [10, 15, 20, 25, 30]
new_list = list(filter(lambda x: (x%2 == 0) , my_list))
print(new_list)
Output:
[10, 20, 30]

7.1.5 Using Anonymous function with map()
The map() function in Python takes in a function and a list.
The function is called with all the items in the list and a new list is returned which contains items returned by that function for each item.
Here is an example of the use of map() function to double all the items in a list.

EXAMPLE:
my_list = [1, 2, 3, 4, 5]
new_list = list(map(lambda x: x * 10 , my_list))
print(new_list)

OUTPUT:
[10, 20, 30, 40, 50]

Questions:
1.   Explain the concept of Anonymous functions?
2.   Explain with example the use ofAnonymous function with filter().
3.   Explain with example the use ofAnonymous function with map().

❖ ❖ ❖ ❖

8

LIST COMPREHENSION

Unit Structure

8.2 List Comprehension

8.2.1 What is list comprehension?

8.2.2 Using list comprehension with an existing list

8.2.3 Using list comprehension with range()

8.2   LIST COMPREHENSION

8.2.1 What is list comprehension?

List comprehension offers a shorter syntax when you want to create a new list based on the values of an existing list.

List comprehension is considerably faster than processing a list using for loop.

The list comprehension syntax contains three parts: an expression, one or more for loop, and optionally, one or more if conditions.

The list comprehension must be in the square brackets [].

The result of the first expression will be stored in the new list. The for loop is used to iterate over the iterable object that optionally includes the if condition.

8.2.2 Using List Comprehension with an existing list

List comprehension can be easily used with existing lists to create a new list.

EXAMPLE:
```
>>>fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
>>>newlist = []

for x in fruits:
  if "a" in x:
    newlist.append(x)

print(newlist)
```

OUTPUT:
```
["apple", "banana", "mango"]
```

8.2.3 Using list comprehension with range()

List comprehension can be easily used with range() to create a new list.

EXAMPLE :
#list of even numbers with list comprehension
even_nums = [x for x in range(31) if x%2 == 0]
print(even_nums)

OUTPUT:
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30]

Questions:

1. Explain list comprehension with an Example.

2. Explain with an example how list comprehensions can be used with an existing list.

3. Explain with an example how list comprehensions can be used with range().

❖❖❖❖

9

INTRODUCTION TO OBJECT ORIENTED PROGRAMMING

Unit Structure

9.3 INTRODUCTION TO OBJECT ORIENTED PROGRAMMING

9.3.1 What is Object oriented programming?

- Object-oriented programming (OOP) is a programming model that relies on the concept of classes and objects.

- An object can be defined as a data field that has unique attributes and behavior.

- It is used to structure a software program into simple, reusable pieces of code, which are used to create individual instances of objects.

9.3.2 Major Python OOPs concept
1. **Class**
2. **Object**
3. **Method**
4. **Inheritance**
5. **Encapsulation**
6. **Polymorphism**
7. **Data Abstraction**

9.3.3Class

- A class is a collection of objects.
- Class creates a user-defined data structure, which holds its own data members and member functions.
- A class can be accessed and used by creating an instance of that class which is an object
- Classes are created by keyword class

Syntax:

      classclassname:

            statements

Example:

            classnewclass:

                  display():

                        print("new class")

9.3.4 Object

- An object is a self-contained component which consists of methods and properties to make a particular type of data useful.

- An Object is an identifiable entity with some characteristics and behavior

- An Object is an instance of a Class.

- When a class is defined, no memory is allocated but when an object is created memory is allocated.

Syntax:

      classclassname:

            statements

Object = classname()

Example:
classnewclass:

      def display():

            print("new class")

      obj=newclass()

9.3.5 Methods in Python

-     Python method is like a Python function, but it must be called on an object. And to create it, you must put it inside a class.
-     **Syntax:**

      classclassname:

            method(arguments):

Example:
classnewclass:
def display(a):
print("new class")
obj=newclass()
obj.display()

9.3.6 Inheritance

Inheritance allows us to define a class that inherits all the methods and properties from another class

Syntax:

Class Child_class(Parent_class)

Example:

classParent_class:
def display(a):
print("new class")
class Child_class(Parent_class):
obj= Child_class()
obj.display()

Types of inheritance

1.   **Single -** When one class inherits members of just one class.
2.   **Multiple -** When one class inherits members of multiple classes

Syntax:
Class Base1:
        Body of the class

Class Base2:
        Body of the class

Class Derived(Base1, Base2):
        Body of the class

Example:

classParent_class:
def display(a):
print("new class")
class Parent_class2:
def display2(a):
print("new class")
class Child_class(Parent_class, Parent_class2):
obj= Child_class()
obj.display()

3.   Multilevel

When there is inheritance at various levels
Syntax:

Class level1:
        Body of the class

Class level2(level1):
        Body of the class

Class level3(level2):
        Body of the class

61

Example:
Class level1:
def display(a):
print("new class")
Class level2(level1):
Class level3(level2):
obj= level3()
obj.display()

4.   Hierarchical
When two or more classes inherit members of one class.
          Class Base:
                    Body of the class
          Class child1(base):
                    Body of the class
          Class child2(base):
                    Body of the class

Example:
classParent_class:
def display(a):
print("new class")
class Child_class1(Parent_class):
class Child_class2(Parent_class):
obj= Child_class1()
obj.display()
obj1= Child_class2()
obj1.display()

5.   Hybrid
When there is a mix of any of the above types

9.3.7 Encapsulation
Encapsulation refers to the bundling of data, along with the methods that operate on that data, into a single unit.

9.3.8 Polymorphism
•   Polymorphism is one of the OOPs features that allow us to perform a single action in different ways.
•   With polymorphism, we can create multiple entities with the same name.

Example:
classnewclass:
def display(a):
print("new class")
def display(a,b):
obj=newclass()
obj.display()

9.3.9 Data Abstraction

Data abstraction hides the details and complexities of the program so the user only has to know how to work with an object.

Questions:

1. State and explain various object-oriented concepts?
2. Explain class in python with an example.
3. Explain inheritance in python with an example.
4. State and explain various types of inheritance.
5. Explain Hierarchical inheritance with an example.
6. Define the terms encapsulation, polymorphism, abstraction.

9.4 BUILT-IN DIR() FUNCTION

9.4.1 What is dir() function?

dir() is a powerful built-in function in Python3, which returns list of the attributes and methods of any object

The dir() function returns all properties and methods of the specified object, without the values.

This function will return all the properties and methods, even built-in properties which are default for all objects.

Syntax:

dir(*object*)

Example

import random

print(dir(random)) #lists all names of attributes in random function

9.5 METHODS

9.5.1 String for functions

Python has a set of built-in methods that you can use on strings.

Some of the important string functions are defined below:

*   format() :- Formats specified values in a string
*   lower():- Converts a string into lower case
*   upper():- Converts a string into upper case
*   replace():- Returns a string where a specified value is replaced with a specified value
*   center():-Returns a centered string
*   count():-Returns the number of times a specified value occurs in a string

```
#string methods
str = "this is string example....wow!!!"

print("str.capitalize(): ", str.capitalize())
print("str.center(40, '*') : ", str.center(40, '*'))
print("str.count('i', 4, 40) : ",str.count('i', 4, 40))
print("str.endswith(is, 2, 4) : ",str.endswith("is", 2, 4))
print("str.find(exam,10) : ",str.find("exam",10))
print("str.isalnum() : ",str.isalnum())
print("str.isalpha() : ",str.isalpha())
print("str.isdigit() : ",str.isdigit())
print("str.islower() : ",str.islower())
print("str.isspace() : ",str.isspace())
print("str.isupper() : ",str.isupper())
print("Total characters in string : ",len(str))
print("In capital letters : ",str.upper())
print("replacing is with was : ",str.replace("is","was"))
print("case change : ",str.swapcase())

str.capitalize():  This is string example....wow!!!
str.center(40, '*') :  ****this is string example....wow!!!****
str.count('i', 4, 40) :  2
str.endswith(is, 2, 4) :  True
str.find(exam,10) :  15
str.isalnum() :  False
str.isalpha() :  False
str.isdigit() :  False
str.islower() :  True
str.isspace() :  False
str.isupper() :  False
Total characters in string :  32
In capital letters :  THIS IS STRING EXAMPLE....WOW!!!
replacing is with was :  thwas was string example....wow!!!
case change :  THIS IS STRING EXAMPLE....WOW!!!
>>> |
```

9.5.2 Methods for Tuples

Python has a set of built-in methods that you can use on tuples.

- cmp(tuple1,tuple2):- Compares elements of both tuples
- len():-Gives the total length of the tuple.
- max():-Returns item from the tuple with max value.
- min():-Returns item from the tuple with min value.
- tuple():- converts a list into tuple

```
>>> tup1 = (123, 'SYIT', 'Zara')
>>> tup2 = (3.14, 'SKC')
>>> print("First tuple length: ", len(tup1))
First tuple length:  3
>>> print("2nd tuple length: ", len(tup2))
2nd tuple length:  2
>>> tup3 = (200, 365, 199, 500)
>>> print("Min element tup3: ", min(tup3))
Min element tup3:  199
>>> print("Max element tup3: ", max(tup3))
Max element tup3:  500
>>> li1 = [1, 2, 3]
>>> tup4 = tuple(li1)
>>> print(tup4)
(1, 2, 3)
```

```
>>> tup1 = ('a','p','p','l','e')
>>> print(tup1.count('p'))
2
>>> print(tup1.count('l'))
1
```

```
>>> ind = tup1.index('p')
>>> print("Index of p is: ",ind)
Index of p is:  1
>>> ind = tup1.index('e')
>>> print("Index of e is: ",ind)
Index of e is:  4
```

9.5.5 Methods of lists

Python has a set of built-in methods that you can use on lists

- sort(): Sorts the list in ascending order.
- type(list): It returns the class type of an object.
- append(): Adds one element to a list.
- extend(): Adds multiple elements to a list.

- index(): Returns the first appearance of a particular value.
- max(list): It returns an item from the list with a max value.
- min(list): It returns an item from the list with a min value.
- len(list): It gives the overall length of the list.
- clear(): Removes all the elements from the list.
- insert(): Adds a component at the required position.
- count(): Returns the number of elements with the required value.
- pop(): Removes the element at the required position.
- remove(): Removes the primary item with the desired value.
- reverse(): Reverses the order of the list.
- copy(): Returns a duplicate of the list.

```
>>> print("index for SYIT : ",li.index('SYIT'))
index for SYIT :  1
>>> li.insert(3,'Seawoods')
>>> print(li)
[2020, 'SYIT', 'SKcollege', 'Seawoods', 80.0, 'Python', 2019, 'student']
>>> li.pop()
'student'
>>> print("removing last item :",li)
removing last item : [2020, 'SYIT', 'SKcollege', 'Seawoods', 80.0, 'Python', 2019]
>>> li.remove('SKcollege')
>>> print("Removing SKcollege : ",li)
Removing SKcollege :  [2020, 'SYIT', 'Seawoods', 80.0, 'Python', 2019]
```

```
>>> li = [2020,'SYIT','SKcollege',80.00]
>>> print("list : ",li)
list :  [2020, 'SYIT', 'SKcollege', 80.0]
>>> li.append('Python')
>>> print("Updated list : ",li)
Updated list :  [2020, 'SYIT', 'SKcollege', 80.0, 'Python']
>>> print("The no of syit in the list : ", li.count('SYIT')
The no of syit in the list :  1
>>> print("The no of xyz in the list : ", li.count('syit'))
The no of xyz in the list :  0
>>> newlist = [2019, 'student']
>>> li.extend(newlist)
>>> print("Extended list ", li)
Extended list  [2020, 'SYIT', 'SKcollege', 80.0, 'Python',
```

```
>>> li.reverse()
>>> print("reversing the list : ",li)
reversing the list :  [2019, 'Python', 80.0, 'Seawoods', 'SYIT', 2020]
>>> li.sort()
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    li.sort()
TypeError: unorderable types: str() < int()
>>> li1 = ['SKcollege', 'Python', 'SYIT','Seawoods']
>>> li1.sort()
>>> print("Sorted : ",li1)
Sorted :  ['Python', 'SKcollege', 'SYIT', 'Seawoods']
```

9.5.6 methods of dictionaries

- values():-Returns a list of all the values in the dictionary
- clear() Removes all the elements from the dictionary
- update:- Updates the dictionary with the specified key-value pairs
- copy()Returns a copy of the dictionary
- fromkeys()Returns a dictionary with the specified keys and value
- get():-Returns the value of the specified key
- items():- Returns a list containing a tuple for each key value pair
- keys():- Returns a list containing the dictionary's keys
- pop():-Removes the element with the specified key
- popitem():-Removes the last inserted key-value pair
- setdefault():-Returns the value of the specified key. If the key does not exist: insert the key, with the specified value

```
>>> syit = {'name':'Tilak', 'age':20}
>>> print(syit)
{'name': 'Tilak', 'age': 20}
>>> syit.clear()
>>> print(syit)
{}
>>> tyit = syit.copy()
>>> print(tyit)
{}
>>> keys = {'a','e','i','o','u'}
>>> vowel = tyit.fromkeys(keys)
>>> print(vowel)
{'u': None, 'a': None, 'i': None, 'o': None, 'e': None}
>>> value = 'vowel'
>>> nwdict = tyit.fromkeys(keys,value)
>>> print(nwdict)
{'u': 'vowel', 'a': 'vowel', 'i': 'vowel', 'o': 'vowel'
```

```
>>> syit = {'name':'Tilak', 'age':20}
>>> syit.get('name')
'Tilak'
>>> syit.get('srn')
>>> syit.get('srn',201901001)
201901001
>>> syit.items()
dict_items([('age', 20), ('name', 'Tilak')])
>>> syit.keys()
dict_keys(['age', 'name'])
```

```
>>> d = {'age':25}
>>> syit.update(d)
>>> print(syit)
{'age': 25, 'name': 'Tilak'}
>>> d = {'srn':2018012001}
>>> syit.update(d)
>>> print(syit)
{'age': 25, 'name': 'Tilak', 'srn': 2018012001}
>>> syit.values()
dict_values([25, 'Tilak', 2018012001])
>>> syit.pop('srn')
2018012001
>>> print(syit)
{'age': 25, 'name': 'Tilak'}
```

Questions:
1. State and explain any 5 string functions.
2. State and explain any 5 tuple functions.
3. State and explain any 5 list functions.
4. State and explain any 5 dictionary functions.

❖❖❖❖