INTRODUCTION TO PYTHON PROGRAMMING- MODULE1

Dr.Thyagraju G S and Palguni GT

Contents

1.1 Python Basics

1.1.1 Introduction: Python is a general-purpose interpreted, interactive, object-oriented, and high-level programming language. It was created by Guido van Rossum during 1985- 1990. Python is named after a TV Show called 'Monty Python's Flying Circus' and not after Python-the snake. Some of the Features that Makes Python more popular are:

- Python is Simple and Easy to learn and code.
- Python is Free and Open Source. It is freely available at the https://www.python.org/. Python source code is also available to the public, one can download it, use it, and share it.
- Python is High Level Language and supports both Procedure oriented and Object-Oriented Language concepts along with dynamic memory management.
- Python is portable. Python code can be run on any platforms like Linux, Unix, Mac and Windows.
- Python is extensible and integrated. Python code can be extended and integrated with among other languages like C, C++, Java, etc.
- Python is an interpreted language. Python code is executed line by line at a time and there is no need to compile, which makes debugging easier. The
- Python has rich set of libraries for data analytics, machine learning, artificial intelligence, deep learning, mathematical computation, web app development, mobile app development, testing, etc.
- Python is a dynamically typed language. Here the data type for variable is decided at run time. As a result, there is no need to specify the type of variable.

1.1.2 Why One Should Learn Python Program?

Python Programming is a fun, creative and rewarding activity. Python is one of the most widely used programming language across the world for developing software applications. It is named as one of top picked programming languages of most of the universities and industries. Python developer is one of the "10 Most in Demand Tech Jobs of 2019"[Source : https://www.techrepublic.com/] As of February 23,

2019, the average salary for a Python developer is $123,201 per year in the United States, making it one of the most popular and lucrative careers today [source: https://www.codingdojo.com/].
Python can be used on the following:

1. Multiple Programming Paradigms
2. Web Testing
3. Data Extraction
4. Artificial Intelligence
5. Machine Learning
6. Data Science
7. Web Application and Internet Development
8. Cybersecurity

Entering Expressions into the interactive Shell

In Python, expressions are combinations of values, variables, operators, and function calls that can be evaluated to produce a result. They represent computations and return a value when executed. Here are some examples of expressions in Python:

Examples: `17, x, x+17 , 1+2*2 , X**2, x**2 + y**2`

Entering expressions into the Python interactive shell allows you to evaluate and execute code in real-time. You can use the shell as a convenient way to test and experiment with Python code. Here are some examples of entering expressions into the Python interactive shell:

Arithmetic Operations:
You can perform basic arithmetic operations, such as addition, subtraction, multiplication, and division, directly in the shell. For example:

```
>>> 2 + 3
5
>>> 4 * 5
20
>>> 10 / 3
3.3333333333333335
```

Variable Assignment:
You can assign values to variables and use those variables in expressions. For example:

```
>>> x = 5
>>> y = 2
>>> x + y
7
>>> x * y
10
```

Function Calls:
You can call built-in functions or user-defined functions within the shell. For example:

```
>>> abs(-10)
10
>>> len("Hello, World!")
13
>>> def add(a, b):
...         return a + b
...
>>> add(3, 4)
7
```

Boolean Expressions:
You can use boolean operators like and, or, and not to evaluate logical expressions. For example:

```
>>> True and False
False
>>> True or False
True
>>> not True
False
```

Conditional Statements:
You can use conditional statements like if, elif, and else to perform different actions based on conditions. For example:

```
>>> x = 10
```

```
>>> if x > 0:
...       print("Positive")
...    elif x < 0:
...       print("Negative")
...    else:
...     print("Zero")
...
Positive
```

Value: A value is a letter or a number. In Python, a value is a fundamental piece of data that can be assigned to variables, used in expressions, and manipulated by operations. Values can be of different types, such as numbers, strings, booleans, lists, tuples, dictionaries, and more. Each type of value has its own characteristics and behaviors.

Examples :

```
x = 10  # integer
y = 3.14  # floating-point number
z = 2 + 3j  # complex number
name = "John"  # string
message = 'Hello, World!'   # string
is_true = True  # Boolean Value
is_false = False
numbers = [1, 2, 3, 4, 5]
fruits = ["apple", "banana", "orange"]
coordinates = (10, 20)
person = ("John", 30, "USA")
student = {"name": "John", "age": 20, "grade": "A"}
```

type() function :

In Python, the type() function is used to determine the type of a given object or value. It returns the data type of the object as a result. The type() function is particularly useful when you need to programmatically determine the type of a variable or check if a value belongs to a specific data type. Here's an example:

Example: 1,2 and “Hello, World!”. Types are the data types to which the Values belong.
type(arg) function returns the data type of the argument as illustrated below :

```
x = 5
y = 3.14
name = "John"
is_true = True
fruits = ["apple", "banana", "orange"]
student = {"name": "John", "age": 20}

print(type(x))  # <class 'int'>
print(type(y))  # <class 'float'>
print(type(name))  # <class 'str'>
print(type(is_true))  # <class 'bool'>
print(type(fruits))  # <class 'list'>
print(type(student))  # <class 'dict'>
```

Data types

In Python, data types represent the classification or categorization of values. Each data type defines the operations that can be performed on the values, the storage format, and the behavior of the values. Python supports several built-in data types. Here are the commonly used data types supported in Python, along with examples:

Numeric Data Types:

int: Integers represent whole numbers, positive or negative, without decimal points.
Example: x = 5
float: Floating-point numbers represent decimal or floating-point values.
Example: y = 3.14
complex: Complex numbers consist of a real and an imaginary part.
Example: z = 2 + 3j

String Data Type:

str: Strings represent sequences of characters enclosed in single quotes (' ') or double quotes (" ").
Example: name = "John"

Boolean Data Type:

bool: Booleans represent either True or False, denoting logical values.
Example: is_true = True

Sequence Data Types:

list: Lists are ordered collections of items enclosed in square brackets ([]). They can contain values of any type and are mutable.
Example: fruits = ["apple", "banana", "orange"]
tuple: Tuples are similar to lists but are enclosed in parentheses (()). They are immutable, meaning their values cannot be changed once defined.
Example: coordinates = (10, 20)

Mapping Data Type:

dict: Dictionaries are unordered collections of key-value pairs enclosed in curly braces ({}). They provide a way to associate values with unique keys.
Example: student = {"name": "John", "age": 20}

Set Data Type:

set: Sets represent unordered collections of unique elements enclosed in curly braces ({}). They do not allow duplicate values.
Example: numbers = {1, 2, 3, 4, 5}

None Type:

None: The None type represents the absence of a value or a null value. It is often used to indicate the absence of a meaningful result.
Example: result = None

String Concatenation and Replication

String concatenation and replication are operations that allow you to combine and repeat strings in Python. Here's an explanation of each operation with examples:

String Concatenation:

String concatenation is the process of combining two or more strings together to create a single string. In Python, you can concatenate strings using the + operator. Here's an example:

```
greeting = "Hello"
name = "John"
message = greeting + " " + name
print(message)  # Output: Hello John
```

In this example, the + operator is used to concatenate the strings greeting, a space (" "), and name, resulting in the string "Hello John". The concatenated string is then stored in the message variable and printed.

String Replication:

String replication allows you to repeat a string multiple times. In Python, you can replicate a string by using the * operator. Here's an example:

```
fruit = "apple"
repeated_fruit = fruit * 3
print(repeated_fruit)  # Output: appleappleapple
```

In this example, the * operator is used to replicate the string "apple" three times, resulting in the string "appleappleapple". The replicated string is stored in the repeated fruit variable and printed.

String concatenation and replication can be combined to achieve more complex string operations. Here's an example that demonstrates both concatenation and replication:

```
word = "Hi"
punctuation = "!"
greeting = word * 3 + punctuation
print(greeting)  # Output: HiHiHi!
```

Variables:

A variable is a name that refers to a value. In Python, a variable is a named storage location that holds a value. It allows you to assign values to identifiers and refer to those values by using the identifiers later in the program. An assignment statement creates new variables as illustrated in the example below:

x = 10

In this example, the value 10 is assigned to the variable x. Now, x holds the value 10, and you can refer to it later in the program.

Examples:

Message = '*Python Programming* ',
p =1000, t= 2, r=3.142,
Si = p*t*r/100,
pi = 3.1415926535897931,
area_of _circle = *pi*r*r.*

To know the type of the variable one can use type () function. **Ex:** type(p)
To display the value of a variable, you can use a print statement:
Ex: print (Si) ; print(pi)

Rules for writing Variable names

1. **Variable names** can be a combination of letters in lowercase (a to z) or uppercase (A to Z) or digits (0 to 9) or an underscore (_).
2. **Variable names** cannot start with a number/digit.
3. **Keywords** cannot be used as Variable names.
4. Special symbols like **!, @, #, $, %** etc. cannot be used in Variable names.
5. Variable names can be of any length.
6. Variable name must be of single word.

Table: Valid Variable Names and Invalid Variable Names

Valid Variable Names	Invalid Variable Names
python12	current- account(hyphens are not allowed)
Simple	savings account (spaces are not allowed)
interest_year	4freinds (can't begin with a number)
_rate_of_interest	1975 (can't begin with a number)
_spam	10April_$ (cannot begin with a number and special characters like $ are not allowed)
HAM	Principle#@(special characters like # and @ are not allowed)
account1234	'bear' (special characters like ' is not allowed)

Note:

Variable names are case-sensitive, meaning that velocity, VELOCITY, Velocity, and velocity are four different variables. It is a Python convention to start your variables with a lowercase letter.

Storing Values in a Variables:

Values can be stored in a variable using an Assignment statement. An assignment statement consists of a variable name, an equal (=) sign and the value to be stored.

Example 1:

x = 40

In this example, the value 10 is assigned to the variable x. The variable x now holds the value 10, and you can use x to refer to that value throughout the program.

Example 2:

a, b, c = 1, 2, 3

In this example, the values 1, 2, and 3 are assigned to variables a, b, and c respectively. Each value is assigned to its corresponding variable based on the order of appearance.

Example 3:

```
x = 5
y = 3
result = x + y
```

In this example, the variables x and y hold the values 5 and 3 respectively. The expression x + y is evaluated, and the result 8 is assigned to the variable result.

Example 4:

```
x = 10
x = x + 5  # x is updated to 15
```

In this example, the variable x initially holds the value 10. The expression x + 5 evaluates to 15, and that value is assigned back to the variable x.

My First Program :

```
# This program demonstrates the usage of common built-in functions in Python

# Using the print() function to display a message
print("Welcome to the Python function demo!")

# Using the len() function to determine the length of a string
text = input("Enter a word or phrase: ")
length = len(text)
print("The length of the entered text is:", length)

# Using the input() function to get user input
name = input("Enter your name: ")
age = input("Enter your age: ")

# Using the int() function to convert a string to an integer
age = int(age)
age_in_future = age + 10
print("In 10 years, you will be", age_in_future, "years old.")

# Using the str() function to convert an integer to a string
message = "Hello, " + name + "! You are " + str(age) + " years old."
print(message)
```

Dissecting the Sample Program

Sample program comprises executable statements containing comments and built in functions like print () , input () , len() , int() and str().

Explanation of the program:

The program starts with a **comment** explaining the purpose of the program.
The **print()** function is used to display the welcome message.
The **input()** function is used to prompt the user to enter a word or phrase. The value entered by the user is stored in the text variable.

The **len()** function is used to determine the length of the text string, and the result is stored in the length variable.
The program displays a message with the length of the entered text using the **print()** function.
The **input()** function is used again to get the user's name and age. The values entered by the user are stored in the name and age variables, respectively.
The **int()** function is used to convert the age value from a string to an integer.
The program calculates the user's age in 10 years by adding 10 to the age variable, and the result is stored in the age_in_future variable.
The program displays a message with the user's name, age, and the calculated age in 10 years using the **print()** function.
The **str()** function is used to convert the age integer back to a string so that it can be concatenated with other strings in the message variable.
The final message is displayed using the **print()** function.

The program demonstrates the usage of these functions: print() for displaying messages, len() for determining the length of a string, input() for obtaining user input, int() for converting a string to an integer, and str() for converting an integer to a string.

Comments:

Comments are readable explanation or descriptions that help programmers better understand the intent and functionality of the source code. Comments are completely ignored by interpreter.

Advantages of Using Comments:

1. Makes code more readable and understandable.
2. Helps to remember why certain blocks of code were written.
3. Can also be used to ignore some code while testing other blocks of code**.**

Single Line Comments in Python:

The hash symbol #is used to write a single line comment.

Example:

 # Printing a message
print(“ Enter your Name “)
myName = input (“ Enter Your Name”) **# Read your name to myName**

Multiline Comments in Python:

1. Using **#** at the beginning of each line of comment on multiple lines

Example:
It is a
multiline
comment

2. Using String Literals **'''** at the beginning and end of multiple lines

Example:
'''
I am a
Multiline comment!
'''

The print() Function :

The print function is used to display the string value written within pair of double quotes inside the parentheses on the screen .

print('It is Good to meet you, ' + myName)
print('The length of your name is:')
print(len(myName))
print('You Will be ' + str(int(myAge)+1) + ' in a year.')

The line *print('The length of your name is:')* means “Print out the text in the string ‘'The length of your name is:’ . When Python executes the print statement, python interpreter calls the *print()*function and the string value is being *passed* to the function. The value within print() is called argument . Quotes within parentheses marks where the string begins and ends ; they are not part of the string value.

The input() function

This function is used to take the input from the user . Whatever the user enter as input, input() function convert it into a string . if you enter an integer value still input() function convert it into a string . The programmer is needed to convert it into an integer in your code using *typecasting*.
Myname = input("Enter your name")

Reading multiple values using input()

Programmer often want as user to enter multiple values in one line . In Python user can take multiple values or inputs in one line by using split() method . It breaks the given input by the specified separator. If separator is not provided then any white space is a separator. Generally, user use a split() method to split a Python string but one can used it in taking multiple input.

Example:

```
>>> x, y,z = input ("Enter three values").split()
Enter three values 2 3 4

>>> x
'2'
>>> y
'3'
>>> z
'4'
```

The len() Function

In Python, the len() function is used to determine the length of an object, such as a string, list, tuple, or any other iterable. It returns the number of elements or characters present in the object. Here's an explanation of the len() function with an example:

```
text = "Hello, World!"
length = len(text)
print(length)  # Output: 13
```

The str() , int() and float Functions :

The str(), int(), and float() functions in Python are used to convert values between different data types. Here's an explanation of each function with examples:

str() Function:
The str() function is used to convert a value to a string data type. It takes any valid Python object as an argument and returns a string representation of that object. Here's an example:

```
number = 10
number_str = str(number)
print(number_str)  # Output: "10"
```

str() function can be used convert integer or floating numbers into string data type.

int() Function:
The int() function is used to convert a value to an integer data type. It can convert a string or a float to an integer by truncating any decimal places. Here are some examples:

```
number_str = "20"
number_int = int(number_str)
print(number_int)  # Output: 20

float_num = 3.14
float_int = int(float_num)
print(float_int)  # Output: 3
```

In the first example, the int() function converts the string "20" to an integer value 20. In the second example, the int() function converts the float value 3.14 to an integer by truncating the decimal places, resulting in the value 3.

float() Function:
The float() function is used to convert a value to a floating-point data type. It can convert a string or an integer to a float. Here's an example:

```
number_str = "3.14"
number_float = float(number_str)
print(number_float)  # Output: 3.14
```

```
integer_num = 10
integer_float = float(integer_num)
print(integer_float)  # Output: 10.0
```

In the first example, the float() function converts the string "3.14" to a floating-point value 3.14. In the second example, the float() function converts the integer value 10 to a float value 10.0.

These functions are useful when you need to convert values between different data types in Python. They allow you to perform operations and manipulate values in the desired format.

Operators and operands

- Operators are special symbols that represent computations like addition and multiplication. The values the operator is applied to are called operands.
- The operators **+, -, *, /, and **** perform *addition, subtraction, multiplication, division, and exponentiation*, as in the following examples:

Table: Operators and Examples

Operator	Operation	Example	Evaluates to
**	Exponent	5**3	125
%	Modulus/Remainder	33%7	5
//	Integer Division/Floored quotient	33//5	6
/	Division	23/7	3.2857142857142856
*	Multiplication	7*8	56
-	Subtraction	8 – 5	3
+	Addition	7+ 3	10

Order of operations

- When more than one operator appears in an expression, the order of evaluation depends on the rules of precedence.

PEMDAS order of operation is followed in Python:

- **Parentheses** have the highest precedence and can be used to force an expression to evaluate in the order you want.
- **Exponentiation** has the next highest precedence,
- **Multiplication and Division** have the same precedence, which is higher than
- **Addition and Subtraction**, which also have the same precedence.
- **Operators with the same precedence** are evaluated from left to right.

Following examples illustrates the evaluation of expressions by Python interpreter. In each case the programmer must enter the expression, python interpreter evaluates the expression to a single value .

```
>>> 5+4*3
17
>>> (4+5)*3
27
>>> 12345678*45678
563925879684
>>> 3**5
243
>>> 22//7
3
>>> 22/7
3.142857142857143
>>> 27%5
2
>>>3 +  3
6
>>> (5-2)*((8+4)/(5-2))
12.0
```

Python interpreter evaluates parts of the expression as per the PEMDAS rule until it becomes a single value as illustrated below:

(5-2)*((8+4)/(5-2))

3 * ((8+4)/(5-2))

3*(12/(5-2))

3*(12/3)

3*4.0

12.0

Note: If you type invalid expressions, python interpreter will not be able to understand it and will display a SyntaxError message as illustrated below:

```
>>> 45+*
SyntaxError: invalid syntax
>>> 45++75)
SyntaxError: unmatched ')'
>>> 43-
SyntaxError: invalid syntax
>>> 43+5+*7
SyntaxError: invalid syntax
```

Python Character Set :

The set of valid characters recognized by Python like letter, digit or any other symbol. The latest version of Python recognizes Unicode character set. Python supports the following character set:

- **Letters** : A-Z ,a-z
- **Digits** :0-9
- **Special Symbols** : space +-/***()[]{}//=!= == <> ,”””,;:%!#?$&^⇔=@_
- **White Spaces** : Blank Space, tabs(->), Carriage return , new line , form feed
- **Other Characters** : All other 256 ACII and Unicode characters

Python Tokens:

A token (lexical unit) is the smallest element of Python script that is meaningful to the interpreter. Python has following categories of tokens: Identifiers, Keywords, Literals , operators and delimiters.

Identifiers: Identifiers are names that you give to a variable, class or Function. There are certain rules for naming identifiers similar to the variable declaration rules, such as : No Special character except_ , Keywords are not used as identifiers , the first character of an identifier should be _ underscore or a character , but a number is not valid for identifiers and identifiers are case sensitive .

```
# Variables
my_variable = 10
counter = 0
# Function
def calculate_area(length, width):
    return length * width
# Class
class MyClass:
    def __init__(self, name):
        self.name = name
# Module
import math
# Usage
rectangle_area = calculate_area(5, 3)
print(rectangle_area)  # Output: 15

my_object = MyClass("John")
print(my_object.name)  # Output: John

print(math.pi)  # Output: 3.141592653589793
```

In the above example, we have used identifiers like **my_variable, counter, calculate_area, MyClass, and math.**

Literals: A fixed numeric or non-numeric value is called a **literal**. Literals may be string, numbers (int, long, float and complex), Boolean (True or False), NONE and Operators.

1. Numeric literals: Numeric literals represent numeric values such as integers, floating-point numbers, and complex numbers.
Examples: x = 10, y = 3.14, z = 2 + 3j
2. **String literals:** String literals represent sequences of characters enclosed in either single quotes (') or double quotes (").
Examples: name = 'John', sage = "Hello, world!“
3. **Boolean literals**: Boolean literals represent the truth values True and False.
Examples: is_valid = True
4. **None literal:** The None literal represents the absence of a value or a null value. It is often used to indicate the absence of a meaningful result or as an initial value for variables.
Example: result = None
4. **Operator Literals** : Operator literals include arithmetic operators, comparison operators, assignment operators, logical operators, and more.
Examples: +,-,/,//,%,*,**, <,>,!=,==,and,or,not,etc.

Operators : A Symbol or a word that performs some kind of operation on given values and returns the result. There are 7 types of operators available for Python: Arithmetic Operator ,Assignment Operator, Comparison Operator, Logical Operator , Bitwise Operator , Identity Operator and Membership Operator .

1. Arithmetic operators: +, -, *, /, %, **, //
2. Assignment operators: =, +=, -=, *=, /=, %=, **=, //=
3. Comparison operators: ==, !=, >, <, >=, <=
4. Logical operators: and, or, not
5. Bitwise operators: &, |, ^, ~, <<, >>
6. Membership operators: in, not in
7. Identity operators: is, is not

Delimiters: Delimiters are the symbols which can be used as separators of values or to enclose some values.

> **Examples** : Comma (,),Colon (:),Parentheses ((and)),Square brackets ([and]),Curly braces ({ and }),Quotation marks (' and ") and Backslash (\)

Note : Comments and # symbol used to insert a comment is not a token.

Keywords: The reserved words of Python which have a special fixed meaning for the interpreter are called keywords. No keyword can be used as an identifier or variable names. There are 35 keywords in python as listed below:

Keyword	Description
and	Logical and operator
as	Alias
assert	Used for debugging
async	Used to make a function asynchronous by adding the async keyword before the function's regular definition
await	Used in asynchronous functions to specify a point in the function where control is given back to the event loop for other functions to run. You can use it by placing the await keyword in front of a call to any async function
break	To break out of a loop
class	To define a class
continue	For skipping the statements and conitinuing the next iteration
def	For defining user defined functions
del	To delete an object
elif	Conditional statement, same as else if
else	Conditional statement
except	Used in exception handling
False	Boolean Value
finally	Used in exception handling , to execute a block of code no matter whether exception is there or not
for	Used to create for loop – iterative statement
from	Used to import specific parts of a module
global	Used to declare global variable
if	Conditional /decision making statement
import	Used to import a module or library
in	Used to check if a value if present in list, tuple, dictionaries , sets ,etc.
is	To check if two variables are equal
lambda	Used for defining an anonymous function
None	Used to represent a null value
nonlocal	To declare a non-local variable

not	A logical operator
or	A logical operator
pass	A null statement , a statement that will do nothing
raise	To raise an exception
return	To exit a function and return a value
True	Boolean Value
try	Used in exception handling
while	For creating a while iterative loop
with	Used to simplify exception handling
yield	To end a function , returns a generator

Following code segment can be used to obtain the list of python keywords :

```
import keyword
# Get the list of keywords
print(keyword.kwlist)
print("\n Total Number of Keywords: ",len(keyword.kwlist))

# Output :

['False', 'None', 'True', '__peg_parser__', 'and', 'as', 'assert', 'async', 'await', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']
 Total Number of Keywords:  36
```

Simple Python Programs

1. **To perform basic calculations**

```
# Sample numbers
num1 = int(input("Enter first number"))
num2 = int(input("Enter second number"))

# Calculate the sum of two numbers
sum_result = num1 + num2
print("Sum:", sum_result)

# Calculate the product of two numbers
product_result = num1 * num2
print("Product:", product_result)

# Calculate the difference of two numbers
difference_result = num1 - num2
print("Difference:", difference_result)

# Calculate the division of two numbers
division_result = num1 / num2
print("Division:", division_result)

# Calculate the integer division of two numbers
integer_division_result = num1 // num2
print("Integer Division:", integer_division_result)

# Calculate the modulo division of two numbers
modulo_division_result = num1 % num2
print("Modulo Division:", modulo_division_result)
```

 # output

```
Enter first number 3
Enter second number 2
Sum: 5
Product: 6
```

Difference: 1
Division: 1.5
Integer Division: 1
Modulo Division: 1

2. To find the area of a circle

```
import math
# Take input for the radius of the circle
radius = float(input("Enter the radius of the circle: "))
# Calculate the area of the circle
area = math.pi * radius**2
# Display the result
print("The area of the circle is:", area)
```

 # output

Enter the radius of the circle: 2.5
The area of the circle is: 19.634954084936208

3. To find the simple interest

```
# Take input for principal amount, rate, and time
principal = float(input("Enter the principal amount: "))
rate = float(input("Enter the interest rate: "))
time = float(input("Enter the time period (in years): "))

# Calculate the simple interest
simple_interest = (principal * rate * time) / 100

# Display the result
print("The simple interest is:", simple_interest)
```

 # Output
Enter the principal amount: 10000
Enter the interest rate: 2.5
Enter the time period (in years): 3
The simple interest is: 750.0

Questions for Practice:

1. What is Python ? Who created Python?
2. What are the features of Python which makes it more popular?
3. List out the different jobs available for Python programmer.
4. What is the role of programmer? Lis two skills required to become good programmer.
5. Discuss, Why Should you learn to write Programs?
6. What is Zen of Python?
7. Discuss, with snapshots how to install latest version of IDLE Python and Jupyter Python.
8. Illustrate with examples how to interact with IDLE Python.
9. Explain how mathematical expressions can be executed in interactive shell.
10. Illustrate with example how write and execute programs in Jupyter Editor.
11. Explain the different components of Jupyter Editor.
12. Define Program. Differentiate between Compiler and Interpreter. Give Examples.
13. What are Python words and sentences? Explain with an example for each.
14. Classify the following list of items into variables, values, operators, strings, and keywords:
    List of items: *, + , - , **, < ,> , 'hello' , ‘ I am ok . How are you’, -88.8, /, 5, and, is , not , while , for, async, x, si, p , time , rate , velocity , speed, acc , % ,&, ! , ||
15. What are expressions? Illustrate the different types of expressions with examples.
16. What are data types? Classify the different data types in python with examples.
17. What are Python Variables? What rules one should follow to name the variables.
18. Give 5 examples for valid and invalid variables.
19. Discuss how to store values in a variable.
20. Write a sample program and dissect the program with explanation.
21. What are comments? What are the advantages of Comments? Explain the different ways of writing comments.
22. Give examples for single and multiline comments.
23. Explain the working and usage of print() function with examples.

24. Explain the working and usage of input() function with examples.
25. Give example to read multiple values using input().
26. Define operands and operators. Discuss PEMDAS rules with examples.
27. What are keywords? How many keywords are there in current version of Python?
28. Write a program to display all keywords in the current version of Python.
29. What are Python comments? Explain their importance with programming examples.
30. Explain print () , input() and split() functions with example.
31. Write a program for the following:

1. To read and print a single value in a single line
2. To read and print multiple values in a single line

32. Explain the following different types of errors: Syntax errors, Semantic errors and Logic Errors.
33. Explain the following functions with example: len() , str() , int() , float()
34. Predict the out put and justify your answer: (i) -11%9 (ii) 7.7//7 (iii) (200 – 70)*10/5 (iv) not "False" (v) 5*|**2
35. List the rules to describe a variable in Python. Demonstrate at least three different types of variables uses with an example program.
36. Explain the following :

1. Skills necessary for a programmer
2. Interactive Mode
3. Short circuit evaluation of expression
4. Modulus operator.

37. Mention three types of errors encountered in python program.
38. Define the following with example : Values and Types , Variables , Expressions , Keywords , Statements , Operators and Operands, Order of Operations , Modulus Operators , String operations and Comments.
39. What three functions can be used to get the integer, floating-point number, or string version of a value?

Programs for Practice:

Write and execute python programs for the following :

1. To prompt a user for their name and then welcomes them.
2. To prompt a user for days and rate per day to compute gross salary.
3. To read the following input and display :
    a. Name :
    b. USN:
    c. Roll No:
    d. Mobile No:
    e. E-Mail Id:
    f. Percentage of Marks:
4. To find the simple interest for a given value of P, T and R. Program should take input from the user.
5. To find the compound interest.
6. To read two integers and find the sum, diff, mult and div.
7. To Convert given Celsius to Fahrenheit temperature.
8. To print ascii value of a character.
9. To display all the keywords.
10. To print the following string in a specific format :""Twinkle, twinkle, little star, How I wonder what you are! Up above the world so high, Like a diamond in the sky. Twinkle, twinkle, little star, How I wonder what you are" .

    Output :

```
Twinkle, twinkle, little star,
   How I wonder what you are!
         Up above the world so high,
         Like a diamond in the sky.
Twinkle, twinkle, little star,
   How I wonder what you are
```

11. To get a python version.
12. To display the current date and time
13. To accept the radius of a circle from the user and compute the area.
14. To print the calendar of a given month and year.
15. To check whether a file exists .

16. To determine if a Python shell is executing in 32 bit or 64 bit mode on OS.
17. To get OS name , platform and release information .
18. To locate Python site packages.
19. To call an external command in python.
20. To get path and name of the file that is currently executing.
21. To parse a string to float or integer.
22. To list all files in a directory in Python.
23. To print without newline or space.
24. To determine profiling of Python programs.
25. To print to stderr.
26. To access environment variables.
27. To get the current username.
28. To find the local IP addresses using Pythons stdlib.
29. To get execution time for a python method.
30. To convert height in meters to centimeters.
31. To Convert all units of time to seconds
32. To convert the distance in feet to inches , yards and miles.
33. To calculate body mass index.
34. Given variables x = 15 and y = 30 , write a Python program to print "15+30=45".
35. To get the identity of the object
36. To check whether a string is numeric.
37. To get the system time .
38. To clear the screen or terminal
39. To calculate the time runs (difference between start and current time) of a program.
40. To input integer if not generate error.

1.2 Flow Control

Syllabus: Boolean Values, Comparison Operators, Boolean Operators, Mixing Boolean and Comparison Operators, Elements of Flow Control, Program Execution, Flow Control Statements, Importing Modules, Ending a Program Early with sys.exit().

Boolean Values: A Boolean value is either true or false. It is named after the British mathematician, George Boole, who first formulated Boolean algebra. In Python the two Boolean Values are True and False and the Python type is bool. Enter the following into the Python shell and observe the output.

type(**True**) # output : **bool**
type(**False**) # output : **bool**
type(**true**) # output: Name Error : name " true" is not defined
type(**false**) # output: Name Error : name " false" is not defined
context = True
print(context) #output : **True**

A **Boolean expression** is an expression that evaluated to produce a result which is a Boolean value. For example, the operator '==' tests if two values are equal. It produces (or yields) a Boolean value:

5 == (1+4) # output : True

5 == 6 # output: False

In the first statement the two operands evaluate to equal values, so the expression evaluates to **True**; in the second statement, 5 is not equal to 6 we get **False.**

P = "hel"
P + "lo" == "hello" # output: True

In the above example since the concatenated value P + "lo" is "hello" the expression evaluates to **True .**

Comparison Operators

Comparison operators compare two values and evaluate down to a single Boolean value. The **==** operator is one of six common comparison operators which all produce a bool result. Table lists all the comparison operators:

Operator	Meaning
==	Equal to
!=	Not Equal to
<	Less than
>	Greater than
<=	Les than or equal to
>=	Greater than or equal to

Comparison operators evaluate to **True or False** depending on the values we provide to them. Consider following expressions:

55 == 55 # output: True

55 == 79 # output: False

7!=10 # output : True

7!=7 #output : False

True == True # output: True

True != False # output: True

Based on the above observations it is clear that == (equal) evaluates to True when the value on both sides are the same , and != (not equal to) evaluates to True when the two values are different . T Equal to and Not equal to operators can work with values of any data type.

The other comparison operators like <, >, <= and >= work properly only with integer and floating-point values.

12< 13 # output: True

55.55 > 66.75 # output : False
“tag”< = 2 # output : Type error : ‘<’ is not supported between instances of ‘str’ and ‘int’.

Difference between == and = Operator

=	==
It is an assignment operator	It is a comparison operator
It is used for assigning the value to a variable	It is used for comparing two values. It returns 1 if both the value is equal otherwise returns 0
Constant term cannot be place on left hand side **Example:** 1= x; is invalid	Constant term can be placed in the left-hand side. Example: 1 ==1 is valid and return 1

Boolean Operators:

Boolean operators evaluate the expression to Boolean Values True/False. Python supports three Boolean operators **and, or & not**. Based on the number of operands required they can be classified into **Binary Boolean operators** and **Unary Boolean Operators**.

Binary Boolean operators : and & or

Since both **and & or** operators takes two operands, they are considered as binary operators. The **and** operator returns true value if both operands are true and return false otherwise. While or operator returns false when both operands are false and returns **true** otherwise.

True and True # output : True
True and False # output : False
False and True # output : False
False and False # output :False

True or True # output : True
True or False # output : True
False or True # output : True
False or False # output :False

Following truth tables illustrates all possible logical combinations and values for OR and AND Boolean binary operators.

Op1	Op2	Op1 and Op2
True	False	False
False	True	False
False	False	False
True	True	True

Op1	Op2	Op1 or Op2
True	False	True
False	True	True
False	False	False
True	True	True

Not operator: It is a unary operator and evaluates the expression to opposite value true or false as illustrated below :

not True # output : False
not False # output : True
not not not not True # output : True

Truth Table for **not** operator:

op	not op
True	False
False	True

Examples for Mixing Boolean and Comparison Operators: Boolean operators and comparison operators can be used in combination as illustrated below :

x = 10

y = 20

x<y and x>y **# output : False**

(x<y) and (x!=y) or (x*2) and (x<20 or y<20) **# output : True**

2+2 == 4 and not 2+2 == 5 and 2*2 == 2+2 **#output : True**

5*7 +8 == 7 or not 5+7 ==10 and 5*4 ==20 **#output : True**

Elements of Flow Control:

Flow control statements often starts with condition followed by a block of code called clause. The two elements of Flow Control are discussed below:

a) Conditions:

Conditions are Boolean expressions with the boolean values True or False. Flow control statements decides what to do based on the condition whether it is true or false.

b) Blocks of Code /Clause:

The set of more than one statements grouped with same indentation so that they are syntactically equivalent to a single statement is known as Block or Compound Statement. One can tell when a block begins, and ends based on indentation of the statements. Following are three rules for blocks:

1. Blocks begin when the indentation increases
2. Blocks can have nested blocks
3. Blocks end when the indentation decreases to zero

Example 1:

```
x = int(input("Enter a number:  ")) # Block
if  x>=10: # Condition
        x = x + 25    # Block belonging to if
        y = x
        print(x,y)
print("Next statement") # Next Block
```

Example 2: Nested Blocks

```
N = int(input("Enter a number of your choice"))
if  n > 0:
        print("Positive") # Block  of outer if
        if n%2==0:
                print("Multiple of Two") # Block of Inner if
print("End")
```

Flow Control statements

Flow control statements in Python are used to control the order of execution and make decisions based on certain conditions. The main flow control statements in Python include:

Conditional Control Statements:

- **if statement**: Executes a block of code if a specified condition is true.
- **elif statement**: Allows you to check additional conditions if the previous if or elif conditions are false.
- **else statement**: Executes a block of code if none of the previous conditions are true.

Looping Statements:

- **for loop:** Iterates over a sequence (such as a list, tuple, string, or range) and executes a block of code for each item in the sequence.
- **while loop:** Repeats a block of code as long as a specified condition is true.

Loop Control Statements:

- **break statement:** Terminates the innermost loop and continues with the next statement after the loop.
- **continue statement**: Skips the rest of the current iteration and moves to the next iteration of the loop.
- **pass statement**: Acts as a placeholder, allowing you to create empty code blocks without causing syntax errors.

Exception Handling Statements:

- **try, except, finally statements:** Used to catch and handle exceptions that occur during program execution.
- **try statement:** Defines a block of code where exceptions might occur.
- **except statement**: Specifies the code to execute if a specific exception occurs within the try block.
- **finally statement**: Defines a block of code that will be executed regardless of whether an exception occurred or not.

Types of Condition Control Statements

The different two-way selection statements supported by Python language are:

1. *if* statement
2. *if- else* statement
3. Nested *if else* statement
4. *if – elif* ladder

1. if statement:

It is basically a two-way decision statement and it is used in conjunction with an expression. *It is used to **execute** a set of statements if the condition is true. If the condition is false it **skips** executing those set of statements.* The syntax and flow chart of if statement is as illustrated below:

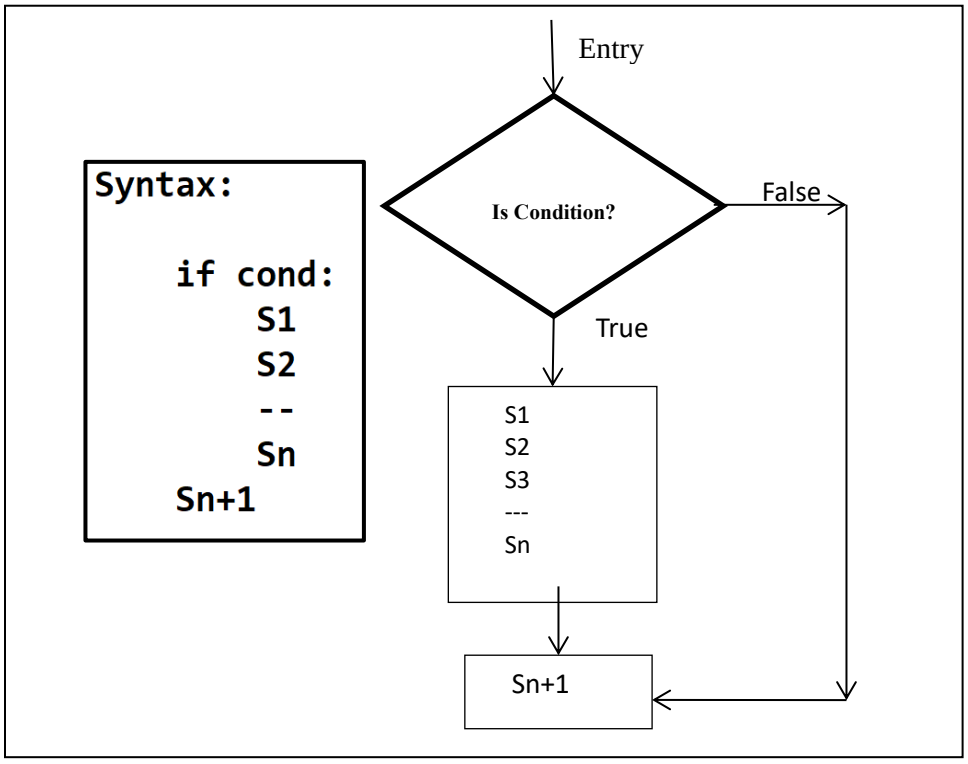


Fig: Syntax and flow diagram for if statement

Example1: Python program to find the largest number using if statement.

```
x,y = input("Enter two integer numbers: ").split()
n1 = int(x)
n2 = int(y)
large = n1
if n2>large:
    large = n2
print("Largest number = ",large)
```

```
Enter two integer numbers: 3 4
Largest number =  4
```

Example2: Python Program to determine whether a person is eligible to vote using if.

```
age = int(input("Enter your age: "))
if age>=18:
    print("You are eligible to vote")
```

```
Enter your age: 25
You are eligible to vote
```

2. If else statement:

It is an extension of if statement. It is used to execute any one set of two set of statements at a time. If condition is true it executes one set of statements otherwise it executes another set of statements. The syntax and flow diagram of if else is as shown in the figure below. As illustrated in the figure if the condition is true the set of statements {S11,S12,------S1n} gets executed else if the condition is false the set of statements {S21,S22,S23------S2n} gets executed.

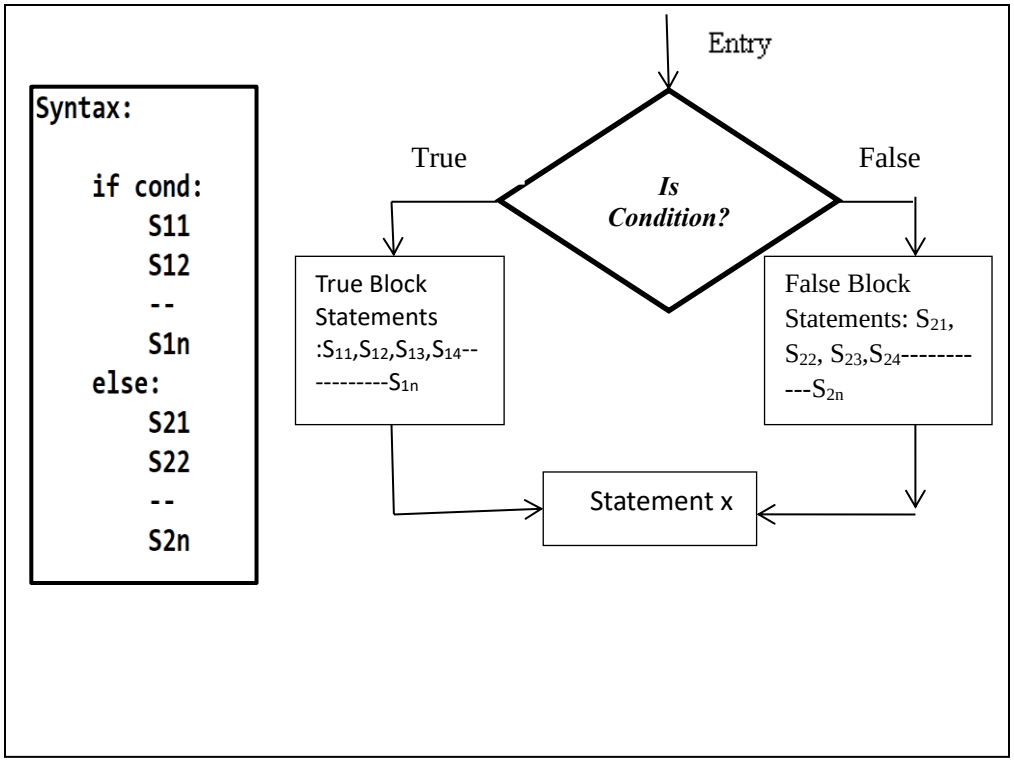


Fig: Syntax and flow diagram for if else statement

Example: Program to check whether a given number is even or odd using if else.

```
1  n = int(input("Enter a number: "))
2  if n%2==0:
3      print("Entered number is Even")
4  else:
5      print("Entered number is Odd")
```

```
Enter a number: 2
Entered number is Even
```

3. Nested if else:

When a series of decisions are involved, we may have to use more than one *if else* statement in nested form. The nested *if else* statements are multi decision statements which consist of *if else* control statement within another *if or else* section. The syntax and flow diagram for nested if else is as shown below:

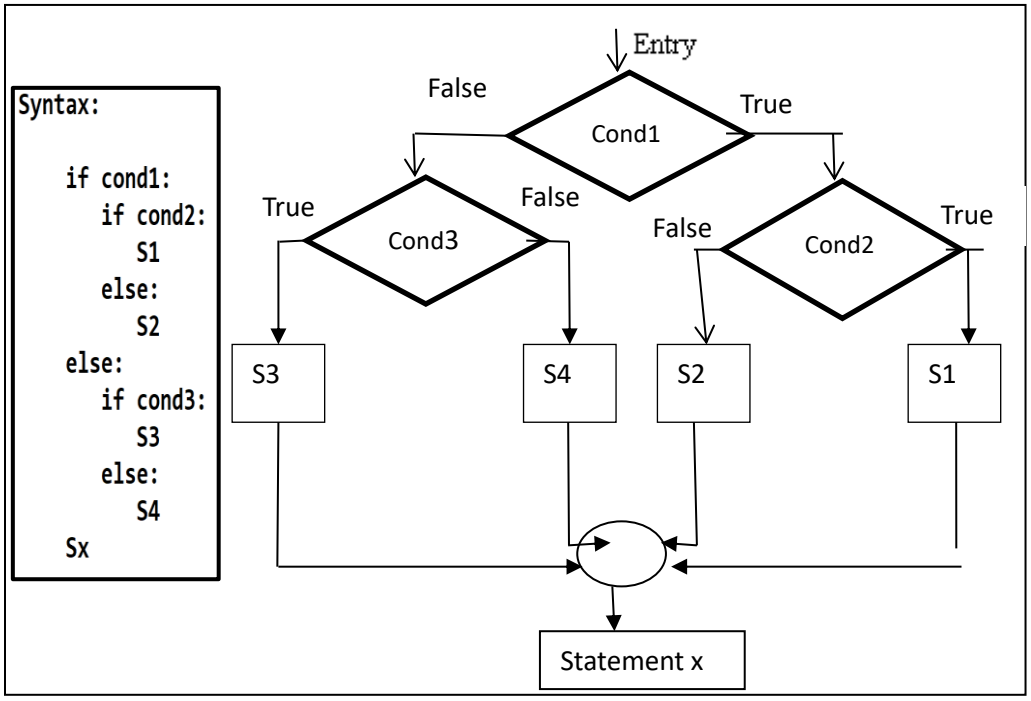


Fig : Syntax and flow diagram for nested if else statement

Example:

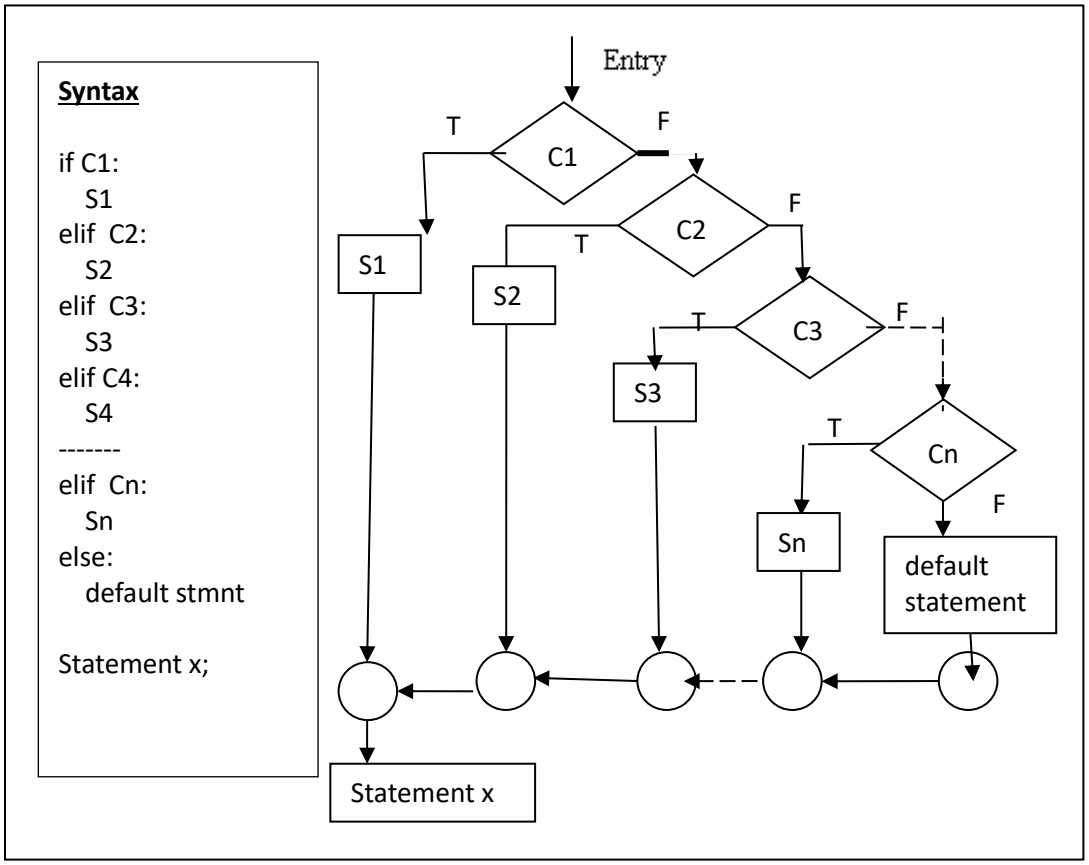
```
a,b,c = input("Enter three numbers : ").split()
a = int(a)
b = int(b)
c = int(c)
if a>b:
    if a>c:
        print("{0} is largest".format(a))
    else:
        print("{0} is largest".format(c));
else:
    if b>c:
        print("{0} is largest".format(b));
    else:
        print("{0} is largest".format(c))
```

```
Enter three numbers : -2 3 -7
3 is largest
```

4. if – elif ladder:

Cascaded *if elif else* is a multipath decision statements which consist of chain of *if elseif* where in the nesting take place only in else block. As soon as one of the conditions controlling the 'if' is true , then statement /statements associated with that 'if' are executed and the rest of the ladder is bypassed. If condition is false then it checks the first *elif* condition , if it is found true then first elif is executed. However, if none of the elif ladder is correct then the final else statement will be executed and control flows from top to down.

The syntax and flow diagram for cascaded *if else* is as shown in figure.



Example:

```
marks = int(input("Enter the marks [0 -100] : "))
if marks>=0 and marks<=39:
    print("Grade F\n");
elif marks>=40 and marks<=49:
    print("Grade E")
elif marks>=50 and marks<=59:
    print("Grade D")
elif marks>=60 and marks<=69:
    print("Grade C")
elif marks>=70 and marks<=79:
    print("Grade B")
elif marks>=80 and marks<=89:
    print("Grade A")
elif marks>=90 and marks<=100:
    print("Outstanding")
else:
    print("Invalid Entry")
```

```
Enter the marks [0 -100] : 90
Outstanding
```

3. Iteration or looping:

The statement which are used to repeat a set of statements repeatedly for a given number of times or until a given condition is satisfied is called as *looping constructs or looping statements*. The set or block of statements used for looping is called loop.

Types of Looping statements:

Depending on the position of the control statement in the loop the looping statements are classified into the following two types:

1. Entry controlled Loop
2. Exit Controlled Loop

1. Entry Controlled Loop: In the entry-controlled loop the control conditions are tested before the start of the loop execution. If the conditions are not satisfied, then the body of the loop will not be executed. It is also known as ***pretest loop or top test loop***. The flow chart for the entry controlled loop is as illustrated in fig .

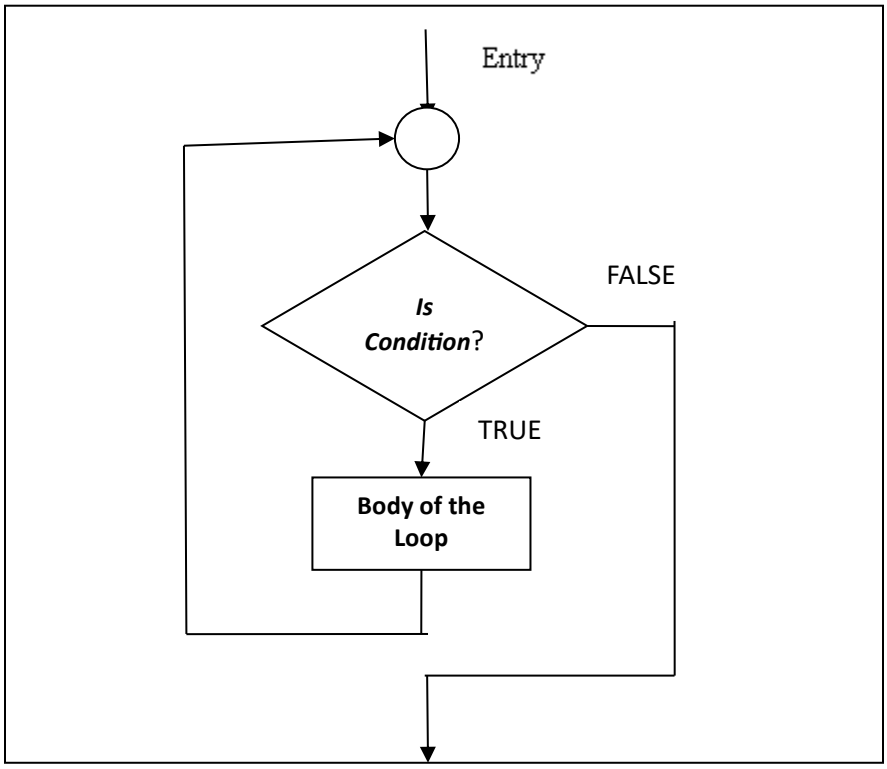


Fig: Flow diagram for Entry Controlled Loop

2.Exit Controlled Loop: In the exit controlled loop the test is performed at the end of the body of the loop and therefore the body is executed unconditionally for the first time. It is also known as ***posttest loop***. Here the body of the loop will get executed at least once before testing.The flow chart for the exit controlled loop is as illustrated in fig .

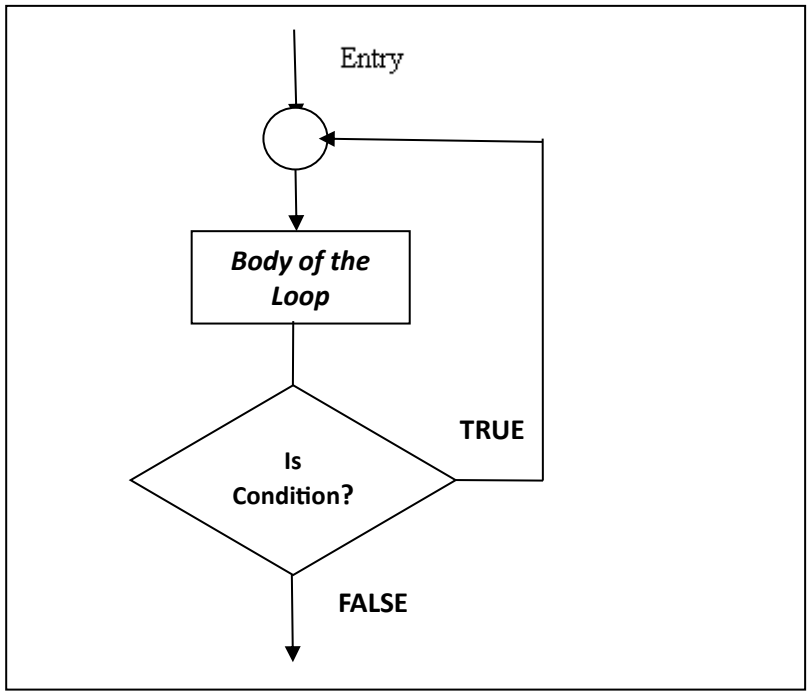


Fig: Flow diagram for Exit Controlled Loop

The looping includes the following four steps:

1. Initialization of a condition variable
2. Testing for a specified value of the condition variable for execution of the loop.
3. Execution of the statements in the loop
4. Updating (Incrementing or Decrementing) the condition variable

Types of Loops Supported in Python: The Python Language supports the following two looping operations:

1. The while statement
2. The for statement

The while statement: It is an entry controlled loop statement. In case of while loop the initialization, testing the condition and incrementation or updation is done in separate statements. First initialization of the loop counter is performed.Next the condition is checked. If the condition evaluates to be true then the control enters the body of the loop and executes the statements in the body.

1. The syntax or basic format of the while statement is as shown in figure below:

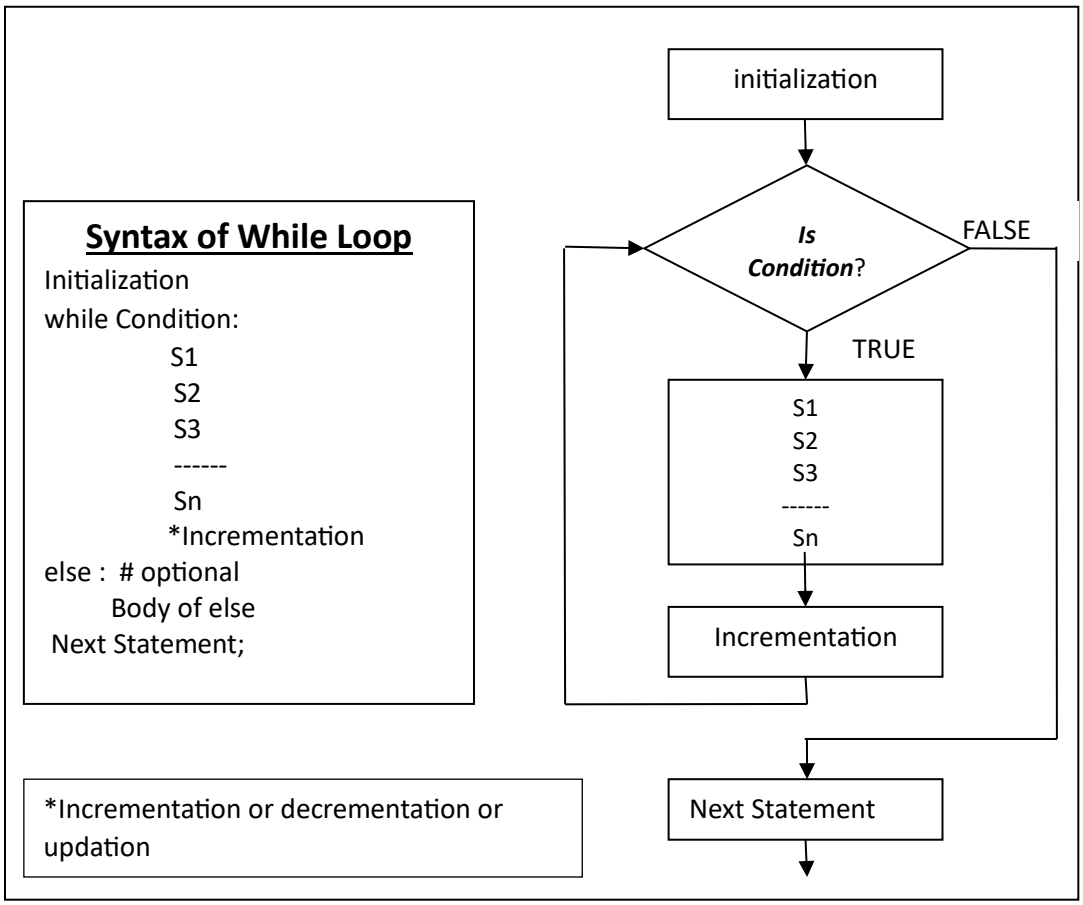


Fig: Flow diagram for Exit Controlled Loop

While statement in Python executes a block of code repeatedly as long as the test/control condition of the loop is true. The control condition of the while loop is executed before any statement in the body of the loop is executed . If the condition is true, the body of the loop is executed. Again, the control condition of the loop is tested, and the loo continue as long as the condition remains true. When the test outcome of this condition remains true. When the test outcome of this condition becomes false, the loop is not entered again and the control is transferred to the statement immediately following the body of the loop as shown in the flow diagram.

Example: Program to find the sum of n natural numbers using while loop.

```
n = int(input("Enter the number: "))
i=1
sum=0
while i<=n:
    sum = sum + i
    i = i+1
print("The sum of natural numbers = {}".format(sum))
```

```
Enter the number: 10
The sum of natural numbers = 55
```

For loop:

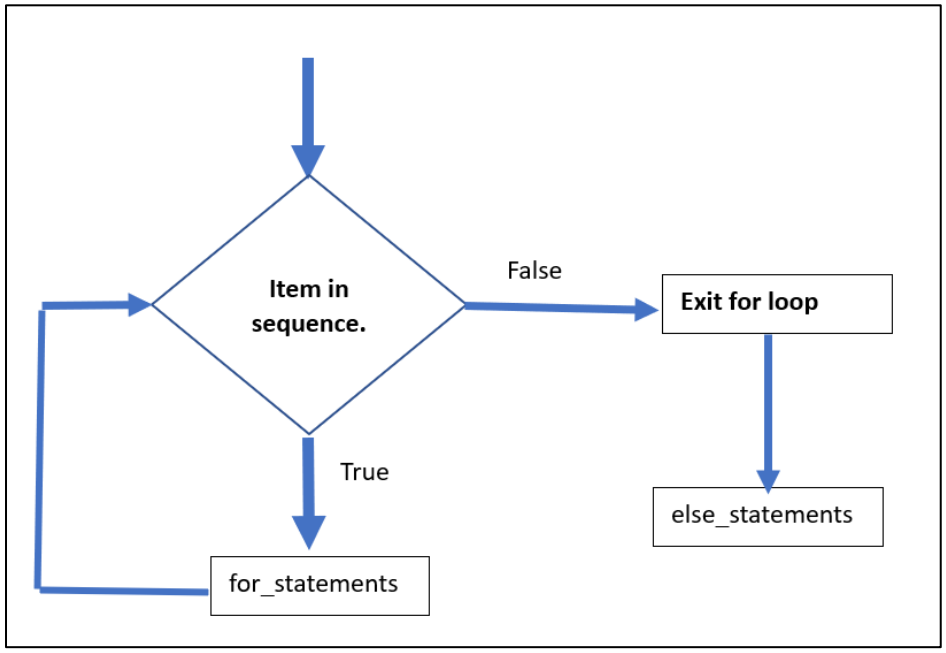
The for loop is another entry-controlled loop. It is used to execute the set of statements repeatedly over a range of values or a sequence. With every iteration of the loop, the control variable checks whether each of the values in the range has been traversed or not. When all the items in the range are traversed the control is then transferred to the statement immediately following for loop.

Syntax of for loop :

```
Syntax:

    for item in iterable:
       for_statements
    else: # optional
       else_statements
```

Flow Chart:



Example: Python Program to find the sum of natural numbers upto n.

```
1  n = int(input("Enter the value of n: "))
2  sum = 0
3  for i in range(1,n+1):
4      sum = sum + i
5  print("Sum of First n natural numbers :",sum)
```

```
Enter the value of n: 10
Sum of First n natural numbers : 55
```

range ( ) function: The range() function returns a sequence of numbers, starting from 0 to a specified number ,incrementing each time by 1.

Syntax :

range(start,step,stop)

Example:

```
x = range(3, 6)
for n in x:
      print(n)
```

```
3
4
5
```

```
x = range(3, 20, 2)
for n in x:
      print(n)
```

```
3
5
7
9
11
13
15
17
19
```

Table: range() examples

Command	Output
range(10)	[0,1,2,3,4,5,6,7,8,9]
range(1,11)	[1,2,3,4,5,6,7,8,9,10]
range(0,30,5)	[0,5,10,15,20,25]
range(0,-9,-1)	[0,-1,-2,-3,-4,-5,-6,-7,-8

The argument of **range()** function must be integers. The step parameter can be any positive or negative integer other than zero.

Infinite Loop: A loop becomes infinite loop if a condition never becomes FALSE. You must use caution when using while loops because of the possibility that this condition never resolves to a FALSE value. This results in a loop that never ends. Such a loop is called an infinite loop.
An infinite loop might be useful in client/server programming where the server needs to run continuously so that client programs can communicate with it as and when required.

```
var = 1
while var == 1 :  # This conditions results in an infinite loop
    num = input("Enter a number  :")
    print("You entered: ", num)
print("Good bye!")
```

```
Enter a number  :20
You entered:  20
Enter a number  :1
You entered:  1
Enter a number  :32
You entered:  32
Enter a number  :12
You entered:  12
Enter a number  :32
You entered:  32

Enter a number  :
```

Nested Loops:

Python programming language allows to use one loop inside another loop.

Syntax for nested for loop:

```
for iterating_var in sequence:
   for iterating_var in sequence:
      statements(s)
   statements(s)
```

Example:

```
for i in range(1,6):
    for j in range(1,i+1):
        print(j,end =' ')
    print()
```

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

Syntax for nested while loop:

The syntax for a **nested while loop** statement in Python programming language is as follows –

```
while expression:
   while expression:
      statement(s)
   statement(s)
```

A final note on loop nesting is that you can put any type of loop inside of any other type of loop. For example a for loop can be inside a while loop or vice versa. The following program uses a nested for loop to find the prime numbers from 2 to 20−

```
i = 2
while(i < 20):
    j = 2
    while(j <= (i/j)):
        if not(i%j): break
        j = j + 1
    if (j > i/j) : print(i, " is prime")
    i = i + 1
print("Good bye!")
```

```
2  is prime
3  is prime
5  is prime
7  is prime
11  is prime
13  is prime
17  is prime
19  is prime
Good bye!
```

Jump statements:

You might face a situation in which you need to exit a loop completely when an external condition is triggered or there may also be a situation when you want to skip a part of the loop and start next execution.

Python provides **break** and **continue** statements to handle such situations and to have good control on your loop.

The **break** statement in Python terminates the current loop and resumes execution at the next statement, just like the traditional break found in C.

The most common use for break is when some external condition is triggered requiring a hasty exit from a loop. The **break** statement can be used in both *while* and *for* loops.

```
1  for letter in 'Python':     # First Example
2      if letter == 'h':
3            break
4      print('Current Letter :', letter)
5
6  var = 10                    # Second Example
7  while var > 0:
8      print('Current variable value :', var)
9      var = var -1
10     if var == 5:
11         break
12 print("Good bye!")
```

```
Current Letter : P
Current Letter : y
Current Letter : t
Current variable value : 10
Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
Good bye!
```

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.

The **continue** statement can be used in both *while* and *for* loops.

Example:

```
for letter in 'Python':     # First Example
    if letter == 'h':
        continue
    print('Current Letter :', letter)

var = 5                    # Second Example
while var > 0:
    var = var -1
    if var == 5:
        continue
    print('Current variable value :', var)
print("Good bye!")
```

```
Current Letter : P
Current Letter : y
Current Letter : t
Current Letter : o
Current Letter : n
Current variable value : 4
Current variable value : 3
Current variable value : 2
Current variable value : 1
Current variable value : 0
Good bye!
```

else statement used with loops :

Python supports to have an **else** statement associated with a loop statements.

- If the **else** statement is used with a **for** loop, the **else** statement is executed when the loop has exhausted iterating the list.
- If the **else** statement is used with a **while** loop, the **else** statement is executed when the condition becomes false.

Example :

The following example illustrates the combination of an else statement with a for statement that searches for prime numbers from 10 through 20.

```
for num in range(10,20):  #to iterate between 10 to 20
    for i in range(2,num): #to iterate on the factors of the number
        if num%i == 0:      #to determine the first factor
            j=num/i #to calculate the second factor
            print('%d equals %d * %d' % (num,i,j))
            break #to move to the next number, the #first FOR
    else:         # else part of the loop
        print(num, 'is a prime number')
```

```
10 equals 2 * 5
11 is a prime number
12 equals 2 * 6
13 is a prime number
14 equals 2 * 7
15 equals 3 * 5
16 equals 2 * 8
17 is a prime number
18 equals 2 * 9
19 is a prime number
```

Similar way you can use **else** statement with **while** loop.

The Pass Statement :

The **pass** statement in Python is used when a statement is required syntactically but you do not want any command or code to execute.The **pass** statement is a *null* operation; nothing happens when it executes. The **pass** is also useful in places where your code will eventually go, but has not been written yet (e.g., in stubs for example):

```
for letter in 'Python':
    if letter == 'h':
          pass
          print('This is pass block')
    print('Current Letter :', letter)

print("Good bye!")
```

```
Current Letter : P
Current Letter : y
Current Letter : t
This is pass block
Current Letter : h
Current Letter : o
Current Letter : n
Good bye!
```

Importing Modules

Python function definition can, usefully, be stored in one or more separate files for easier maintenance and to allow them to beused in several programs without copying the definitions into each one. Each file storing function definitions is called a “module” and the module name is the file name with “**.py**” extension.

Example:

Functions stored in the module are made available to a program using the Python import keyword followed by the module name. Although not essential, it is customary to put any import statements at the beginning of the program.

Imported functions can be called using their name dot -suffixed after the module name. For example, a “**f1()**” function from an imported module named “userdefined” can be called with ***userdefined.f1()*** .

Design a new Python module called userdefined by defining all functions as illustrated below. Save the file as userdefined.py and run the file.

userdefined.py

```
1  def f1():
2      print("I am in f1")
3
4  def f2():
5      print("I am in f2")
6
7  def f3():
8      print("I am in f3")
9
10 def n5(n):
11     return n**5
12
13 def n6(n):
14     return n**6
```

Start a new script with name **importexample.py** (/ipynb). Next call each function with and without arguments as per the requirements. Run the program and get the output as illustrated below:

Importing Modules:

```
import userdefined
userdefined.f1()
userdefined.f2()
userdefined.f3()
x = userdefined.n5(2)
print(x)
y = userdefined.n6(3)
print(y)
```

```
I am in f1
I am in f2
I am in f3
32
729
```

Importing the sys and keyword modules :

Python includes "**sys**" and "keyword" modules that are useful for interrogating the Python system itself. The keyword module contains a list of all Python keywords in its ***kwlist*** attribute and provides as ***iskeyword ()*** method if you want to rest a word.

```
import sys,keyword
```

```
print("Python Version:",sys.version)
```

```
Python Version: 3.7.3 (default, Mar 27 2019, 17:13:21) [MSC v.1915 64 bit (AMD6
4)]
```

```
print("Python Interpreter Location:",sys.executable)
```

```
Python Interpreter Location: C:\Users\thyagaraj\Anaconda3\python.exe
```

```
print('Python Module Search Path:')
for dir in sys.path:
    print(dir)
```

```
Python Module Search Path:
C:\Users\thyagaraj\V SEM 2020
C:\Users\thyagaraj\Anaconda3\python37.zip
C:\Users\thyagaraj\Anaconda3\DLLs
C:\Users\thyagaraj\Anaconda3\lib
C:\Users\thyagaraj\Anaconda3
C:\Users\thyagaraj\AppData\Roaming\Python\Python37\site-packages
C:\Users\thyagaraj\Anaconda3\lib\site-packages
C:\Users\thyagaraj\Anaconda3\lib\site-packages\web.py-0.40.dev1-py3.7.egg
C:\Users\thyagaraj\Anaconda3\lib\site-packages\win32
C:\Users\thyagaraj\Anaconda3\lib\site-packages\win32\lib
C:\Users\thyagaraj\Anaconda3\lib\site-packages\Pythonwin
C:\Users\thyagaraj\Anaconda3\lib\site-packages\IPython\extensions
C:\Users\thyagaraj\.ipython
```

```
# To display the list of all the Python Keywords
print('Python Keywords:')
for word in keyword.kwlist:
    print(word ,',',end=' ')
```

```
Python Keywords:
False , None , True , and , as , assert , async , await , break , class , continue , def , del , elif , else , except , finally , for , from , global , if , import , in , is , lambda , nonlocal , not , or , pass , raise , return , try , while , with , yield ,
```

Performing Mathematics

```
import math
```

```
print("2 power 3",math.pow(2,3))
print("Rounded up 3.2",math.ceil(3.2))
print("Rounded Down3.7",math.floor(3.7))
print("Square root of 16",math.sqrt(16))
print("Sin of 30", math.sin(30))
print("Cosin of 45",math.cos(45))
print("tan of 60", math.tan(60))

```

```
2 power 3 8.0
Rounded up 3.2 4
Rounded Down3.7 3
Square root of 16 4.0
Sin of 30 -0.9880316240928618
Cosin of 45 0.5253219888177297
tan of 60 0.320040389379563
```

Random

```
import random
```

```
nums = random.sample(range(1,49),6)
print("Your Lucky Lotto Number Are:",nums)
```

```
Your Lucky Lotto Number Are: [38, 44, 19, 48, 31, 13]
```

```
random.seed(10)
print(random.random())
```

```
0.5714025946899135
```

```
# Returns a randomly selected element from the range created by the start,
# stop and step arguments.
print(random.randrange(3,9,2))
```

```
7
```

```python
import random
for i in range(5):
    print(random.randint(1, 10),end=' ')
```

```
1 7 3 10 6
```

Ending a Program Early with sys.exit ()

```python
import sys
while True:
    print('Type exit to exit.')
    response = input()
    if response == 'exit':
        sys.exit()
    print('You typed ' + response + '.')
```

```
Type exit to exit.
one
You typed one.
Type exit to exit.
two
You typed two .
Type exit to exit.
exit

An exception has occurred, use %tb to see the full traceback.

SystemExit
```

Questions for Practice:

1. What is Flow Chart ? Give the meaning of different flow chart symbols.

2. Write a Flow chart and Python program for the following:

a. To find the average of two numbers.
b. To find the simple interest given the value of P,T and R
c. To find the maximum of two numbers
d. To find the sum of first 50 natural numbers
e. To find the factorial of a given number N.

3. Compare == and = operator.

4. What are operators? Explain the following Operators with example:

a. Binary Boolean Operators: and, or & not

5. What is Flow Control? Explain the different Elements of Flow Control?

6. Define Block and explain what nested blocks with example are.

7. Define condition control statement. Explain the different types of Condition Control Statement.

8. Explain with flow chart and programming example , the following condition control statements :

a. if
b. if – else
c. nested if else
d. if – elif ladder

9. What is iteration or looping ? Describe the different types of looping statements.

10. Explain with syntax , flow chart and programming example the following looping operations.

a. While loop
b. For loop

11. Discuss the working of range() function with programming example.

12. Give the output of the following :

a. range(10)

b. range(1,11)

c. range(0,30,5)

d. range(0,-9,-1)

13. What is infinite loop ? Explain with example.

14. What are nested Loops ? Explain with examples.

15. Discuss the following with examples :

a. break

b. continue

c. else statement with loop

d. pass

16. What are Python Modules ? Explain with examples how to import Python Modules .

17. What is the difference between break and continue statements.

18. What is the purpose of else in loop?

19. Write logical expressions for the following :

a. Either A is greater than B or A is less than C

b. Name is Snehith and age is between 18 and 35.

c. Place is either Mysore or Bengaluru but not "Dharwad".

20. *Convert the following while loop into for loop :*

```
x =10
while (x<20):
    print(x+10)
    x+=2
```

21. *Explain while and for loop . Write a program to generate Fibonacci series upto the given limit by defining FIBONACCI(n) function.*

22. *Mention the advantage of continue statement. Write a program to compute only even numbers sum within the given natural number using continue statement.*

23. *Demonstrate the use of break and continue keywords in looping structures using a snippet code.*

24. *With syntax explain the finite and infinite looping constructs in python. What is the break and continue statements .[VTU June /July 2019]*

Programming Questions for Practice:

1.Construct a logical expressions to represent each of the following conditions :

a. Mark is greater than or equal to 100 but less than 70
b. Num is between 0 and 5 but not equal to 2
c. Answer is either 'N' or 'n'
d. Age is greater than or equal to 18 and gender is male
e. City is either 'Kolkata' or 'Mumbai'

2.Write a program to check if the number of positive or negative and display an appropriate message.

3.Write a program to convert temperature in Fahrenheit to Celsius.

4.Write a program to display even numbers between 10 and 20.

5.Write a program to perform all the mathematical operations of calculator.

6.Write a program to accept a number and display the factorial of that number.

7.Write a program to convert binary number to decimal number.

8.Write a program to find the sum of the digits of a number.

9.Write a program to display prime number between 30.

10. Write a program to find the best of two test average marks out of three tests marks accepted from the user.

11. Write a program to find the largest of three numbers . [VTU June /July 2019]

12. Write a program to check whether the given year is leap or not. [VTU June /July 2019]

13. Write a program to generate and print prime number between 2 to 50.

14. Write a program to find those numbers which are divisible by 7 and multiple of 5 , between 1000 and 3000.

15. Write a program to guess a number between 1 to 10.

16. Write a program that accepts a word from the user and reverse it.

17. Write a program to count the number of even and odd numbers from a series of numbers.

18. Write a program that prints all the numbers from 0 to 10 except 3, 7 and 10.

19. Write a program to generate Fibonacci series between 0 and 50.

20. Write a program which iterates the integers from 1 to 50. For multiple of three print “Fizz” instead of the numbers and fro the multiples of five print “Buzz” . For numbers which are multiples of both three and five print “FizzBuzz”.

21. Write a program that accepts a string and calculate the number of digits and letters.

22. Write a program to check the validity of password input by users.

23. Write a program to find the numbers between 100 and 400 where each digit of a number is an even number . The numbers obtained should be printed in a comma-separated sequence.

24. Write a program to print alphabet patterns ‘A’ ,D, ‘E’, ‘G’,’L’, ‘T’ and ‘S’ with * symbol.

25. Write a python program to create the multiplication table (from 1 to 20) of a number.

26. Write a program to print the following patterns :

```
*
* *
* * *
* * * *
* * * * *
* * * *
* * *
* *
*
```

```
1
22
333
4444
55555
666666
7777777
88888888
999999999
```

```
*
* *
* * *
* * * *
* * * * *
```

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

```
A
B B
C CC
D DDD
E EEEE
```

```
* * * * *
* * * *
* * *
* *
*
```

```
1 2 3 4 5
1 2 3 4
1 2 3
1 2
1
```

```
        *
      * * *
    * * * * *
  * * * * * * *
* * * * * * * * *
```

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

```
* * * * * * * * *
  * * * * * * *
    * * * * *
      * * *
        *
```

```
          1
        1   1
      1   2   1
    1   3   3    1
   1  4    6   4   1
  1  5   10   10  5   1
```

```
1
2 3
4 5 6
7 8 9 10
```

1.3 Functions

1.3.1 Introduction

A function is a group (or block) of statements that perform a specific task . Functions run only when it is called. One can pass data into the function in the form of parameters. Function can also return data as a result. Instead of writing a large program as one long sequence of instructions , it can be written as several small function , each performing a specific part of the task . They constitute line of code(s) that are executed sequentially from top to bottom by Python interpreter. A python function is written once and is used / called as many time as required . Functions are the most important building blocks for any application in Python and work on the divide and conquer approach. Functions can be conquered into the following three types :

(i) User Defined
(ii) Built in
(iii) Modules

i.User Defined Functions:

In Python, user-defined functions are functions that are created by the programmer to perform specific tasks. These functions are defined using the **def** keyword followed by the function name, parentheses for optional parameters, and a colon to start the function block. The syntax for defining and calling a function is as illustrated below :

Syntax for Defining a function:

Function is defined using def keyword in Python.

```
def fun_name(comma_seprated_parameter_ list):
        stmt_1
        -------
        stmt_n
        return stmt
```

Statements below def begin with four spaces. This is called indentation. It is a requirement of Python that the code following a colon must be indented. A function definition consists of the following components;

1. Keyword **def** marks the start of function header
2. A function name to uniquely identify it. Function naming follows the same rules as rules of writing identifiers in Python.
3. Parameters (arguments) through which we pass values to a function . They are optional.
4. A colon (:) to mark the end of function header.
5. Optional documentation string (docstring) to describe what the function does.
6. One or more valid Python statements that make up the function body. Statements must have same indentation level (usually) 4 spaces)
7. An optional return statement to return a value from the function .

Example :

```
def  cube(n):
     ncube = n**3
     return  ncube
```

Syntax for calling a function:

fun_name(parameter list)

Example: cube(3)

```
def cube(n):
    ncube = n**3
    return ncube
```

```
cube(10)
```

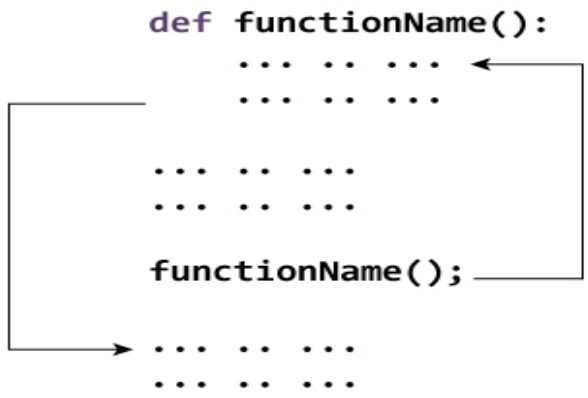
```
1000
```

Parameters and arguments

Parameters are temporary variable names within functions. The argument can be thought of as the value that is assigned to that temporary variable.

- '**n**' here is the ***parameter*** for the function '**cube**'. This means that anywhere we see '**n**' within the function will act as a placeholder until number is passed an argument.
- Here **3** is the ***argument***.
- Parameters are used in ***function definition*** and arguments are used in ***function call***.

Working of function



Example 1: Function without parameters

```
# Function without parameters

# Function definition
def  display():
    print("Rocky Robin")
    print("Lucknow")
    print("India")
# Function Call
display()
```

```
Rocky Robin
Lucknow
India
```

Example 2 : Function with parameters but without returning values.

```
# function with paramters and without return values
def Simple_Interest( p,t,r):
    si = p*t*r/100
    print("Simple Interest: ",si)

p = float(input("Enter the value of Principal Amount in Rupees: "))
t = float(input("Enter the value of Time Period in years: "))
r = float(input("Enter the value of Rate of Interest in percentage: "))
Simple_Interest(p,t,r)
```

```
Enter the value of Principal Amount in Rupees: 1000
Enter the value of Time Period in years: 2.5
Enter the value of Rate of Interest in percentage: 1.5
Simple Interest:  37.5
```

Example 3: Function with parameters and return values

```
# function with parameter and return value
def area_rect(length,breadth):
    area = length * breadth
    return area

length = float(input("Enter the length of rectangle: "))
breadth =float(input("Enter the breadth of rectangle: "))
area = area_rect(length,breadth)
print ("The area of rectangle :",area)
```

```
Enter the length of rectangle: 10
Enter the breadth of rectangle: 20
The area of rectangle : 200.0
```

Example 4: Function which return multiple values

```
def multireturn(n):
    return n**2,n**3,n**4,n**5  # Returning four values
```

```
n = int(input("Enter the number: "))
np2,np3,np4,np5 = multireturn(n)
print("np2 : ",np2)
print("np3 : ",np3)
print("np4 : ",np4)
print("np5 : ",np5)
```

```
Enter the number: 2
np2 :  4
np3 :  8
np4 :  16
np5 :  32
```

The None-Value

In Python there is a value called None, which represents the absence of a value. None is the only value of the None Type data type. (Other programming languages might call this value null, nil, or undefined.) Just like the Boolean True and False values, None must be typed with a capital N.

```
def f1():
    print(1)

x= f1()
y = print("End")
print(x)
print(y)
```

```
1
End
None
None
```

Keyword Arguments and print()

Keyword arguments are identified by the keyword put before them in the function call. Keyword arguments are often used for optional parameters. For example, the print( ) function has the optional parameters ***end*** and ***sep*** to specify what should be printed at the end of its arguments and between its arguments (separating them), respectively. Following examples illustrates the behavior of print with ***end*** , without ***end*** and with ***sep***.

```
1 print('Hello', end=' ')
2 print('World')
```

```
Hello World
```

```
1 print('Hello')
2 print('World')
```

```
Hello
World
```

```
1 print('Hello', end='##')
2 print('World')
```

```
Hello##World
```

```
1 print('cats','dogs','mice')
```

```
cats dogs mice
```

```
1 print('cats', 'dogs', 'mice', sep='>')
```

```
cats>dogs>mice
```

1.3.6 Local and Global Scope

Parameters and variables that are assigned in a called function are said to exist in that function's local scope. Variables that are assigned outside all functions are said to exist in the global scope. A variable that exists in a local scope is called a local variable, while a variable that exists in the global scope is called a global variable. A variable must be one or the other; it cannot be both local and global. Following example illustrates the difference between local and global variable .

```
1  x = 20  # global variable
2  def f1():
3      x = 15
4      l = 20 # local variable
5      print("Function f1",x,l)
6
7  def f2():
8      y = x + 20
9      print("Function f2",x,y)
```

```
1  f1()
2  f2()
3  print(x)
```

```
Function f1 15 20
Function f2 20 40
20
```

Local variable cannot be used in the global scope

Consider this program which will cause an error when you run it:

```
def fun1():
    x =31337
fun1()
print(x)
```

If you run this program the output will look like this

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-12-8e05a42defd6> in <module>
      2     x =31337
      3 fun1()
----> 4 print(x)

NameError: name 'x' is not defined
```

The error happens because the x variable exists only in the local scope created when fun1() is called. Once the program execution returns from fun1(), that local scope is destroyed, and there is no longer a variable named x. So when your program tries to run print(x), Python gives you an error saying that x is not defined. This makes sense if you think about it; when the program execution is in the global scope, no local scopes exist, so there can't be any local variables. This is why only global variables can be used in the global scope.

Local Scopes Cannot Use Variables in Other Local Scopes

A new local scope is created whenever a function is called, including when a function is called from another function. Consider this program:

```
def fun1():
    x =10
    fun2()
    print(x)
def fun2():
    y = 21
    x = 0
fun1()
```

```
10
```

When the program starts the func1() is called and a local scope is created . The local variable x is set to 10. Then fun2() is called and a second local scope is created . Multiple local scopes can exist at the same time . In this new local scope , the local variable y is set to 21 and a local variable x which is different from the one in fun1()'s local scope is also created and set to 0. When fun2() returns the local scope for the call to fun1() still exists here the x variable is set to 10. This is what the programs prints.

The local variables in one function are completely separate the local variable in another function.

Global Variable Can be read from a local scope :

Consider the following program :

```
def fun1():
    print(x)
x = 42
fun1()
print(x)
```

```
42
42
```

Since there is no parameter named x or any code that assigns x a value in the fun1() function , when x is used in fun1(), Python considers it a reference to the global variable x. This is why 42 is printed when the previous program is run.

Local and Global Variables with the same Name :

One should avoid using local variables that have the same name as a global variable or another local variable. Consider the following program :

```
def fun1():
    x = 'I am local to fun1'
    print(x)
def fun2():
    x = 'I am local to fun2'
    print(x)
    fun1()
    print(x)
x = 'I am global'
fun2()
print(x)
```

When you run the program , it outputs the following :

```
I am local to fun2
I am local to fun1
I am local to fun2
I am global
```

There are actually three different variables in this program, but confusingly they are all named x.

A variable named x that exists in a local scope when fun1() is called.

A variable named x that exists in a local scope when fun2() is called

A variable named x that exists in the global scope

Since these three separate variables all have the same name, it can be confusing to keep of which one is being used at any given time . This is why you should avoid using the same variable name in different scopes.

Global Statement

If you need to modify a global variable from within a function , used the global statement . If you have a line such as global x at the top of a function , it tells Python, In this function, x refers to the global variable, so don't create a local variable with this name." For example, type the following code and run

```
def fun1():
    global x
    x ='spam'
x = "global"
fun1()
print(x)
```

When you run this program the final print() call will output this :

Because x is declared global at the top of spam() , when x is set to 'spam' , this assignment is done to the globally scoped x. No local x variable is created.

There are four rules to tell whether a variable is in a local scope or global scope:

1. If a variable is being used in the global scope (that is, outside of all functions), then it is always a global variable.

2. If there is a global statement for that variable in a function, it is a global variable.

3. Otherwise, if the variable is used in an assignment statement in the function, it is a local variable.

4. But if the variable is not used in an assignment statement, it is a global variable.

Exceptional Handling

Right now, getting an error, or exception, in your Python program means the entire program will crash. You don't want this to happen in real-world programs. Instead, you want the program to detect errors, handle them, and then continue to run. For example, consider the following program, which has a "divide-byzero" error. Open a new file editor window and enter the following code, saving it as zeroDivide.py:

```
def spam(divideBy):
    return 42 / divideBy
```

```
print(spam(2))
print(spam(12))
print(spam(0))
print(spam(1))
```

We've defined a function called spam, given it a parameter, and then printed the value of that function with various parameters to see what happens. This is the output you get when you run the previous code:

```
21.0
3.5

---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-8-108f30dfb85b> in <module>
      1 print(spam(2))
      2 print(spam(12))
----> 3 print(spam(0))
      4 print(spam(1))

<ipython-input-7-6657c085a5bf> in spam(divideBy)
      1 def spam(divideBy):
----> 2     return 42 / divideBy

ZeroDivisionError: division by zero
```

A ZeroDivisionError happens whenever you try to divide a number by zero. From the line number given in the error message, you know that the return statement in spam() is causing an error.

Try and Except :

Errors can be handled with try and except statements. The code that could potentially have an error is put in a try clause. The program execution moves to the start of a following except clause if an error happens. You can put the previous divide-by-zero code in a try clause and have an except clause contain code to handle what happens when this error occurs.

```
def spam(divideBy):
    try:
        return 42 / divideBy
    except ZeroDivisionError:
         print('Error: Invalid argument.')
```

```
print(spam(2))
print(spam(12))
print(spam(0))
print(spam(1))
```

When code in a try clause causes an error, the program execution immediately moves to the code in the except clause. After running that code, the execution continues as normal. The output of the previous program is as follows:

```
21.0
3.5
Error: Invalid argument.
None
42.0
```

Note that any errors that occur in function calls in a try block will also be caught. Consider the following program, which instead has the spam() calls in the try block:

```
# Consider the following program, which instead has the spam()
# calls in the try block:
def spam(divideBy):
    return 42 / divideBy
try:
    print(spam(2))
    print(spam(12))
    print(spam(0))
    print(spam(1))
except ZeroDivisionError:
    print('Error: Invalid argument.')
```

When this program is run, the output looks like this:

```
21.0
3.5
Error: Invalid argument.
```

The reason print(spam(1)) is never executed is because once the execution jumps to the code in the except clause, it does not return to the try clause. Instead, it just continues moving down as normal.

ii. Built in functions

Built in functions are the predefined functions that are already available in python. Functions provide efficiency and structure to a programming language .python has many useful built in functions to make programming easier , faster and more powerful.

Some of the important builtin functions are listed below:

abs() Returns the absolute value of a number

ascii() Returns a readable version of an object. Replaces none-ascii characters with escape character

bin() Returns the binary version of a number

bool() Returns the boolean value of the specified object

bytearray() Returns an array of bytes

bytes() Returns a bytes object

chr() Returns a character from the specified Unicode code.

classmethod() Converts a method into a class method

complex() Returns a complex number

delattr() Deletes the specified attribute (property or method) from the specified object

dict() Returns a dictionary (Array)

dir() Returns a list of the specified object's properties and methods

divmod() Returns the quotient and the remainder when argument1 is divided by argument2

enumerate() Takes a collection (e.g. a tuple) and returns it as an enumerate object

eval() Evaluates and executes an expression

filter() Use a filter function to exclude items in an iterable object

float() Returns a floating point number

format() Formats a specified value

getattr() Returns the value of the specified attribute (property or method)

globals() Returns the current global symbol table as a dictionary

hex() Converts a number into a hexadecimal value

id() Returns the id of an object

input() Allowing user input

int() Returns an integer number

isinstance() Returns True if a specified object is an instance of a specified object

issubclass() Returns True if a specified class is a subclass of a specified object

iter() Returns an iterator object

len() Returns the length of an object

list() Returns a list

locals() Returns an updated dictionary of the current local symbol table

map() Returns the specified iterator with the specified function applied to each item

max() Returns the largest item in an iterable

min() Returns the smallest item in an iterable

next() Returns the next item in an iterable

object() Returns a new object

oct() Converts a number into an octal

open() Opens a file and returns a file object

ord() Convert an integer representing the Unicode of the specified character

pow() Returns the value of x to the power of y

print() Prints to the standard output device

property() Gets, sets, deletes a property

range() Returns a sequence of numbers, starting from 0 and increments by 1 (by default)

round() Rounds a numbers

set() Returns a new set object

setattr() Sets an attribute (property/method) of an object

slice() Returns a slice object

sorted() Returns a sorted list

str() Returns a string object

sum() Sums the items of an iterator

tuple() Returns a tuple

type() Returns the type of an object

vars() Returns the __dict__ property of an object

zip() Returns an iterator, from two or more iterators

iii. Modules :

As the programs become more lengthy and complex, there arises a need for the tasks to be split into smaller segments called modules.

A module is a file containing functions and variable defined in separate files. A module is simply a file that contains Python code or a series of instructions . When we break a program into modules , each module should contain functions that performs related tasks . There are some commonly used modules in Python that are used for certain predefined tasks and they are called libraries.

Modules also make it easier to reuse the same code in more than one program. If we have written a ste of functions that is needed in several different programs , we can place those functions that is needed in several different programs, we can place those functions in a module. Then we can import the module in each program that needs to call one of the functions . Once we import a module we can refer to any of its functions or variable in our program.

A module in Python is a file that contains a collection of related functions.

Importing Modules in a python Program

Python language provides two important methods to import modules in a program which are as follows :

(i) Import statement : To import entire module
(ii) From :To import all functions or selected ones
(iii) Import : To use modeule in a program , we import them using the import statement.

Syntax : import modulename1 [modulname2,------]

It is the simplest and the most common way to use modules in our code.\

Example:

import math

On executing

```
# math module and functions
import math
# Returns the smallest integer that is greater than or equal to x
print("\n math.ceil(3.5): ",math.ceil(3.5))
# Returns the largest integer that is less than or equal to x
print("\n math.floor(3.7: ",math.floor(3.7))
# Returns the value of x^y , where x and  y are numeric expressions
print("\n math.pow(2,5) : ", math.pow(2,5))
# Returns the absolute value (Positive value) of the expression/number
print("\n math.fabs(-15): ",math.fabs(-15))
# Returns the sqare root of x
print("\n math.sqrt(81):  ", math.sqrt(81))
# Returns the base 10 ogarithm of x
print("\n math.log10(100): ",math.log10(100))
# Returns the cosine of x in radians
print("\n math.cos(90):  ",math.cos(90) )
# Returns the sine of x in radians
print("\n math.sin(45): ", math.sin(45))
# Returns tangent of x in radians
print("\n math.tan(-30):", math.tan(-30))
print("\n\n")
# help is used to know the purpose of a function and how it is used.
print(help(math.cos),"\n")
```

Output

```
math.ceil(3.5):  4

math.floor(3.7:  3

math.pow(2,5) :  32.0

math.fabs(-15):  15.0

math.sqrt(81):   9.0

math.log10(100):  2.0

math.cos(90):   -0.4480736161291701

math.sin(45):  0.8509035245341184

math.tan(-30): 6.405331196646276


Help on built-in function cos in module math:

cos(x, /)
    Return the cosine of x (measured in radians).

None
```

Random module (Generating random numbers)

The various functions associated with the module are explained as follows:

randrange () : This method generates an integer between its lower and upper argument. By default, the lower argument is 0 and upper argument is 1. The following line of code generates random numbers from 0 to 29. In this instance it is 15.

```
import random
random.randrange(30)
```

```
15
```

Random() : This function generates a random number from 0 to 1 such as 0.564388 . This function can be used to generate random floating point values. It takes no parameters and returns values uniformly distributed between 0 and 1 (including 0 , but excluding 1).

```
from random import random
random()
```

```
0.5437173858981815
```

```
random()
```

```
0.16257180156868323
```

```
random()
```

```
0.14006004266226424
```

-Example :

```
import random
ran_number = random.random()
print("The random number os :",ran_number)
```

```
The random number os : 0.41512320186675467
```

Example :

```
ran_number = random.random()*5 + 10
print(" This random number generatd is :",ran_number)
```

```
 This random number generatd is : 14.313330696078674
```

Example:

```
# To calculate the sum of the digits of arandom three digit number
from random import random
n = random()*900 + 100
n = int(n)
print(n)
a = n//100
b= (n//10)%10
c = n%10
print(a+b+c)
```

```
963
18
```

Example:

```
# To generate a number between 0 and 9
import random
print(random.randint(0,9))
```

```
9
```

Example:

```
# To generate a random number between 0 and 5
import random
print(random.randint(0,5))
```

```
4
```

Example :

```
# Program that fills a list with numbers
from random import randint
def fill_list(lst,limit_num,low,high):
    for i in range(limit_num):
        lst.append(randint(low,high))
minimum = int(input("Min :  "))
maximum = int(input("MAx :  "))
n = int(input("Numbers limit : "))
a = []
fill_list(a,n,minimum,maximum)
print(a)
```

```
Min :  10
MAx :  50
Numbers limit : 5
[34, 15, 10, 39, 45]
```

Example :

```
import random
print("Uniform lotter number (1,100) :",random.uniform(1,100))
```

```
Uniform lotter number (1,100) : 52.43568337894924
```

Example :

```
import random
direction_choice = random.choice(["1: East",'2:West','3:North','4:South'])
print("My Direction is : ",direction_choice)
```

```
My Direction is :  3:North
```

Example :

```
import random
fruits =['Apple','Orange','Banana','Pineapple','Apricot']
random.shuffle(fruits)
print(random.shuffle(fruits))
```

```
None
```

Example :

```
import random
fruits =['Apple','Orange','Banana','Pineapple','Apricot']
random.shuffle(fruits)
print("Rehuffled Fruits:",fruits)
```

```
Rehuffled Fruits: ['Banana', 'Apricot', 'Pineapple', 'Apple', 'Orange']
```

Example :

```
random.shuffle(fruits)
print("Rehuffled Fruits:",fruits)
```

```
Rehuffled Fruits: ['Apricot', 'Apple', 'Orange', 'Pineapple', 'Banana']
```

Sample Programs

Example1: A Short Program , Guess the Number

```
# This is a guess the number game.
import random
secretNumber = random.randint(1, 20)
print('I am thinking of a number between 1 and 20.')
```

```
I am thinking of a number between 1 and 20.
```

```
# Ask the player to guess 6 times.
for guessesTaken in range(1, 7):
    print('Take a guess.')
    guess = int(input())
    if guess < secretNumber:
        print('Your guess is too low.')
    elif guess > secretNumber:
        print('Your guess is too high.')
    else:
        break # This condition is the correct guess!
if guess == secretNumber:
    print('Good job! You guessed my number in ' + str(guessesTaken) + ' guesses!'
else:
    print('Nope. The number I was thinking of was ' + str(secretNumber))
```

```
Take a guess.
8
Your guess is too low.
Take a guess.
17
Your guess is too high.
Take a guess.
13
Your guess is too low.
Take a guess.
15
Your guess is too high.
Take a guess.
14
Good job! You guessed my number in 5 guesses!
```

Simple Project:

```
def collatz(n):
    if n%2==0:
        print(n//2)
        return n//2
    elif n%2==1:
        print(3*n+1)
        return 3*n+1
```

```
while True:
    try:
        n = int(input("enter a number "))
    except ValueError:
            print("Invalid Argument")
    x = collatz(n)
    if x==1:
        break
```

```
enter a number 5
16
enter a number 3
10
enter a number 4
2
enter a number 2
1
```

Questions for Practice:

1. What are the advantages of using functions in a program?
2. When the code in the function does executes during definition or calling.
3. Discuss How to create a function with example.
4. Explain the following with example
    a. Functions
    b. Function Call
    c. Built in Function
    d. Type Conversion Functions
    e. Random Numbers
    f. Math Functions
    g. Adding new Functions
    h. Defining and using the new functions
    i. Flow of execution
    j. Parameter and Arguments
    k. Fruitful and void Functions
    l. Advantages of Functions
5. Differentiate between argument and parameter.
6. What is the difference between a function and a function call?
7. How many global scopes and local scopes are there in Python program?
8. What happens to variables in a local scope when the function call returns?
9. What is a return value? Can a return value be part of an expression?
10. What is the return value of the function which does not have return statement?
11. How can you force a variable in a function to refer to the global variable?
12. What is the data type of None?
13. What does the ***import allname*** statement do?
14. If you had a function named **radio()** in a module named **car** who would you call it after importing ***car***.
15. How can you prevent a program from crashing when it gets an error?
16. What goes in the try clause? What goes in the except clause?
17. Illustrate the flow of execution of a python function with an example program to convert given Celsius to Fahrenheit temperature.

18. Explain the function arguments in python.
19. Explain call by value and call by reference in python
20. Briefly explain about function prototypes
21. Define the scope and lifetime of a variable in python
22. Point out the uses of default arguments in python
23. Generalize the uses of python module.
24. Demonstrate how a function calls another function. Justify your Apply answer
25. List the syntax for function call with and without arguments.
26. Define recursive function
27. Define the syntax for passing arguments.
28. What are the two parts of function definition give the syntax
29. Briefly discuss in detail about function prototyping in python. With suitable example program
30. *Analyze the difference between local and global variables*.
31. *Explain with an example program to circulate the values of n variables*
32. Describe in detail about lambda functions or anonymous function.
33. Describe in detail about the rules to be followed while using Lambda function.
34. Explain with an example program to return the average of its argument
35. Explain the various features of functions in python.
36. Describe the syntax and rules involved in the return statement in python.
37. Write a program to demonstrate the flow of control after the return statement in python.
38. Formulate with an example program to pass the list arguments to a function.
39. Write a program to perform selection sort from a list of numbers using python.
40. Give the use of return () statement with a suitable example.
41. What are the advantages and disadvantages of recursion function? A
42. Explain the types of function arguments in python
43. Explain recursive function. How do recursive function works? Explain with a help of a program
44. Illustrate the concept of local and global variables.

45. A polygon can be represented by a list of (x, y) pairs where each pair is a tuple: [(x1, y1), (x2, y2), (x3, y3) , … (xn, yn)]. Write a Recursive function to compute the area of a polygon. This can be accomplished by "cutting off" a triangle, using the fact that a triangle with corners (x1, y1), (x2, y2), (x3, y3) has area (x1y1 + x2y2 + x3y2 – y1x2 –y2x3 – y3x1) / 2.

46. What is a lambda function?

47. How Do We Write A Function In Python?

48. What Is "Call By Value" In Python?

49. What Is "Call By Reference" In Python?

50. Is It Mandatory For A Python Function To Return A Value? Comment?

51. What Does The *Args Do In Python?

52. What Does The **Kwargs Do In Python?

53. Does Python Have A Main() Method?

54. What Is The Purpose Of "End" In Python?

55. What Does The Ord() Function Do In Python?

56. What are split(), sub(), and subn() methods in Python?

57. Describe the syntax for the following functions and explain with an example: a) abs() b) max() c) divmod() d) pow() e) len()

Programs for Practice:

1. Write a program to generate Fibonacci series upto the given limit FIBONACCI(n) function.
2. Write a single user defined function named 'Solve' that returns the Remainder and Quotient separately on the Console.
3. Write a program to find i) The largest of three numbers and ii) check whether the given year is leap year or not with functions.
4. Find the area and perimeter of a circle using functions. Prompt the user for input
5. Write a Python program using functions to find the value of nP_r and nC_r without using inbuilt factorial() function.
6. Write a program to find the product of two matrices.
7. A prime number is an integer greater than 1 that is evenly divisible by only 1 and itself. For example, the number 5 is prime because it can only be

evenly divided by 1 and 5. The number 6, however, is not prime because it can be divided by 1, 2, 3, and 6.

8. Write a function named isPrime, which takes an integer as an argument and returns True if the argument is a prime number, and False otherwise. Define a function main() and call isPrime() function in main() to display a list of the prime numbers from 100 to 500.

9. Write a program that lets the user perform arithmetic operations on two numbers. Your program must be menu driven, allowing the user to select the operation (+, -, *, or /) and input the numbers. Furthermore, your program must consist of following functions:

1. Function *showChoice*: This function shows the options to the user and explains how to enter data.
2. Function *add*: This function accepts two number as arguments and returns sum.
3. Function *subtract*: This function accepts two number as arguments and returns their difference.
4. Function *mulitiply*: This function accepts two number as arguments and returns product.
5. Function *divide*: This function accepts two number as arguments and returns quotient.
   Define a function main() and call functions in main().

10. Write a Python function for the following :

a. To find the Max of three numbers.
b. To sum all the numbers in a list
c. To multiply all the numbers in a list.
d. To reverse a string
e. To calculate the factorial of a number (a non-negative integer). The function accepts the number as an argument
f. To check whether a number is in a given range
g. That accepts a string and calculate the number of upper case letters and lower case letters.
h. That takes a list and returns a new list with unique elements of the first list.
i. That takes a number as a parameter and check the number is prime or not.
j. To print the even numbers from a given list
k. To check whether a number is perfect or not.
l. That checks whether a passed string is palindrome or not.

m. That prints out the first n rows of Pascal's triangle.
n. To check whether a string is a pangram or not.

11. Write a Python program that accepts a hyphen-separated sequence of words as input and prints the words in a hyphen-separated sequence after sorting them alphabetically.

12. Write a Python function to create and print a list where the values are square of numbers between 1 and 30 (both included).

13. Write a Python program to make a chain of function decorators (bold, italic, underline etc.) in Python.

14. Write a Python program to execute a string containing Python code

15. Write a Python program to access a function inside a function

16. Write a Python program to detect the number of local variables declared in a function.

17. Write a function calculation() such that it can accept two variables and calculate the addition and subtraction of it. And also it must return both addition and subtraction in a single return call

18. Create a function showEmployee() in such a way that it should accept employee name, and it's salary and display both, and if the salary is missing in function call it should show it as 9000

19. Create an inner function to calculate the addition in the following way

a. Create an outer function that will accept two parameters a and b
b. Create an inner function inside an outer function that will calculate the addition of a and b
c. At last, an outer function will add 5 into addition and return it

20. Write a recursive function to calculate the sum of numbers from 0 to 10

21. Write a function to calculate area and perimeter of a rectangle.

22. Write a function to calculate area and circumference of a circle.

23. Write a function to calculate power of a number raised to other. E.g.- ab.

24. Write a function to tell user if he/she is able to vote or not.(Consider minimum age of voting to be 18.)

25. Print multiplication table of 12 using recursion.

26. Write a function to calculate power of a number raised to other (ab) using recursion.

27. Write a function "perfect()" that determines if parameter number is a perfect number. Use this function in a program that determines and prints all the perfect numbers between 1 and 1000.[An integer number is said to

be “perfect number” if its factors, including 1(but not the number itself), sum to the number. E.g., 6 is a perfect number because 6=1+2+3].

28. Write a function to check if a number is even or not.

29. Write a function to check if a number is prime or not.

30. Write a function to find factorial of a number but also store the factorials calculated in a dictionary as done in the Fibonacci series example.

31. Write a function to calculate area and perimeter of a rectangle.

32. What is the output of the following Code ?

```
def foo(u, v, w=3):
        return u * v * w

def bar(x):
    y = 4
    return foo(x, y)

print(foo(4,5,6))
print(bar(10))
```

33. What is the output of the following Code ?

```
def change(t1,t2):
    t1 = [100,200,300]
    t2[0]= 8

list1 = [10,20,30]
list2 = [1,2,3]
change(list1, list2)
print(list1)
print(list2)
```

34. What is the output of the following Code ?

```
def dot(a, b):
    total = 0
    for i in range(len(a)):
        total += a[i] * b[i]
    print(total)

x = [10,20,30]
y = [1,2,3,4]
dot(x,y)
```

35. What is the output of the following Code ?

```
def buz(arr, n):
    for i in range(n-1):
        if arr[i]>arr[i+1]:
            arr[i], arr[i+1] = arr[i+1], arr[i]
    print(arr)

def bar(arr,n):
    for i in range(n-1):
        buz(arr,n-i)

t = [19,7,4,1]
bar(t,len(t))
```

36. What is the output of the following Code ?

```
def change(num1 ,num2=50):
    num1 = num1 + num2
    num2 = num1 - num2
    print(num1, '#', num2)

n1 = 150
n2 = 100
change(n1,n2)
change(n2)
change(num2=n1,num1=n2)
```