SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF ELECTRICAL AND ELECTRONICS

DEPARTMENT OF ELECTRICAL AND ELECTRONICS ENGINEERING

UNIT - 1

PYTHON – SBI1605
INTRODUCTION TO PYTHON

INTRODUCTION TO PYTHON

Program

A program performs a task in the computer. But, in order to be executed, a program must be written in the machine language of the processor of a computer. Unfortunately, it is extremely difficult for humans to read or write a machine language program. This is because a machine language is entirely made up of sequences of bits. However, high level languages are close to natural languages like English and only use familiar mathematical characters, operators and expressions. Hence, people prefer to write programs in high level languages like C, C++, Java, or Python. A high level program is translated into machine language by translators like compiler or interpreter.

ABOUT PYTHON

Python is a high level programming language that is translated by the python **interpreter**. As is known, an interpreter works by translating line-by-line and executing. It was developed by Guido-van-rossum in 1990, at the National Research Institute for Mathematics and Computer Science in Netherlands. Python doesn‟t refer to the snake but was named after the famous British comedy troupe, Monty Python‟s Flying Circus.

The following are some of the features of Python:

- Python is an Open Source: It is freely downloadable,

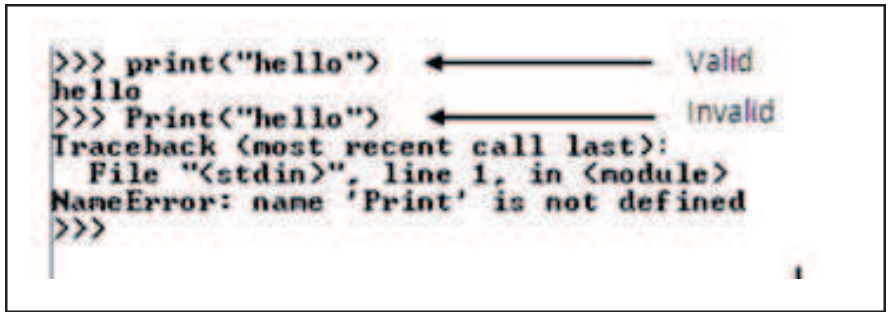
from the link “http:// python.org/”

- Python is portable: It runs on different operating systems / platforms
- Python has automatic memory management
- Python is flexible with both procedural oriented and object oriented programming
- Python is easy to learn, read and maintain

It is very flexible with the console program, Graphical User Interface (GUI) applications, Web related programs etc.

POINTS TO REMEMBER WHILE WRITING A PYTHON PROGRAM

- Case sensitive : Example - In case of print statement use only lower case and not upper case, (See the snippet below)



- Punctuation is not required at end of the statement
- In case of string use single or double quotes i.e. „ “ or “ ”
- Must use proper indentation: The screen shots given below show, how the value of “i” behaves with indentation and without indentation.

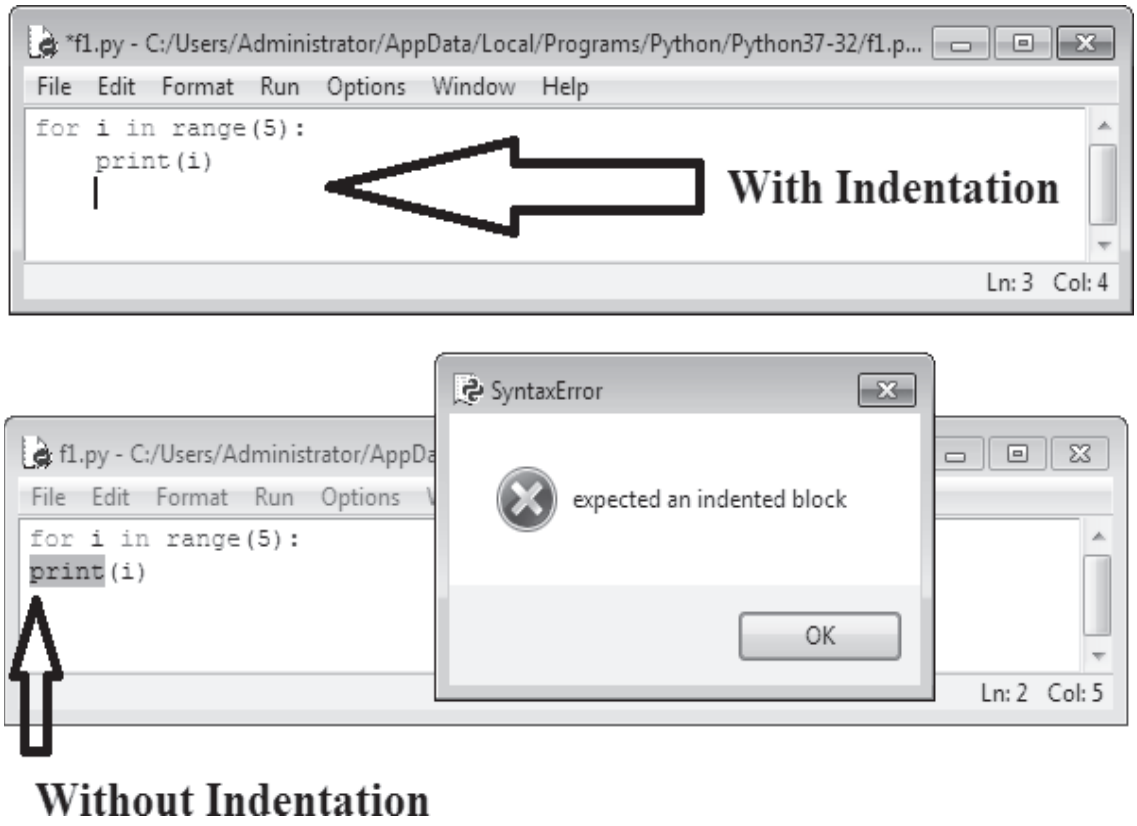


Fig 1.1 With and without Indentation

- Special characters like (,), # etc. are used
- ( ) -> Used in opening and closing parameters of functions
- # -> The Pound sign is used to comment a line

TWO MODES OF PYTHON PROGRAM

Python Program can be executed in two different modes:

- Interactive mode programming
- Script mode programming

(i) Interactive Mode Programming

It is a command line shell which gives immediate output for each statement, while keeping previously fed statements in active memory. This mode is used when a user wishes to run one single line or small block of code. It runs very quickly and gives instant output. A sample code is executed using interactive mode as below.

Interactive mode can also be opened using the following ways:

i) From command prompt c :> users\\...>python

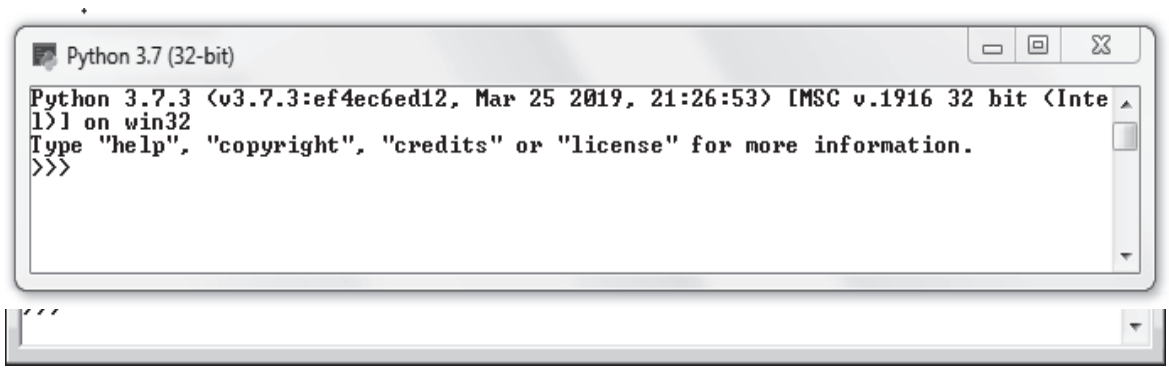


Fig 1.2 Command Prompt

The symbol ">>>" in the above screen indicates that the Python environment is in interactive mode.

ii) From the start menu select Python (As shown below)

Fig 1.3 Python in Start Menu

(ii) Script Mode Programming

When the programmer wishes to use more than one line of code or a block of code, script mode is preferred. The Script mode works the following way:

iii) Open the Script mode
iv) Type the complete program. Comment, edit if required.
v) Save the program with a valid name.
vi) Run
vii) Correct errors, if any, Save and Run until

proper output The above steps are described in detail

below:

i) To open script mode, select the menu "***IDLE (Python 3.7 32-bit)"*** from start menu

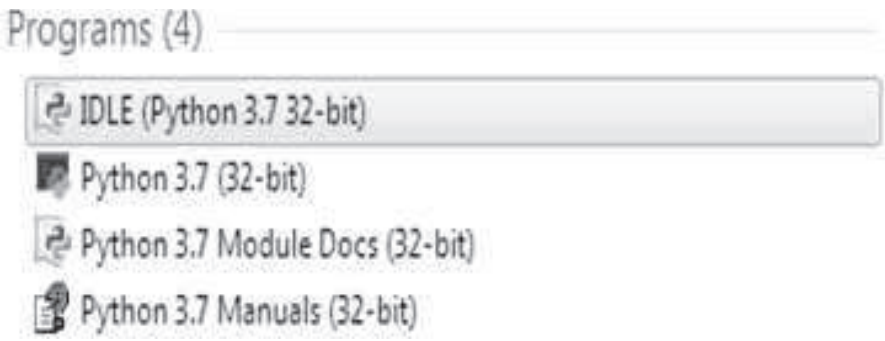


Fig 1.4 IDLE in Start Menu

ii) After clicking on the menu "***IDLE (Python 3.7 32-bit)"*** **,** a new window with the text Python 3.7.3 shell will be opened as shown below:

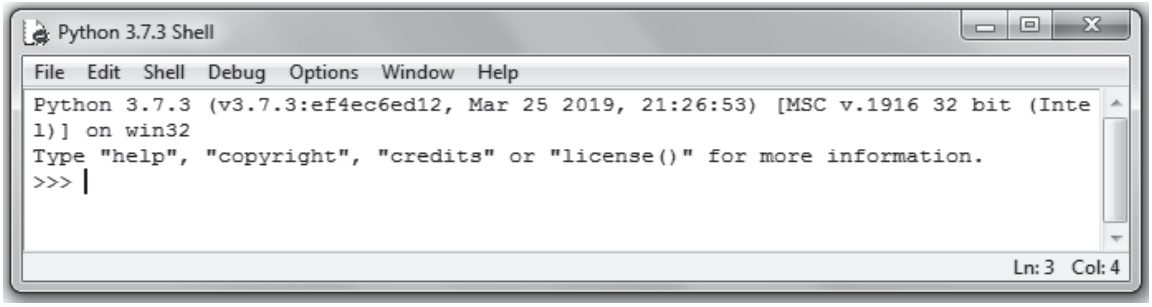


Fig 1.5 Python 3.7.3 Shell

iii) Select File New, to open editor. Type the complete program.

iv) Select File again; Choose Save.

This will automatically save the file with an extension ".py".

v) Select Run Run Module or Short Cut Key **F5** ***(As shown in the screen below)***

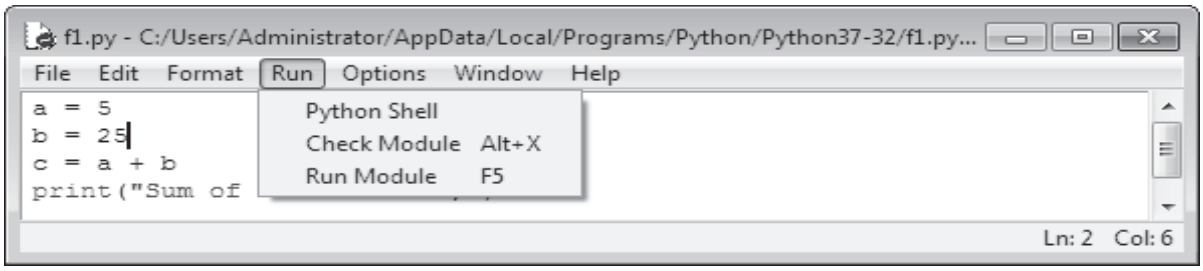


Fig 1.6 Run Module

The output of the program will be displayed as below:

```
>> Sum of a and b is: 30
```

References:

1. Hetland., “Beginning Python”, Apress, 2008
2. Mark Pilgrim, “Drive Into Python”, Apress, 2004
3. Martin C. Brown, “ Python: The Complete Reference (English)”, McGraw-Hill/Osborne Media, 2001.
4. Mark Summerfiled, “Programming in Python 3”, 2nd ed (PIP3), Addison Wesley.
5. https://www.academia.edu/41039821/Python_Tutorial_Release_3_7_0_Guido_van_Rossum_and_the_Python_development_team

UNIT - 1

PART – A

1. Define statement. List its types.
2. Write the pseudo code to calculate the sum and product of two numbers and display it.
3. Compare machine language, assembly language and high-level language.
4. What is meant by interpreter?
5. How will you invoke the python interpreter?
6. What is meant by interactive mode of the interpreter?
7. Write a snippet to display “Hello World” in python interpreter.
8. Define a variable and write down the rules for naming a variable.
9. Define keyword and enumerate some of the keywords in Python.
10. Define statement and what are its types?
11. What do you mean by an operand and an operator? Illustrate your answer with relevant example.
12. Illustrate the use of * and + operators in string with example.
13. What is the symbol for comment? Give an example.
14. What is a local variable?
15. Explain the concept of floor division.
16. Write a math function to perform √2 / 2.
17. What are the different types of operators?
18. Explain modulus operator with example.
19. Explain relational operators.
20. Explain Logical operators

PART – B

1. Describe Arithmetic Operators, Assignment Operators, Comparison Operators, Logical Operators and Bitwise Operators in detail with examples.
2. Explain the Identifiers, Keywords, Statements, Expressions, and Variables in Python programming language with examples.
3. Explain the basic data types available in Python with examples.
4. Write Python Program to reverse a number and also find the Sum of digits in the reversed number. Prompt the user for input.
5. Write Pythonic code to check if a given year is a leap year or not.
6. Write Python program to find the GCD of two positive numbers.
7. Write Python code to determine whether the given string is a Palindrome or not using slicing.
8. Explain the use of join() and split() string methods with examples. Describe why strings are immutable with an example.
9. Write Python program to count the total number of vowels, consonants and blanks in a String.
10. Discuss the relation between tuples and lists, tuples and dictionaries in detail.
11. Write Python program to swap two numbers without using Intermediate/Temporary variables. Prompt the user for input.
12. Write a program that accepts a sentence and calculate the number of digits, uppercase and lowercase letters.
13. Illustrate interpreter and interactive mode in python with example.
14. Explain the data types in python
15. Write short notes on types of operators in python with appropriate example
16. Explain briefly constant, variables, expression, keywords and statements available in python
17. What is String? How do u create a string in Python?
18. How to perform a user input in Python? Explain with example.

19. Write short note on following
    I. Write a program to check whether entered string is palindrome or not.
    II. Illustrate a program to display different data types using variables and literal constants.
    III. Show how an input and output function is performed in python with an example.
20. i)Discuss the difference between tuples and list
21. ii) Discuss the various operation that can be performed on a tuple and Lists (minimum 5)with an example program.
22. i)What is membership and identity operators.
    ii) Write a program to perform addition, subtraction, multiplication, integer division, floor division and modulo division on two integer and float.
23. I) Define methods in a string with an example program using at least five methods.
    ii) How to access characters of a string?

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF ELECTRICAL AND ELECTRONICS

DEPARTMENT OF ELECTRICAL AND ELECTRONICS ENGINEERING

UNIT - 2

PYTHON – SBI1605

VARIABLES AND DATA TYPES

VARIABLES AND DATA TYPES

VARIABLES

Variable is the name given to a reserved memory locations to store values. It is also known as Identifier in python.

Need for variable:

Sometimes certain parameters will take different values at different time. Hence, in order to know the current value of such parameter we need to have a temporary memory which is identified by a name that name is called as variable. For example, our surrounding temperature changes frequently. In order to know the temperature at a particular time, we need to have a variable.

Naming and Initialization of a variable

1. A variable name is made up of alphabets (Both upper and lower cases) and digits
2. No reserved words
3. Initialize before calling
4. Multiple variables initialized
5. Dynamic variable initialization

i. Consist of upper and lower case alphabets, Numbers (0-9). E.g. X2

In the above example, a memory space is assigned to variable X2. The value of X2 is stored in this space.

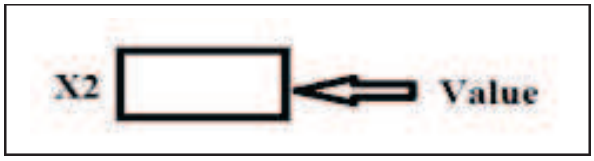


Fig 2.1 Memory space assigned to a variable

ii. Reserved words should not be used as variables names.

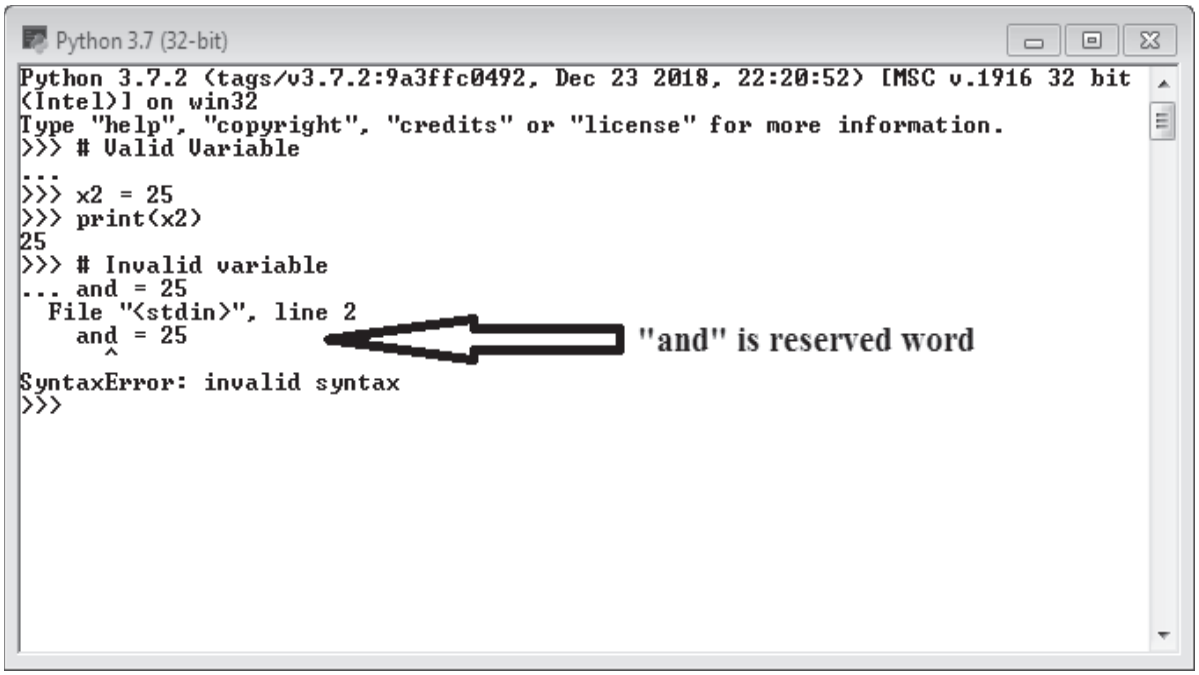


Fig 2.2 No Reserved words as variable names

In the above example “and” is a reserved word, which leads to Syntax error

iii. Variables must be initialized before it called , else it reports “is not defined ” error message as below E.g.: a = 5 print(a)

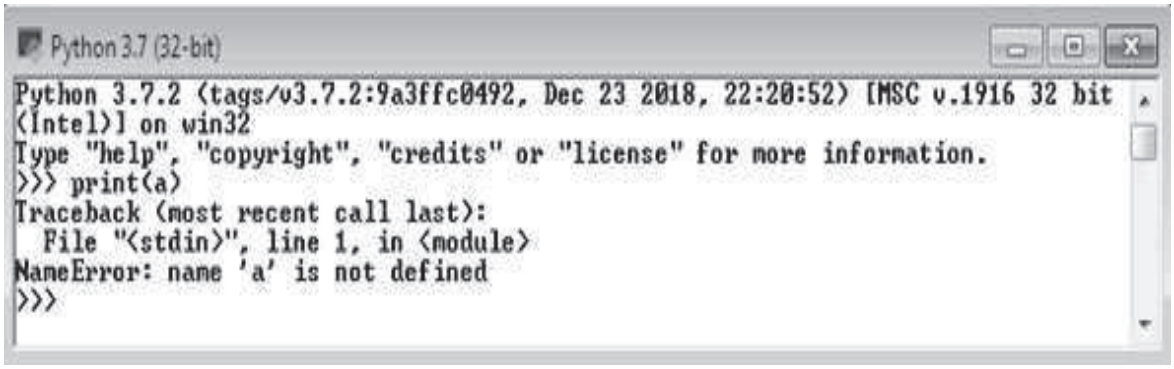


Fig 2.3 Name Error

In the above example "a" is called before it initialized. Hence, the python interpreter generates the error message: NameError: „a" is not defined.

iv. Multiple variables can be initialized with a common value. E.g. : x = y = z = 25

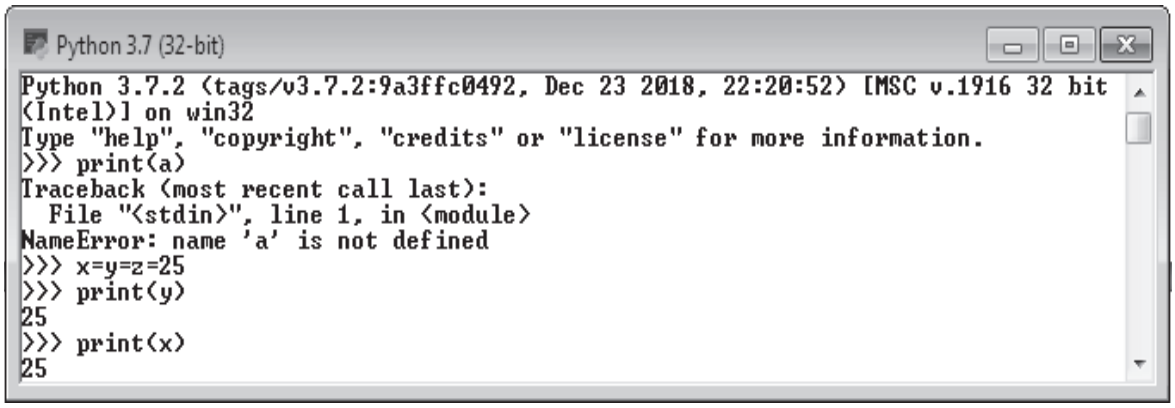


Fig 2.4 Multiple Variables

In the above three variables x, y, z is assigned with same value 25.

v. Python also supports dynamic variable initialization. E.g.: x, y, z = 1, 2, 3

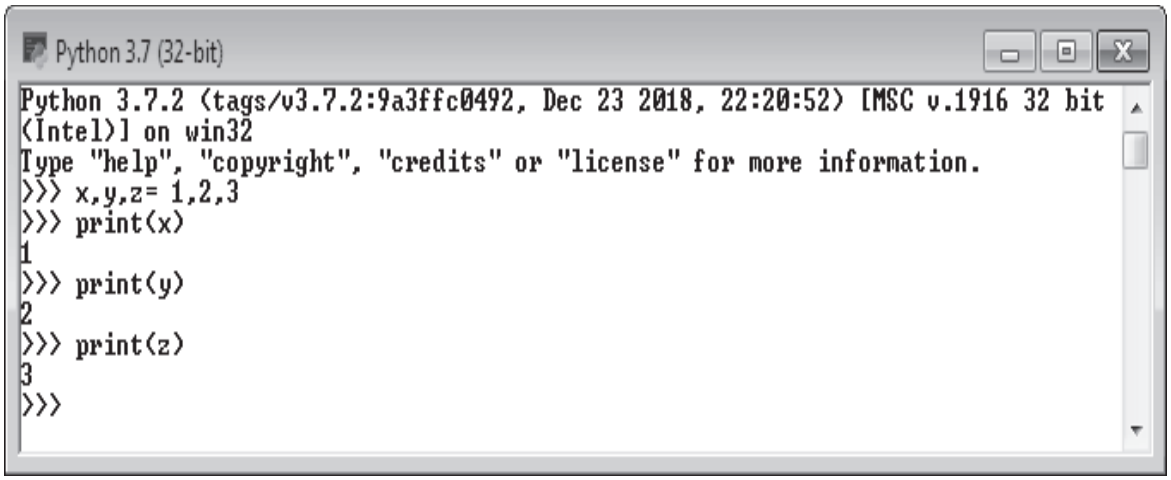


Fig 2.5 Dynamic Variable Initialization

Proper spacing should be given

- print (10+20+30) bad style
- print (20 + 30 + 10) good style

Expression:

An expression is a combination of variables, operators, values and calls to functions. Expressions need to be evaluated.

Need for Expression:

Suppose if you wish to calculate area. Area depends on various parameters in different situations. E.g. Circle, Rectangle and so on…

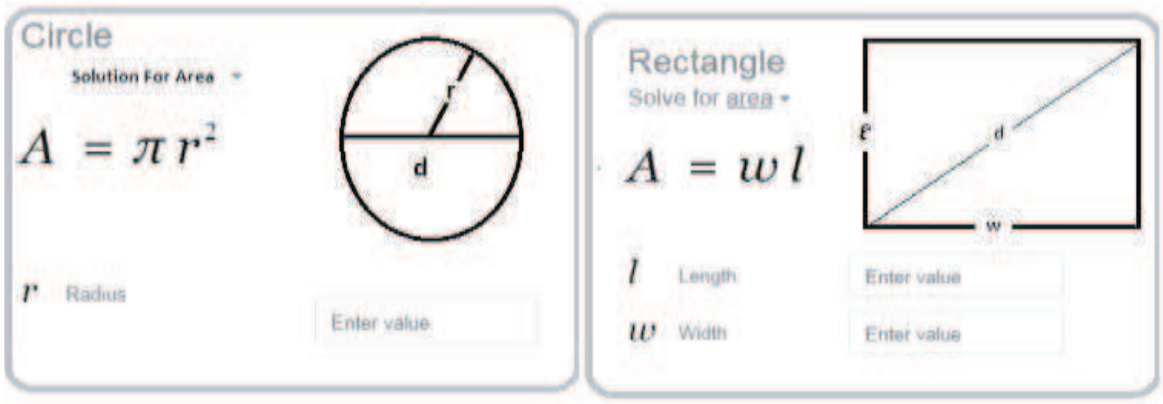


Fig 2.6 Need for Expression

In order to find area of circle, the expression π * r * r must be evaluated and for the rectangle the expression is w * l in case of rectangle. Hence, in this case a variable / value / operator are not enough to handle such situation. So expressions are used. Expression is the combination of variables, values and operations.

A simple example of an expression is 10 + 15. An expression can be broken down into operators and operands. Few valid examples are given below.

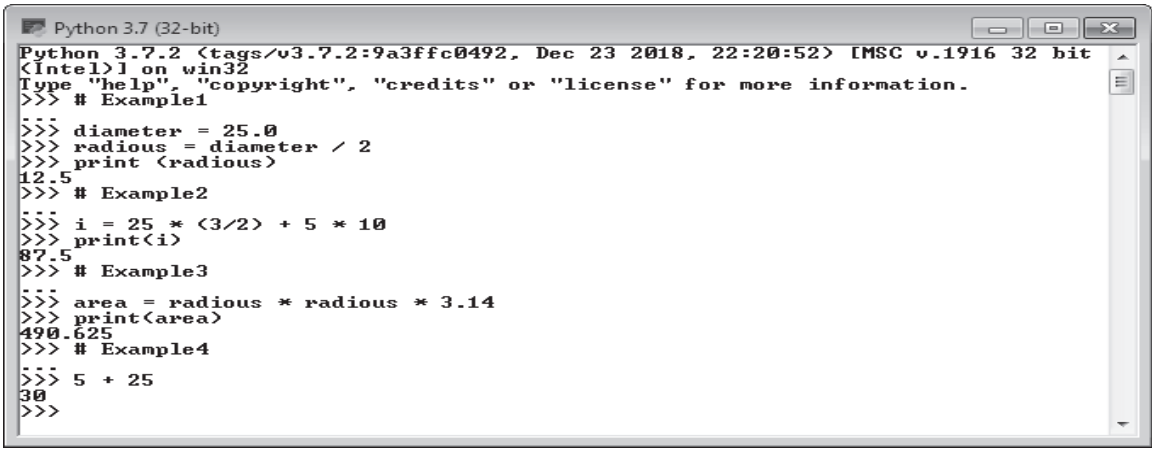


Fig 2.7 Example of and Expression

Invalid Expression:

Always values should be assigned in the right hand side of the variable, but in the below example, the value is given in the left hand side of the variable, which is an invalid syntax for expression.

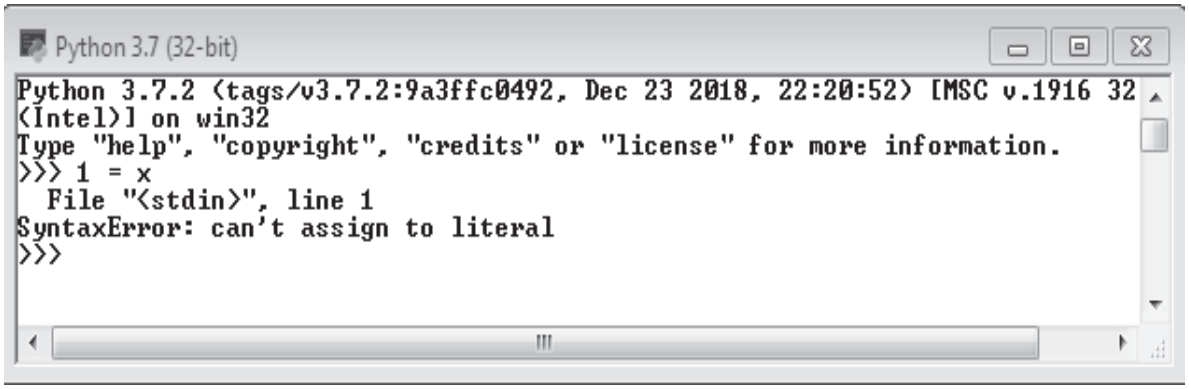


Fig 2.8 Invalid Expression

DATA TYPES

A Data type indicates which type of value a variable has in a program. However a python variables can store data of any data type but it is necessary to identify the different types of data they contain to avoid errors during execution of program. The most common data types used in python are str(string), int(integer) and float (floating-point).

Strings: Sequence of characters inside single quotes or double quotes.

E.g. myuniv = “Sathyabama !..”

Integers: Whole number values such as 50, 100,-3

Float: Values that use decimal point and therefore may have fractional point E.g.: 3.415, -5.15

By default when a user gives input it will be stored as string. But strings cannot be used for performing arithmetic operations. For example while attempting to perform arithmetic operation add on string values it just concatenates (joins together) the values together rather performing addition. For example : „25“ + „20“ = „45“ (As in the below Example)

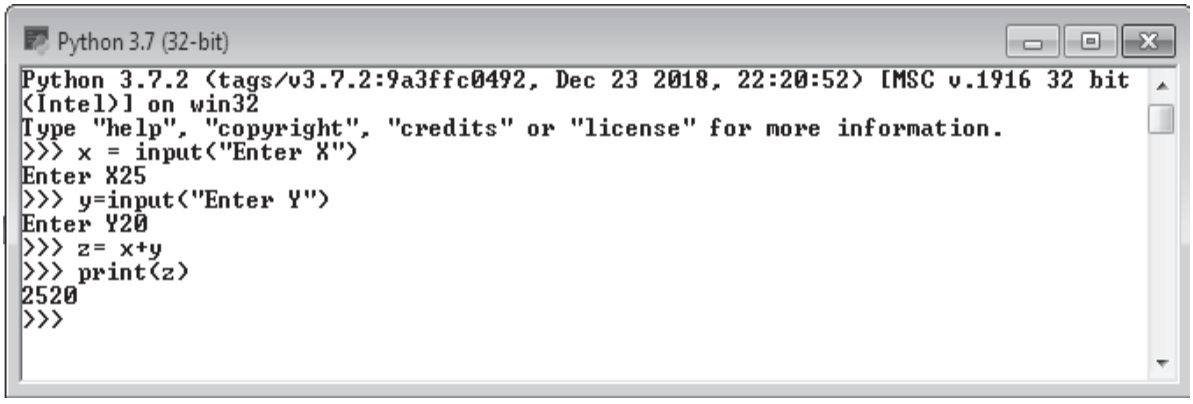


Fig 2.9 Arithmetic operation without casting

Fortunately python have an option of converting one date type into another data type (Called as “Casting”) using build in functions in python. The build function int() converts the string into integer before performing operation to give the right answer. (As in the below Program)

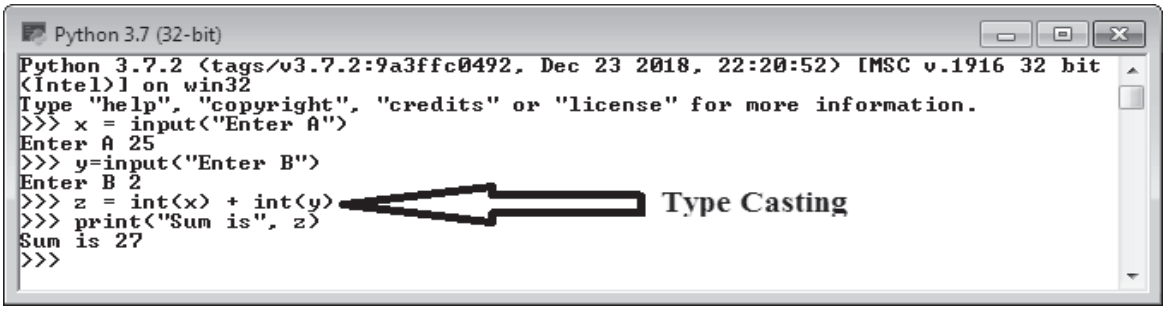


Fig 2.10 Arithmetic operation with casting

Compound Data Types in Python:

i) List

The List is an ordered sequence of data items . It is one of the flexible and very frequently used data type in Python. All the items in a list are not necessary to be of the same data type.

Declaring a list is straight forward methods. Items in the list are just separated by commas and enclosed within brackets [ ].

```
>>> list1 = [3.141, 100, 'CSE', 'ECE', 'IT', 'EEE']
```

Methods used in list

Table 2.1 List Method

list1.append(x)	To add item x to the end of the list "list1"
list1.reverse()	Reverse the order of the element in the list "list1"
list1.sort()	To sort elements in the list
list1.reverse()	To reverse the order of the elements in list1.

ii) Tuple

Tuple is also an ordered sequence of items of different data types like list. But, in a list data can be modified even after creation of the list whereas Tuples are

immutable and cannot be modified after creation.

The advantages of tuples is to write-protect data and are usually very fast when compared to lists as a tuple cannot be changed dynamically.

The elements of the tuples are separated by commas and are enclosed inside open and closed brackets.

```
>>> t = (50,'python', 2+3j)
```

Table : 2.2 List Vs Tuple

List	Tuple

>>> list1[12,45,27]
>>> list1[1] = 55
>>> print(list1)
>>> [12,55,27]
	>>> t1 = (12,45,27)
	>>> t1[1] = 55
	>>> Generates Error Message # Because Tuples are immutable

iii) Set

The Set is an unordered collection of unique data items. Items in a set are not ordered, separated by comma and enclosed inside { } braces. Sets are helpful in performing operations like union and intersection. However, indexing is not done because sets are unordered.

Table : 2.3 List Vs Set

List	Set

>>> L1 = [1,20,25]
>>> print(L1[1])
>>> 20
	>>> S1= {1,20,25,25}
	>>> print(S1)
	>>> {1,20,25}
	>>> print(S1[1])
	>>> Error , Set object does not support
	indexing.

iv) Dictionary

The Python Dictionary is an unordered collection of key-value pairs. Dictionaries is optimized for retrieving data when there is huge volume of data. They provide the key to retrieve the value.

In Python, dictionaries are defined within braces {} with each item being a pair in the form key: value. Key and value can be of any type.

```
>>> d1 = {1:'value','key':2}

>>> type(d)
```

References:

1. Hetland., “Beginning Python”, Apress, 2008
2. Mark Pilgrim, “Drive Into Python”, Apress, 2004
3. Martin C. Brown, “ Python: The Complete Reference (English)”, McGraw-Hill/Osborne Media, 2001.
4. Mark Summerfiled, “Programming in Python 3”, 2nd ed (PIP3), Addison Wesley.
5. https://www.academia.edu/41039821/Python_Tutorial_Release_3_7_0_Guido_van_Rossum_and_the_Python_development_team

UNIT - 2

PART – A

1. What is List?
2. What is tuple?
3. What is Dictionary?
4. What is sets?
5. What are string methods?
6. Compare string and string slices.
7. Explain about string module.
8. Mention a few string functions.
9. Solve a)[0] * 4 and b) [1, 2, 3] * 3.
10. Let list = [’a’, ’b’, ’c’, ’d’, ’e’, ’f’]. Find a) list[1:2] b) list[:3] c) list[2:] .
11. Mention any 5 list methods.
12. State the difference between lists and dictionary.
13. What is List mutability in Python? Give an example.
14. Write a program in Python to delete first element from a list.
15. Write a program in Python to delete last element from a list.
16. What is the benefit of using tuple assignment in Python?
17. Define dictionary with an example.

PART – B

1. What are the basic list operations that can be performed in Python?
2. Explain each operation with its syntax and example.
3. What is Dictionary? Explain Python dictionaries in detail discussing its operations and methods.
4. Explain the features of a dictionary.
5. What is the difference between lists, tuples and dictionaries? Give an example for their usage.

6. Demonstrate the various expressions in python with suitable examples.
7. Describe the following
    a) Creating the List
    b) Accessing values in the Lists
    c) Updating the Lists
    d) Deleting the list Elements
8. Explain the basic List Operations in details with necessary programs
9. Write a Python program to multiply two Matrices.
10. Illustrate List Comprehension with suitable examples
11. Write a python program to concatenate two lists
12. What is a Python Tuple? What are the Advantages of Tuple over List?
13. “Tuples are immutable”. Explain with Examples
14. Illustrate the ways of creating the tuple and the tuple assignment with suitable programs
15. What are the accessing elements in a tuple? Explain With suitable Programs.
16. Explain the properties of Dictionary keys with examples
17. Illustrate the python Dictionary Comprehension with example.
18. Explain the use of slice operator for accessing elements of a tuple.
19. What are the different methods for adding elements to a list? Give example for each method.
20. What is exception handling? How does it work?
21. Write a program to sort a dictionary in ascending and descending order of values.

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF ELECTRICAL AND ELECTRONICS

DEPARTMENT OF ELECTRICAL AND ELECTRONICS ENGINEERING

UNIT - 3

PYTHON – SBI1605
REGULAR EXPRESSION

REGULAR EXPRESSIONS

A *regular expression* is a special sequence of characters that helps you match or find other strings or sets of strings, using a specialized syntax held in a pattern. Regular expressions are widely used in UNIX world.

The Python module **re** provides full support for Perl-like regular expressions in Python. The re module raises the exception re.error if an error occurs while compiling or using a regular expression.

We would cover two important functions, which would be used to handle regular expressions. But a small thing first: There are various characters, which would have special meaning when they are used in regular expression. To avoid any confusion while dealing with regular expressions, we would use Raw Strings as **r'expression'**.

The *match* Function

This function attempts to match RE *pattern* to *string* with

optional *flags*. Here is the syntax for this function −

re.match(pattern, string, flags=0)

Here is the description of the parameters −

Sr.No.	Parameter & Description
1	**pattern** This is the regular expression to be matched.
2	**String** This is the string, which would be searched to match the pattern at the beginning of string.
3	**Flags** You can specify different flags using bitwise OR (|). These are modifiers, which are listed in the table below.

The *re.match* function returns a **match** object on success, **None** on failure. We use*group(num)* or *groups()* function of **match** object to get matched expression.

Sr.No.	Match Object Method & Description
1	**group(num=0)** This method returns entire match (or specific subgroup num)
2	**groups()** This method returns all matching subgroups in a tuple (empty if there weren't any)

Example

```
#!/usr/bin/python
import re

line = "Cats are smarter than dogs"

matchObj = re.match( r'(.*) are (.*?) .*', line, re.M|re.I)

if matchObj:
   print "matchObj.group() : ", matchObj.group()
   print "matchObj.group(1) : ", matchObj.group(1)
   print "matchObj.group(2) : ", matchObj.group(2)
else:
   print "No match!!"
```

When the above code is executed, it produces following result −

```
matchObj.group() :  Cats are smarter than dogs
matchObj.group(1) :  Cats
matchObj.group(2) :  smarter
```

The *search* Function

This function searches for first occurrence of RE *pattern* within *string* with optional *flags*.

Here is the syntax for this function −

re.search(pattern, string, flags=0) Here is the

description of the parameters –

Sr.No.	Parameter & Description
1	**pattern**

	This is the regular expression to be matched.
2	**string** This is the string, which would be searched to match the pattern anywher
3	**flags** You can specify different flags using bitwise OR (|). These are modifiers, in the table below.

The *re.search* function returns a **match** object on success, **none** on failure. We use *group(num)* or *groups()* function of **match** object to get matched expression.

Sr.No.	Match Object Methods & Description
1	**group(num=0)** This method returns entire match (or specific subgroup num)
2	**groups()** This method returns all matching subgroups in a tuple (empty if there weren't any)

Example

```
#!/usr/bin/python
import re

line = "Cats are smarter than dogs";

searchObj = re.search( r'(.*) are (.*?) .*', line, re.M|re.I)

if searchObj:
   print "searchObj.group() : ", searchObj.group()
   print "searchObj.group(1) : ", searchObj.group(1)
   print "searchObj.group(2) : ", searchObj.group(2)
else:
   print "Nothing found!!"
```

When the above code is executed, it produces following result −

```
searchObj.group() :     Cats are smarter than dogs
searchObj.group(1) :      Cats
searchObj.group(2) :      smarter
```

Matching Versus Searching

Python offers two different primitive operations based on regular expressions: **match** checks for a match only at the beginning of the string, while **search** checks for a match anywhere in the string (this is what Perl does by default).

Example

```
#!/usr/bin/python
import re

line = "Cats are smarter than dogs";

matchObj = re.match( r'dogs', line, re.M|re.I)
if matchObj:
   print "match --> matchObj.group() : ", matchObj.group()
else:
   print "No match!!"

searchObj = re.search( r'dogs', line, re.M|re.I)
if searchObj:
   print "search --> searchObj.group() : ", searchObj.group()
else:
   print "Nothing found!!"
```

When the above code is executed, it produces the following result −

```
No match!!
search --> searchObj.group() :                    dogs
```

Search and Replace

One of the most important **re** methods that use regular expressions is **sub**.

Syntax

re.sub(pattern, repl, string, max=0)

This method replaces all occurrences of the RE *pattern* in *string* with *repl*, substituting all occurrences unless *max* provided. This method returns modified string.

Example

```
#!/usr/bin/python
import re
```

```
phone = "2004-959-559 # This is Phone Number"

# Delete Python-style comments
num = re.sub(r'#.*$', "", phone)
print "Phone Num : ", num

# Remove anything other than digits
num = re.sub(r'\D', "", phone)
print "Phone Num : ", num
```

When the above code is executed, it produces the following result −

```
Phone Num :  2004-959-559
Phone Num :  2004959559
```

Regular Expression Modifiers: Option Flags

Regular expression literals may include an optional modifier to control various aspects of matching. The modifiers are specified as an optional flag. You can provide multiple modifiers using exclusive OR (|), as shown previously and may be represented by one of these −

Sr.No.	Modifier & Description
1	**re.I** Performs case-insensitive matching.
2	**re.L** Interprets words according to the current locale. This interpretation affects the alphabetic group (\w and \W), as well as word boundary behavior(\b and \B).
3	**re.M** Makes $ match the end of a line (not just the end of the string) and makes ^ match the start of any line (not just the start of the string).
4	**re.S** Makes a period (dot) match any character, including a newline.
5	**re.U** Interprets letters according to the Unicode character set. This flag affects the behavior of \w, \W, \b, \B.

6	**re.X** Permits "cuter" regular expression syntax. It ignores whitespace (except insi when escaped by a backslash) and treats unescaped # as a comment mark

Regular Expression Patterns

Except for control characters, **(+ ? . * ^ $ ( ) [ ] { } | \)**, all characters match themselves. You can escape a control character by preceding it with a backslash.

Following table lists the regular expression syntax that is available in Python –

Sr.No.	Pattern & Description
1	**^** Matches beginning of line.
2	**$** Matches end of line.
3	**.** Matches any single character except newline. Using m option allows it to match newline as well.
4	**[...]** Matches any single character in brackets.
5	**[^...]** Matches any single character not in brackets
6	**re*** Matches 0 or more occurrences of preceding expression.
7	**re+** Matches 1 or more occurrence of preceding expression.

8	**re?** Matches 0 or 1 occurrence of preceding expression.
9	**re{ n}** Matches exactly n number of occurrences of preceding expression.
10	**re{ n,}** Matches n or more occurrences of preceding expression.
11	**re{ n, m}** Matches at least n and at most m occurrences of preceding expression.
12	**a| b** Matches either a or b.
13	**(re)** Groups regular expressions and remembers matched text.
14	**(?imx)** Temporarily toggles on i, m, or x options within a regular expression. If in parentheses, only that area is affected.
15	**(?-imx)** Temporarily toggles off i, m, or x options within a regular expression. If in parentheses, only that area is affected.
16	**(?: re)** Groups regular expressions without remembering matched text.
17	**(?imx: re)** Temporarily toggles on i, m, or x options within parentheses.
18	**(?-imx: re)** Temporarily toggles off i, m, or x options within parentheses.

19	**(?#...)** Comment.
20	**(?= re)** Specifies position using a pattern. Doesn't have a range.
21	**(?! re)** Specifies position using pattern negation. Doesn't have a range.
22	**(?> re)** Matches independent pattern without backtracking.
23	**\w** Matches word characters.
24	**\W** Matches nonword characters.
25	**\s** Matches whitespace. Equivalent to [\t\n\r\f].
26	**\S** Matches nonwhitespace.
27	**\d** Matches digits. Equivalent to [0-9].
28	**\D** Matches nondigits.
29	**\A** Matches beginning of string.

30	**\Z** Matches end of string. If a newline exists, it matches just before newline.
31	**\z** Matches end of string.
32	**\G** Matches point where last match finished.
33	**\b** Matches word boundaries when outside brackets. Matches backspace (0x08) when inside brackets.
34	**\B** Matches nonword boundaries.
35	**\n, \t, etc.** Matches newlines, carriage returns, tabs, etc.
36	**\1...\9** Matches nth grouped subexpression.
37	**\10** Matches nth grouped subexpression if it matched already. Otherwise refers to the octal representation of a character code.

Regular Expression Examples

Literal characters

Sr.No.	Example & Description

1	**python** Match "python".

Character classes

Sr.No.	Example & Description
1	**[Pp]ython** Match "Python" or "python"
2	**rub[ye]** Match "ruby" or "rube"
3	**[aeiou]** Match any one lowercase vowel
4	**[0-9]** Match any digit; same as [0123456789]
5	**[a-z]** Match any lowercase ASCII letter
6	**[A-Z]** Match any uppercase ASCII letter
7	**[a-zA-Z0-9]** Match any of the above
8	**[^aeiou]** Match anything other than a lowercase vowel
9	**[^0-9]** Match anything other than a digit

Special Character Classes

Sr.No.	Example & Description
1	**.** Match any character except newline
2	**\d** Match a digit: [0-9]
3	**\D** Match a nondigit: [^0-9]
4	**\s** Match a whitespace character: [\t\r\n\f]
5	**\S** Match nonwhitespace: [^ \t\r\n\f]
6	**\w** Match a single word character: [A-Za-z0-9_]
7	**\W** Match a nonword character: [^A-Za-z0-9_]

Repetition Cases

Sr.No.	Example & Description
1	**ruby?** Match "rub" or "ruby": the y is optional

2	**ruby*** Match "rub" plus 0 or more ys

3	**ruby+** Match "rub" plus 1 or more ys
4	**\d{3}** Match exactly 3 digits
5	**\d{3,}** Match 3 or more digits
6	**\d{3,5}** Match 3, 4, or 5 digits

Nongreedy repetition

This matches the smallest number of repetitions −

Sr.No.	Example & Description
1	**<.*>** Greedy repetition: matches "<python>perl>"
2	**<.*?>** Nongreedy: matches "<python>" in "<python>perl>"

Grouping with Parentheses

Sr.No.	Example & Description
1	**\D\d+** No group: + repeats \d

2	**(\D\d)+** Grouped: + repeats \D\d pair

3	**([Pp]ython(,)?)+** Match "Python", "Python, python, python", etc.

This matches a previously matched group again −

Sr.No.	Example & Description
1	**([Pp])ython&\1ails** Match python&pails or Python&Pails
2	**(['"])[^\1]*\1** Single or double-quoted string. \1 matches whatever the 1st group matched. \2 matches whatever the 2nd group matched, etc.

Alternatives

Sr.No.	Example & Description
1	**python|perl** Match "python" or "perl"
2	**rub(y|le))** Match "ruby" or "ruble"
3	**Python(!+|\?)** "Python" followed by one or more ! or one ?

Anchors

This needs to specify match position.

Sr.No.	Example & Description
1	**^Python** Match "Python" at the start of a string or internal line
2	**Python$** Match "Python" at the end of a string or line
3	**\APython** Match "Python" at the start of a string
4	**Python\Z** Match "Python" at the end of a string
5	**\bPython\b** Match "Python" at a word boundary
6	**\brub\B** \B is nonword boundary: match "rub" in "rube" and "ruby" but not alone
7	**Python(?=!)** Match "Python", if followed by an exclamation point.
8	**Python(?!!)** Match "Python", if not followed by an exclamation point.

Special Syntax with Parentheses

Sr.No.	Example & Description
1	**R(?#comment)** Matches "R". All the rest is a comment
2	**R(?i)uby**
	Case-insensitive while matching "uby"
3	**R(?i:uby)** Same as above
4	**rub(?:y|le))** Group only without creating \1 backreference

References:

1. Hetland., “Beginning Python”, Apress, 2008
2. Mark Pilgrim, “Drive Into Python”, Apress, 2004
3. Martin C. Brown, “ Python: The Complete Reference (English)”, McGraw-Hill/Osborne Media, 2001.
4. Mark Summerfiled, “Programming in Python 3”, 2nd ed (PIP3), Addison Wesley.
5. https://www.academia.edu/41039821/Python_Tutorial_Release_3_7_0_Guido_van_Rossum_and_the_Python_development_team

UNIT - 3

PART – A

1. Which module in Python supports regular expressions?
2. What does the function re.match do?
3. What does the function re.findall do?
4. What does the function re.split do?
5. What does the function re.sub do?
6. Define metacharacter?
7. Define modifier.
8. Express the importance of Pattern in Regular expression?
9. Why do you need regular expressions in Python?
10. Discuss the search() methods supported by compiled regular expression objects.
11. Discuss the match() methods supported by compiled regular expression objects.
12. Discuss the findall() methods supported by compiled regular expression objects.

PART – B

1. Discuss the following methods supported by compiled regular expression objects.
   a) search() b) match() c) findall()
2. Why do you need regular expressions in Python? Consider a file “ait.txt”. Write a Python program to read the file and look for lines of the form X-DSPAM-Confidence: 0.8475 X-DSPAM-Probability: 0.458 Extract the number from each of the lines using a regular expression. Compute the average of the numbers and print out the average.
3. Write Pythonic code to read the file and extract email address from the lines starting from the word “From”. Use regular expressions to match email address.

4. Explain replace () and split () methods of regular expression with suitable examples.
5. Explain match () and findall () methods of regular expression with suitable examples.
6. Explain group (),span (),string () and sub () methods of regular expression with suitable examples.
7. What is regular expression ? What are diffrent type of regular expression?

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF ELECTRICAL AND ELECTRONICS

DEPARTMENT OF ELECTRICAL AND ELECTRONICS ENGINEERING

UNIT - 4

PYTHON – SBI1605
CONTROL STATEMENTS

CONDITIONAL STATEMENTS

When there is no condition placed before any set of statements, the program will be executed in sequential manure. But when some condition is placed before a block of statements the flow of execution might change depends on the result evaluated by the condition. This type of statement is also called decision making statements or control statements. This type of statement may skip some set of statements based on the condition.

Logical Conditions Supported by Python

- Equal to (==) Eg : a == b
- Not Equal (!=)Eg : a != b
- Greater than (>) Eg : a > b
- Greater than or equal to (>=) Eg : a >= b
- Less than (<) Eg : a < b
- Less than or equal to (<=) Eg : a <= b

Indentation

To represent a block of statements other programming languages like C, C++ uses "{ …}" curly – brackets , instead of this curly braces python uses indentation using white space which defines scope in the code. The example

given below shows the difference between usage of Curly bracket and white space to represent a block of statement.

Table 4.1 : C- Program Vs Python

C Program	Python

x = 500
y = 200
if (x > y)
{
printf("x is greater than y")
}
else if(x == y)
{
printf("x and y are equal")
}
else
{
printf("x is less than y")
	x = 500
	y = 200
	if x > y:
	print("x is greater than y")
	elif x == y:
	print("x and y are equal")
	else:
	print("x is less than y")
	 
	Indentation (At least one White Space instead of curly bracket)

Without proper Indentation:

```
x = 500
y = 200
if x > y:
print("x is greater than y")
```

In the above example there is no proper indentation after if statement which will lead to Indentation error.

If statement:

The „if‟ statement is written using “if” keyword, followed by a condition. If the condition is true the block will be executed. Otherwise, the control will be

transferred to the first statement after the block.

Syntax :

if <Boolean>:

<block>

In this statement, the order of execution is purely based on the evaluation of boolean expression.

Example:

```
x = 200
y = 100
if x > y:
  print("X is greater than Y")
print(“End”)
```

Output :

X is greater than

Y End

In the above the value of x is greater than y , hence it executed the print statement whereas in the below example x is not greater than y hence it is not executed the first print statement.

```
x = 100
y = 200
if x > y:
  print("X is greater than Y")
print(“End”)
```

Output :

End

Elif

The **elif** keyword is useful for checking another condition when one condition is false.

Example :

```
mark = 55
if (mark >=75):
  print("FIRST CLASS")
elif mark >= 50:
  print("PASS")
```

Output :

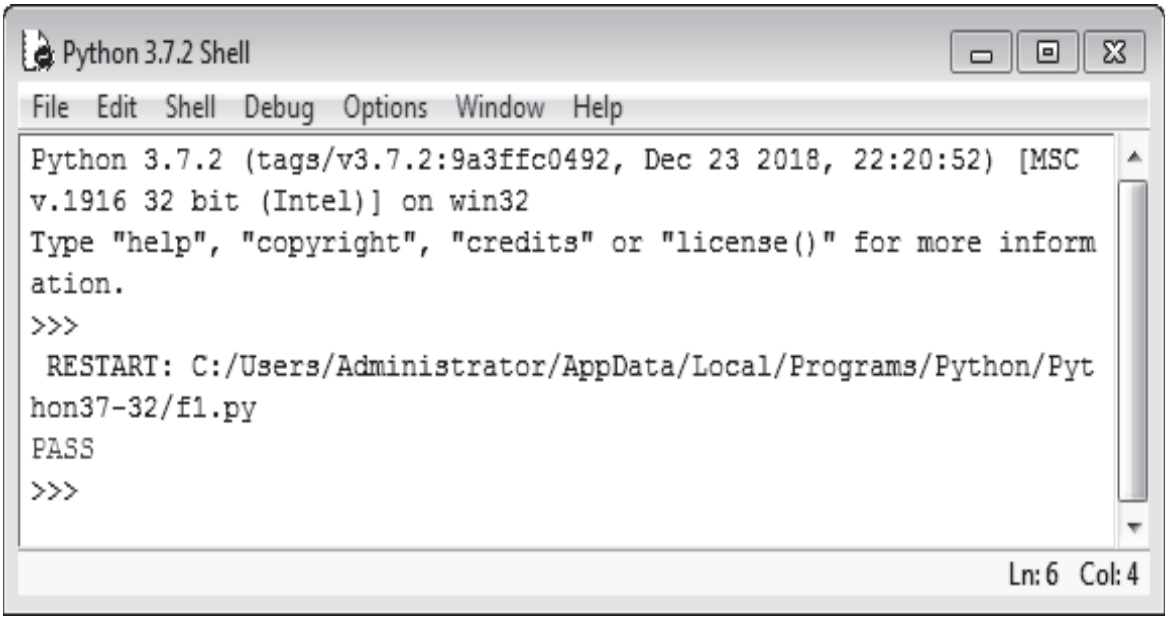


Fig 4.1 Output

In the above the example, the first condition (mark >=75) is false then the control is transferred to the next condition (mark >=50), Thus, the keyword **elif** will be helpful for having more than one condition.

Else

The **else** keyword will be used as a default condition. i.e. When there are many conditions, when the **if-condition** is not true and all **elif-conditions** are also not true, then **else** part will be executed..

Example :

```
mark = 10
if mark >= 75:
  print("FIRST CLASS")
elif mark >= 50:
print("PASS")
else:
print("FAIL")
```

Output:

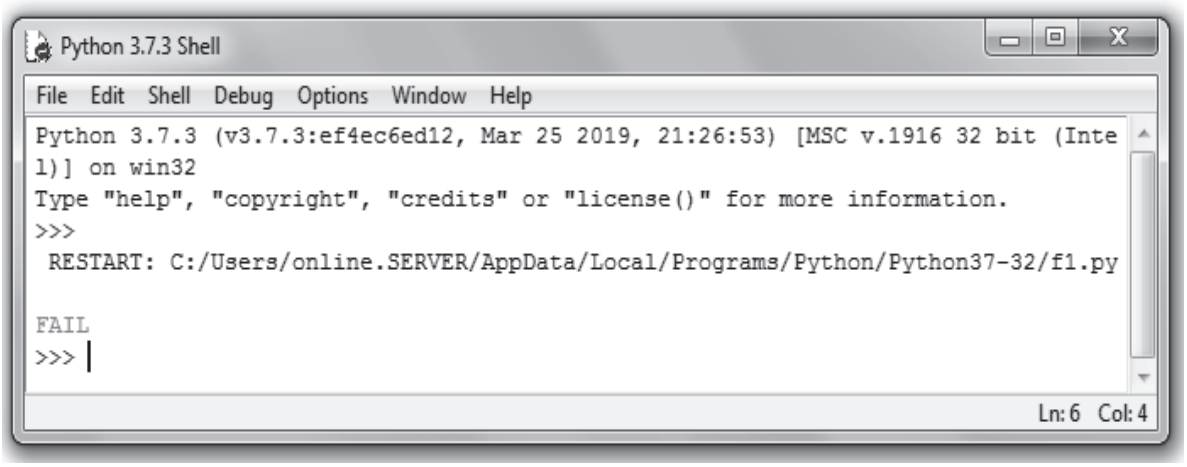


Fig 4.2 Output

In the example above, condition 1 and condition 2 fail. None of the preceding condition is true. Hence, the **else** part is executed.

ITERATIVE STATEMENTS

Sometimes certain section of the code (block) may need to be repeated again and again as long as certain condition remains true. In order to achieve this, the iterative statements are used. The number of times the block needs to be repeated is controlled by the test condition used in that statement. This type of

statement is also called as the “Looping Statement”. Looping statements add a surprising amount of new power to the program.

Need for Looping / Iterative Statement

Suppose the programmer wishes to display the string “Sathyabama !...” 150 times. For this, one can use the print command 150 times.

```
print(“Sathyabama !...”)
print(“Sathyabama !...”)
…..
…..
```

150 times

The above method is somewhat difficult and laborious. The same result can be achieved by a loop using just two lines of code. (As below)

```
for count in range(1,150) :
    print (“Sathyabama
```

Types of looping statements

1) **for** loop
2) **while** loop

The 'for' Loop

The **for** loop is one of the powerful and efficient statements in python which is used very often. It specifies how many times the body of the loops needs to be executed. For this reason it uses control variables which keep tracks, the count of execution. The general syntax of a „for" loop looks as below:

```
for <variable> in range (A,B):

    <body of the loop >
```

Flow Chart:

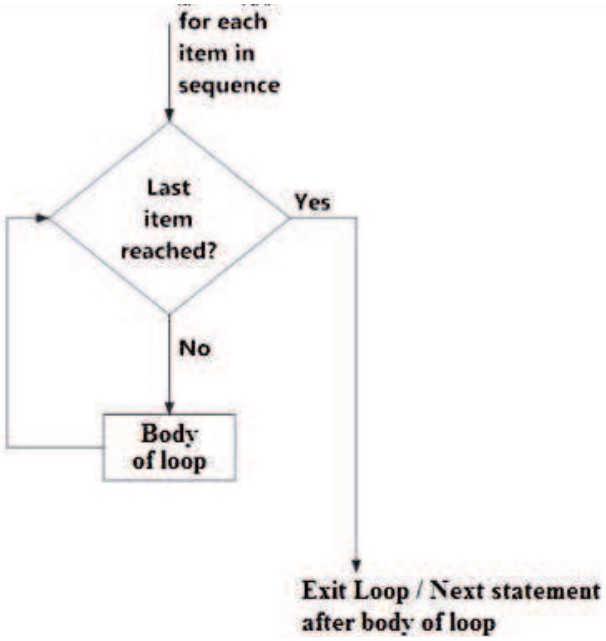


Fig 4.3 Flow Chart

Example: To compute the sum of first n numbers (i.e. 1 + 2 + 3 + ……. + n)

```
# Sum.py
total = 0
n = int (input ("Enter a Positive Number"))
for i in range(1,n+1):
    total = total + i
print ("The Sum is ", total)
```

Note: Why (n+1)? Check in table given below.

Output:

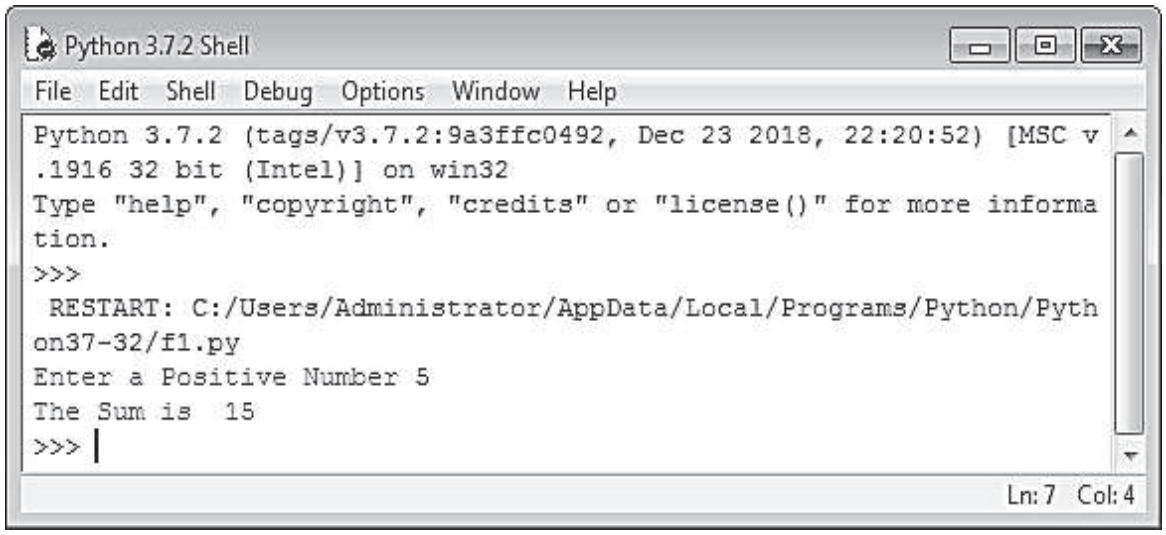


Fig 4.4 Output

In the above program, the statement total = total + i is repeated again and again „n" times. The number of execution count is controlled by the variable „i". The range value is specified earlier before it starts executing the body of loop. The initial value for the variable i is 1 and final value depends on „n". You may

also specify any constant value.

The range( ) Function:

The **range( )** function can be called in three different ways based on the number of parameters. All parameter values must be integers.

Table 4.2 : Categories of range function

Type	Example	Explanation
range(end)	for i in range(5): print(i) **Output :** 0,1,2,3,4	This is begins at 0. Increments by 1. End just before the value of end parameter.
range(begin, end)	for i in range(2,5): print(i) **Output :** 2,3,4	Starts at begin, End before end value, Increment by 1
range(begin,end,step)	for i in range(2,7,2) print(i) **Output :** 2,4,6	Starts at begin, End before end value, increment by step value

Example : To compute Harmonic Sum (ie: 1 + ½ + 1/3 + ¼ + …..1/n)

```
# harmonic.py
total = 0
n= int(input("Enter a Positive Integer:"))
for i in range(1,n+1):
    total+= 1/i
print("The Sum of range 1 to ",n, "is", total)
```

Output:

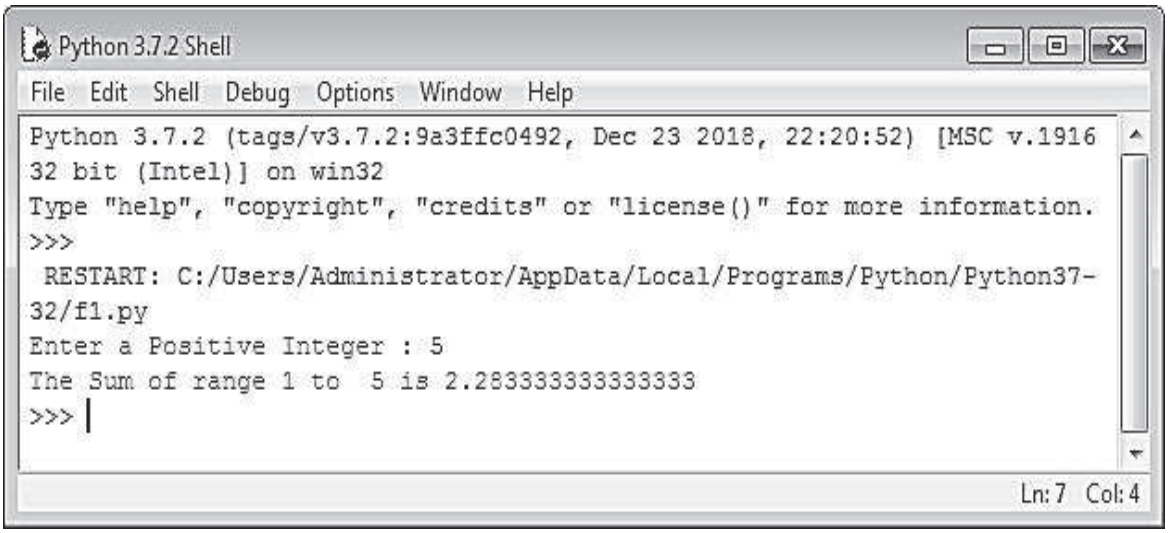


Fig 4.5 Output

Example :

Factorial of a number “n"

```
n= int(input("Enter a Number :"))
factorial = 1
# Initialize factorial value by 1
# To verify whether the given number is negative / positive / zero
if n < 0:
   print("Negative Number , Enter valid Number !...")
elif n == 0:
   print("The factorial of 0 is 1")
else:
   for i in range(1, n + 1):
       factorial = factorial*i
   print("The factorial of" ,n, "is", factorial)
```

Output:

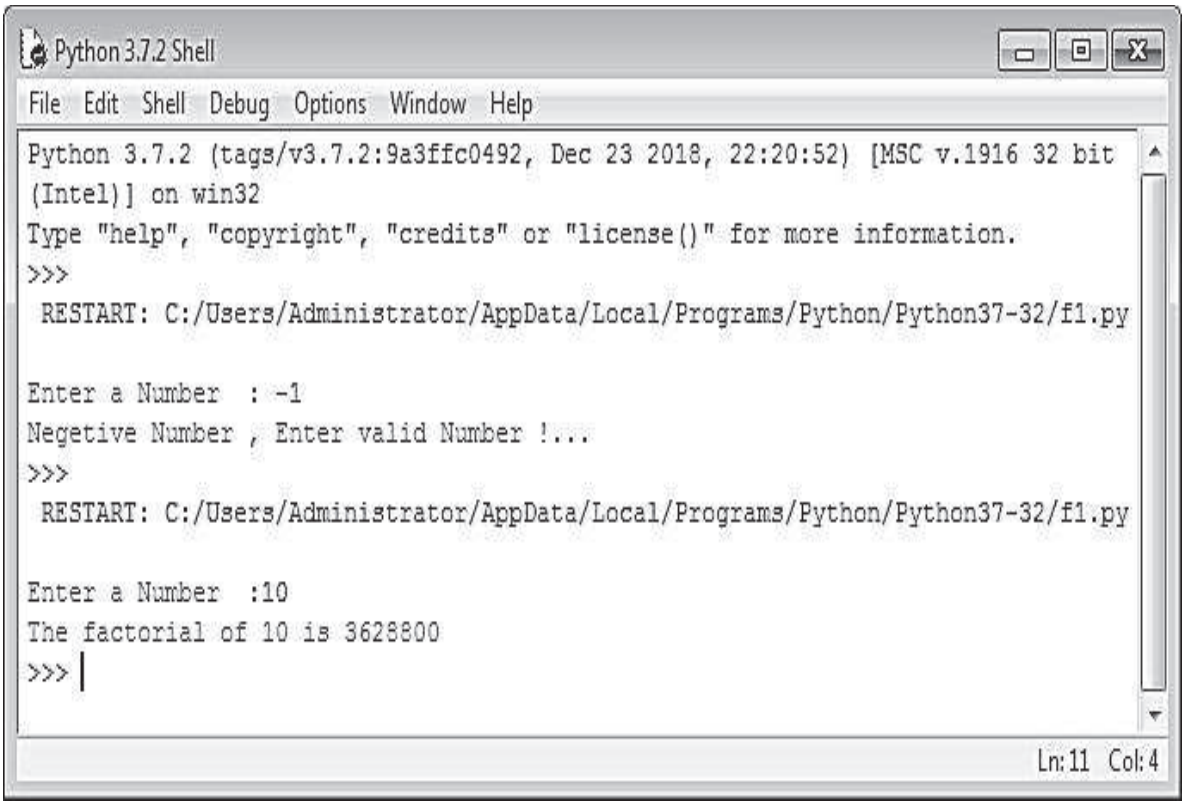


Fig 4.5 Output

The while Loop

The **while** loop allows the program to repeat the body of a loop, any number of times, when some condition is true.

The drawback of **while** loop is that, if the condition is not proper it may lead to infinite looping.

So the user has to carefully choose the condition in such a way that it will terminate at a particular stage.

Flow Chart:

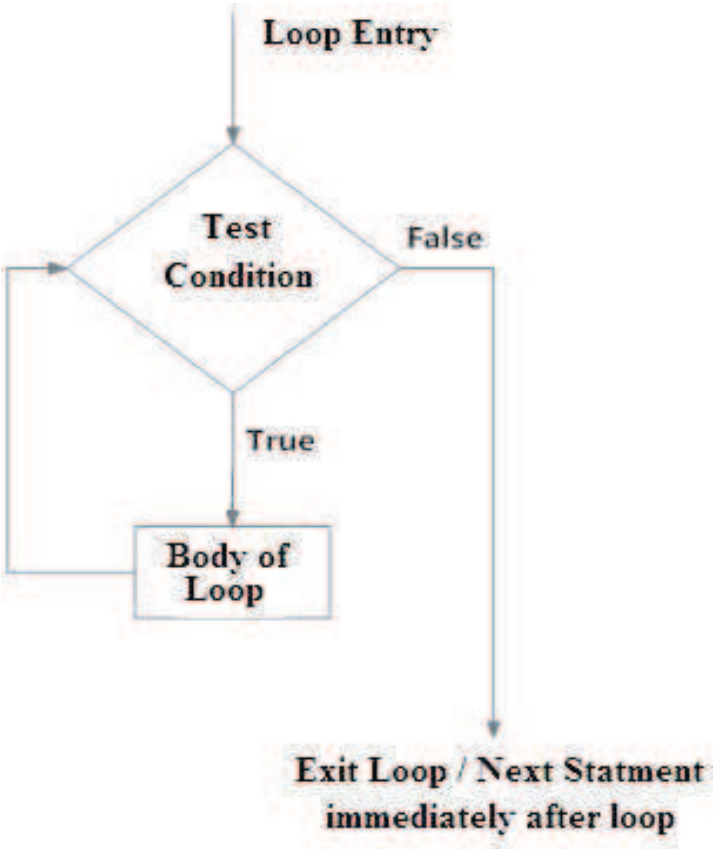


Fig 4.6 Flow Chart

Syntax:

```
while (condition):
        <body of the loop>
```

In this type of loop, The execution of the loop body is purely based on the output of the given condition. As long as the condition is TRUE or in other

words until the condition becomes FALSE the program will repeat the body of loop.

Valid Example	**Invalid Example**
i = 10 while i<15 : print(i) i = i + 1 **Output :** 10,11,12,13,14	i = 10 while i<15 : print(i) **Output :** 10,10,10,10… ..... Indeterminate number of times

Example: Program to display Fibonacci Sequence

```
# Program to Display Fibonacci Sequence based on number of terms n
n = int(input("Enter number of terms in the sequence you want to display"))
# n1 represents -- > first term and n2 represents --> Second term
n1 = 0
n2 = 1
count = 0
# count -- To check number of terms
if n <= 0:                    # To check whether valid number of terms
  print ("Enter a positive integer")
elif n == 1:
  print("Fibonacci sequence up to ", n,":")
  print(n2)
else:
  print("Fibonacci sequence of ",n, " terms :" )
  while count < n:
     print(n1,end=' , ')
     nth = n1 + n2
     n1 = n2
     n2 = nth
     count = count + 1
```

References:

1. Hetland., “Beginning Python”, Apress, 2008
2. Mark Pilgrim, “Drive Into Python”, Apress, 2004
3. Martin C. Brown, “ Python: The Complete Reference (English)”, McGraw-Hill/Osborne Media, 2001.
4. Mark Summerfiled, “Programming in Python 3”, 2nd ed (PIP3), Addison Wesley.
5. https://www.academia.edu/41039821/Python_Tutorial_Release_3_7_0_Guido_van_Rossum_and_the_Python_development_team

UNIT - 4

PART – A

1. What is conditional execution?
2. What is alternative execution?
3. What are chained conditionals?
4. Explain if loop with example.
5. Explain while loop with example.
6. Explain nested loop with example.
7. What is a break statement?
8. What is a continue statement?
9. What is a pass statement?
10. Define List Comprehension.
11. Write the syntax for list comprehension.

PART – B

1. Explain the syntax and flow chart of the following loop statements
    (i) for loop
    (ii) while loop
2. Explain the syntax and flow chart of the following loop statements
    (i) Nested loop
    (ii) continue inside if loop
3. (i). Illustrate the flow chart and syntax of if-elif- else statements
4. (ii). Develop a program to find the largest among three numbers
5. Which are two types of else clauses in Python, define them?

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF ELECTRICAL AND ELECTRONICS

DEPARTMENT OF ELECTRICAL AND ELECTRONICS ENGINEERING

UNIT - 5

PYTHON – SBI1605

FUNCTIONS AND MODULES

FUNCTIONS AND MODULES

FUNCTIONS

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

As you already know, Python gives you many built-in functions like print(), etc. but you can also create your own functions. These functions are called *user-defined functions.*

Defining a Function

You can define functions to provide the required functionality. Here are simple rules to define a function in Python.

- Function blocks begin with the keyword **def** followed by the function name and parentheses (( )).
- Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
- The first statement of a function can be an optional statement - the documentation string of the function or *docstring*.
- The code block within every function starts with a colon (:) and is indented.
- The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

Syntax

```
def functionname( parameters ):
   "function_docstring"
   function_suite
   return [expression]
```

By default, parameters have a positional behavior and you need to inform them in the same order that they were defined.

Example

The following function takes a string as input parameter and prints it on standard screen.

```
def printme( str ):
   "This prints a passed string into this function"
   print str
   return
```

Calling a Function

Defining a function only gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code.

Once the basic structure of a function is finalized, you can execute it by calling it from another function or directly from the Python prompt. Following is the example to call printme() function −

```
#!/usr/bin/python

# Function definition is here
def printme( str ):
   "This prints a passed string into this function"
   print str
   return;

# Now you can call printme function
printme("I'm first call to user defined function!")
printme("Again second call to the same function")
```

When the above code is executed, it produces the following result −

```
I'm first call to user defined function!
Again second call to the same function
```

Pass by reference vs value

All parameters (arguments) in the Python language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back in the calling function. For example −

```
#!/usr/bin/python

# Function definition is here
def changeme( mylist ):
   "This changes a passed list into this function"
   mylist.append([1,2,3,4]);
   print "Values inside the function: ", mylist
   return

# Now you can call changeme function
mylist = [10,20,30];
changeme( mylist );
print "Values outside the function: ", mylist
```

Here, we are maintaining reference of the passed object and appending values in the same object. So, this would produce the following result −

Values inside the function: [10, 20, 30, [1, 2, 3, 4]]
Values outside the function: [10, 20, 30, [1, 2, 3, 4]]

There is one more example where argument is being passed by reference and the reference is being overwritten inside the called function.

```
#!/usr/bin/python

# Function definition is here
def changeme( mylist ):
   "This changes a passed list into this function"
   mylist = [1,2,3,4]; # This would assig new reference in mylist
   print "Values inside the function: ", mylist
   return

# Now you can call changeme function
mylist = [10,20,30];
changeme( mylist );
print "Values outside the function: ", mylist
```

The parameter *mylist* is local to the function changeme. Changing mylist within the function does not affect *mylist*. The function accomplishes nothing and finally this would produce the following result −

Values inside the function: [1, 2, 3, 4]
Values outside the function: [10, 20, 30]

Function Arguments

You can call a function by using the following types of formal arguments −

- Required arguments
- Keyword arguments
- Default arguments
- Variable-length arguments

Required arguments

Required arguments are the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition.

To call the function *printme()*, you definitely need to pass one argument, otherwise it gives a syntax error as follows −

```
#!/usr/bin/python

# Function definition is here
def printme( str ):
```

```
   "This prints a passed string into this function"
   print str
   return;

# Now you can call printme function
printme()
```

When the above code is executed, it produces the following result −

```
Traceback (most recent call last):
   File "test.py", line 11, in <module>
      printme();
TypeError: printme() takes exactly 1 argument (0 given)
```

Keyword arguments

Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.

This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters. You can also make keyword calls to the *printme()* function in the following ways −

```
#!/usr/bin/python

# Function definition is here
def printme( str ):
   "This prints a passed string into this function"
   print str
   return;
# Now you can call printme function
printme( str = "My string")
```

When the above code is executed, it produces the following result − My string

The following example gives more clear picture. Note that the order of parameters does not matter.

```
#!/usr/bin/python


# Function definition is here
def printinfo( name, age ):
   "This prints a passed info into this function"
   print "Name: ", name
   print "Age ", age
   return;

# Now you can call printinfo function
```

```
printinfo( age=50, name="miki" )
```

When the above code is executed, it produces the following result –

```
Name:  miki
Age  50
```

Default arguments

A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument. The following example gives an idea on default arguments, it prints default age if it is not passed −

```
#!/usr/bin/python

# Function definition is here
def printinfo( name, age = 35 ):
   "This prints a passed info into this function"
   print "Name: ", name
   print "Age ", age
   return;

# Now you can call printinfo function
printinfo( age=50, name="miki" )
printinfo( name="miki" )
```

When the above code is executed, it produces the following result −

```
Name:  miki
Age  50
Name:  miki
Age  35
```

Variable-length arguments

You may need to process a function for more arguments than you specified while defining the function. These arguments are called *variable-length* arguments and are not named in the function definition, unlike required and default arguments.

Syntax for a function with non-keyword variable arguments is this −

```
def functionname([formal_args,] *var_args_tuple ):
   "function_docstring"
   function_suite
   return [expression]
```

An asterisk (*) is placed before the variable name that holds the values of all nonkeyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call. Following is a simple example −

```
#!/usr/bin/python

# Function definition is here
def printinfo( arg1, *vartuple ):
   "This prints a variable passed arguments"
   print "Output is: "
   print arg1
   for var in vartuple:
      print var
   return;

# Now you can call printinfo function
printinfo( 10 )
printinfo( 70, 60, 50 )
```

When the above code is executed, it produces the following result −

```
Output is:
10
Output is:
70
60
50
```

The *Anonymous* Functions

These functions are called anonymous because they are not declared in the standard manner by using the *def* keyword. You can use the *lambda* keyword to create small anonymous functions.

- Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.
- An anonymous function cannot be a direct call to print because lambda requires an expression
- Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.
- Although it appears that lambda's are a one-line version of a function, they are not equivalent to inline statements in C or C++, whose purpose is by passing function stack allocation during invocation for performance reasons.

Syntax

The syntax of *lambda* functions contains only a single statement, which is as follows −

```
lambda [arg1 [,arg2,.....argn]]:expression
```

Following is the example to show how *lambda* form of function works −

```
#!/usr/bin/python

# Function definition is here
sum = lambda arg1, arg2: arg1 + arg2;

# Now you can call sum as a function
print "Value of total : ", sum( 10, 20 )
print "Value of total : ", sum( 20, 20 )
```

When the above code is executed, it produces the following result −

```
Value of total :  30
Value of total :  40
```

The *return* Statement

The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

All the above examples are not returning any value. You can return a value from a function as follows −

```
#!/usr/bin/python

# Function definition is here
def sum( arg1, arg2 ):
   # Add both the parameters and return them."
   total = arg1 + arg2
   print "Inside the function : ", total
   return total;

# Now you can call sum function
total = sum( 10, 20 );
print "Outside the function : ", total
```

When the above code is executed, it produces the following result −

```
Inside the function :  30
Outside the function :  30
```

Scope of Variables

All variables in a program may not be accessible at all locations in that program. This depends on where you have declared a variable.

The scope of a variable determines the portion of the program where you can access a particular identifier. There are two basic scopes of variables in Python −

- Global variables

- Local variables

Global vs. Local variables

Variables that are defined inside a function body have a local scope, and those defined outside have a global scope.

This means that local variables can be accessed only inside the function in which they are declared, whereas global variables can be accessed throughout the program body by all functions. When you call a function, the variables declared inside it are brought into scope. Following is a simple example −

```
#!/usr/bin/python

total = 0; # This is global variable.
# Function definition is here
def sum( arg1, arg2 ):
   # Add both the parameters and return them."
   total = arg1 + arg2; # Here total is local variable.
   print "Inside the function local total : ", total
   return total;

# Now you can call sum function
sum( 10, 20 );
print "Outside the function global total : ", total
```

When the above code is executed, it produces the following result −

```
Inside the function local total :  30
Outside the function global total :  0
```

MODULES

A module allows you to logically organize your Python code. Grouping related code into a module makes the code easier to understand and use. A module is a Python object with arbitrarily named attributes that you can bind and reference.

Simply, a module is a file consisting of Python code. A module can define functions, classes and variables. A module can also include runnable code.

Example

The Python code for a module named *aname* normally resides in a file named *aname.py*. Here's an example of a simple module, support.py

```
def print_func( par ):
   print "Hello : ", par
   return
```

The *import* Statement

You can use any Python source file as a module by executing an import statement in some other Python source file. The *import* has the following syntax −

import module1[, module2[,... moduleN]

When the interpreter encounters an import statement, it imports the module if the module is present in the search path. A search path is a list of directories that the interpreter searches before importing a module. For example, to import the module support.py, you need to put the following command at the top of the script −

```
#!/usr/bin/python

# Import module support
import support

# Now you can call defined function that module as follows
support.print_func("Zara")
```

When the above code is executed, it produces the following result −

Hello : Zara

A module is loaded only once, regardless of the number of times it is imported. This prevents the module execution from happening over and over again if multiple imports occur.

The *from...import* Statement

Python's *from* statement lets you import specific attributes from a module into the current namespace. The *from...import* has the following syntax −

from modname import name1[, name2[, ... nameN]]

For example, to import the function fibonacci from the module fib, use the following statement −

```
from fib import fibonacci
```

This statement does not import the entire module fib into the current namespace; it just introduces the item fibonacci from the module fib into the global symbol table of the importing module.

The *from...import* * Statement

It is also possible to import all names from a module into the current namespace by using the following import statement −

```
from modname import *
```

This provides an easy way to import all the items from a module into the current namespace; however, this statement should be used sparingly.

Locating Modules

When you import a module, the Python interpreter searches for the module in the following sequences −

- The current directory.
- If the module isn't found, Python then searches each directory in the shell variable PYTHONPATH.
- If all else fails, Python checks the default path. On UNIX, this default path is normally /usr/local/lib/python/.

The module search path is stored in the system module sys as the **sys.path** variable. The sys.path variable contains the current directory, PYTHONPATH, and the installation-dependent default.

The *PYTHONPATH* Variable

The PYTHONPATH is an environment variable, consisting of a list of directories. The syntax of PYTHONPATH is the same as that of the shell variable PATH.

Here is a typical PYTHONPATH from a Windows system −

```
set PYTHONPATH = c:\python20\lib;
```

And here is a typical PYTHONPATH from a UNIX system −

```
set PYTHONPATH = /usr/local/lib/python
```

Namespaces and Scoping

Variables are names (identifiers) that map to objects. A *namespace* is a dictionary of variable names (keys) and their corresponding objects (values).

A Python statement can access variables in a *local namespace* and in the *global namespace*. If a local and a global variable have the same name, the local variable shadows the global variable.

Each function has its own local namespace. Class methods follow the same scoping rule as ordinary functions.

Python makes educated guesses on whether variables are local or global. It assumes that any variable assigned a value in a function is local.

Therefore, in order to assign a value to a global variable within a function, you must first use the global statement.

The statement *global VarName* tells Python that VarName is a global variable. Python stops searching the local namespace for the variable.

For example, we define a variable *Money* in the global namespace. Within the function *Money*, we assign *Money* a value, therefore Python assumes *Money* as a local variable. However, we accessed the value of the local variable *Money* before setting it, so an UnboundLocalError is the result. Uncommenting the global statement fixes the problem.

```
#!/usr/bin/python

Money = 2000
def AddMoney():
   # Uncomment the following line to fix the code:
   # global Money
   Money = Money + 1

print Money
AddMoney()
print Money
```

The dir( ) Function

The dir() built-in function returns a sorted list of strings containing the names defined by a module.

The list contains the names of all the modules, variables and functions that are defined in a module. Following is a simple example −

```
#!/usr/bin/python

# Import built-in module math
import math

content = dir(math)
print content
```

When the above code is executed, it produces the following result −

['__doc__', '__file__', '__name__', 'acos', 'asin', 'atan',

'atan2', 'ceil', 'cos', 'cosh', 'degrees', 'e', 'exp',
'fabs', 'floor', 'fmod', 'frexp', 'hypot', 'ldexp', 'log',
'log10', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh',
'sqrt', 'tan', 'tanh']

Here, the special string variable *__name__* is the module's name, and *__file__* is the filename from which the module was loaded.

The globals() and locals() Functions

The *globals()* and *locals()* functions can be used to return the names in the global and local namespaces depending on the location from where they are called.

If locals() is called from within a function, it will return all the names that can be accessed locally from that function.

If globals() is called from within a function, it will return all the names that can be accessed globally from that function.

The return type of both these functions is dictionary. Therefore, names can be extracted using the keys() function.

The reload() Function

When the module is imported into a script, the code in the top-level portion of a module is executed only once.

Therefore, if you want to reexecute the top-level code in a module, you can use the *reload()* function. The reload() function imports a previously imported module again. The syntax of the reload() function is this −

reload(module_name)

Here, *module_name* is the name of the module you want to reload and not the string containing the module name. For example, to reload *hello* module, do the following −

reload(hello)

Packages in Python

A package is a hierarchical file directory structure that defines a single Python application environment that consists of modules and subpackages and sub-subpackages, and so on.

Consider a file *Pots.py* available in *Phone* directory. This file has following line of source code −

```
#!/usr/bin/python

def Pots():
   print "I'm Pots Phone"
```

Similar way, we have another two files having different functions with the same name as above −

- *Phone/Isdn.py* file having function Isdn()
- *Phone/G3.py* file having function G3()

Now, create one more file __init__.py in *Phone* directory −

- Phone/__init__.py

To make all of your functions available when you've imported Phone, you need to put explicit import statements in __init__.py as follows −

```
from Pots import Pots
from Isdn import Isdn
from G3 import G3
```

After you add these lines to __init__.py, you have all of these classes available when you import the Phone package.

```
#!/usr/bin/python

# Now import your Phone Package.
import Phone

Phone.Pots()
Phone.Isdn()
Phone.G3()
```

When the above code is executed, it produces the following result −

```
I'm Pots Phone
I'm 3G Phone
I'm ISDN Phone
```

In the above example, we have taken example of a single functions in each file, but you can keep multiple functions in your files. You can also define different Python classes in those files and then you can create your packages out of those classes.

References:

1. Hetland., “Beginning Python”, Apress, 2008

2. Mark Pilgrim, “Drive Into Python”, Apress, 2004

3. Martin C. Brown, “ Python: The Complete Reference (English)”, McGraw-Hill/Osborne Media, 2001.

4. Mark Summerfiled, “Programming in Python 3”, 2nd ed (PIP3), Addison Wesley.

5. https://www.academia.edu/41039821/Python_Tutorial_Release_3_7_0_Guido_van_Rossum_and_the_Python_development_team

UNIT - 5

PART – A

1. What is a function?
2. What is function call?
3. What is the function of raise statement? What are its two arguments?
4. How does try and execute work?
5. How do you handle the exception inside a program when you try to open a non-existent file?
6. What are modules?
7. What is a package?
8. What is the special file that each package in Python must contain?

PART – B

1. What are packages? Give an example of package creation in Python
2. Write a program to enter a number in Python and print its octal and hexadecimal equivalent.
3. What are modules in Python? Explain.
4. Explain about the import statement in modules.
5. Explain about the different types of Exceptions in Python.
6. Describe about Handling Exceptions in detail with examples.
7. Explain in detail about Python Files, its types, functions and operations that can be performed on files with examples.

i) Discuss the need and importance of function in python.

   a. Illustrate a program to exchange the value of two variables with temporary variables

8. Briefly discuss in detail about function prototyping in python. With suitable example program
9. i)Explain with an example program to return the average of its argument
   ii) Explain the various features of functions in python.