Module -1
Python Basic Concepts and Programming

- Variables, Keywords
- Statements and Expressions
- Operators, Precedence and Associativity
- Data Types, Indentation, Comments
- Reading Input, Print Output
- Type Conversions, The type( ) Function and Is Operator
- Control Flow Statements
    - The if Decision Control Flow Statement,
    - The if…else Decision Control Flow Statement
    - The if…elif…else Decision Control Statement
    - Nested if Statement
    - The while Loop
    - The for Loop
    - The continue and break Statements

  Built-In Functions, Commonly Used Modules

  Function Definition and Calling the Function

  The return Statement and void Function

  Scope and Lifetime of Variables

  Default Parameters, Keyword Arguments

  *args and **kwargs, Command Line Arguments

Why to choose python language?

There are many reasons:

- **Easy-to-learn** − Python has few keywords, simple structure, and a clearly defined syntax. This allows the student to pick up the language quickly.
- **Easy-to-read** − Python code is more clearly defined and visible to the eyes.
- **Easy-to-maintain** − Python's source code is fairly easy-to-maintain.
- **A broad standard library** − Python's bulk of the library is very portable and cross- platform compatible on UNIX, Windows, and Macintosh.
- **Interactive Mode** − Python has support for an interactive mode which allows interactive testing and debugging of snippets of code.
- **Portable** − Python can run on a wide variety of hardware platforms and has the same interface on all platforms.
- **Databases** − Python provides interfaces to all major commercial databases.
- **Scalable** − Python provides a better structure and support for large programs than other scripting languages.

Conversing with Python

- Before we can converse with Python, we must first install the Python software on the computer and learn how to start Python on computer.
- There are multiple IDEs (Integrated Development Environment) available for working with Python. Some of them are PyCharm, LiClipse, IDLE, Jupyter, Spyder etc.
- When you install Python, the IDLE editor will be available automatically. Apart from all these editors, Python program can be run on command prompt also. One has to install suitable IDE depending on their need and the Operating System they are using. Because, there are separate set of editors (IDE) available for different OS like Window, UNIX, Ubuntu, Mac, etc.
- The basic Python can be downloaded from the link: https://www.python.org/downloads/

Python has rich set of libraries for various purposes like large-scale data processing, predictive analytics, scientific computing etc. Based on one's need, the required packages can be downloaded. But there is a free open source distribution Anaconda, which simplifies package management and deployment. Hence, it is suggested for the readers to install Anaconda from the below given link, rather than just installing a simple Python.

https://anaconda.org/anaconda/python

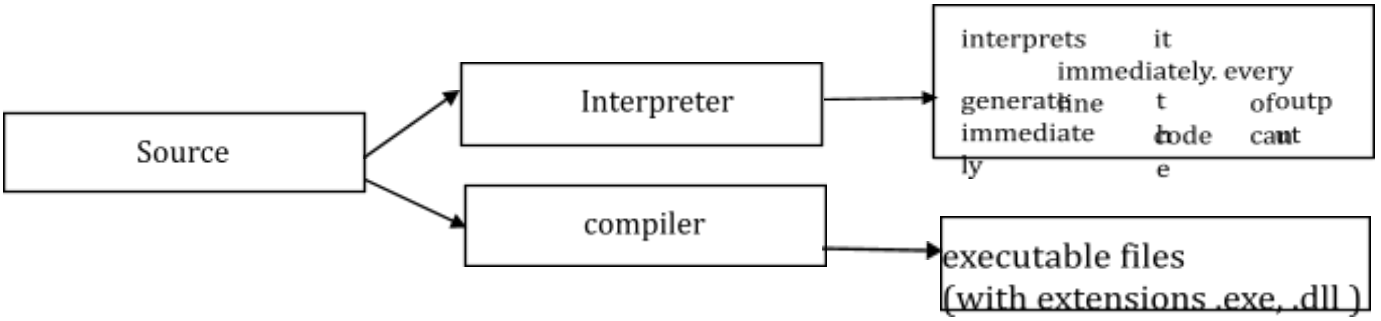
Successful installation of anaconda provides you Python in a command prompt, the default editor IDLE and also a browser-based interactive computing environment known as Jupyter notebook.

Terminology: Interpreter and compiler

- Python is a high-level language intended to be relatively straightforward for humans to read and write and for computers to read and process.
- The CPU understands a language called machine language. Machine language is very tiresome to write because it is represented all in zeros and ones:

```
00101000111010010010101000000111
11
10011000001110101001010110110 1 ...
```

Hence these high-level programming language has to be translated into machine language using translators such as : (1) interpreters and (2) compilers.



The difference between an interpreter and a compiler is given below:

Interpreter	Compiler
Translates program one statement at a time.	Scans the entire program and translates it as a whole into machine code.
It takes less amount of time to analyze the source code but the overall execution time is slower.	It takes large amount of time to analyze the source code but the overall execution time is comparatively faster.
No intermediate object code is generated, hence are memory efficient.	Generates intermediate object code which further requires linking, hence requires more memory.

Continues translating the program until the first error is met, in which case it stops. Hence debugging is easy.	It generates the error message only after scanning the whole program. Hence debugging is comparatively hard.
Programming language like Python, Ruby use interpreters.	Programming language like C, C++ use compilers.

Writing a program

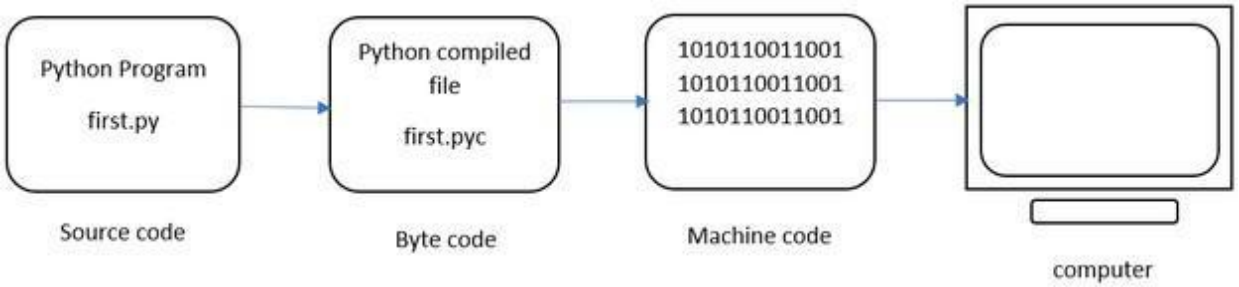
- Program can be written using a text editor.
- To write the Python instructions into a file, which is called a *script*. By convention, Python scripts have names that end with *.py* .
- To execute the script, you have to tell the Python interpreter the name of the file. In a command window, you would type python hello.py as follows:

```
$ cat hello.py
print('Hello world!')
$ python hello.py
Hello world!
```

The "$" is the operating system prompt, and the "cat hello.py" is showing us that the file "hello.py" has a one-line Python program to print a string. We call the Python interpreter and tell it to read its source code from the file "hello.py" instead of prompting us for lines of Python code

The execution of the Python program involves 2 Steps:

— Compilation
— Interpreter



Compilation

The program is converted into byte code. Byte code is a fixed set of instructions that represent arithmetic, comparison, memory operations, etc. It can run on any operating system and

hardware. The byte code instructions are created in the .pyc file. he compiler creates a directory named pycache_where it stores the .pyc file.

Interpreter

The next step involves converting the byte code (.pyc file) into machine code. This step is necessary as the computer can understand only machine code (binary code). Python Virtual Machine (PVM) first understands the operating system and processor in the computer and then converts it into machine code. Further, these machine code instructions are executed by processor and the results are displayed.

There are some low-level conceptual patterns that we use to construct programs. These constructs are not just for Python programs, they are part of every programming language from machine language up to the high-level languages. They are listed as follows:

- **Sequential execution:** Perform statements one after another in the order they are encountered in the script.
- **Conditional execution:** Check for certain conditions and then execute or skip a sequence of statements. (Ex: statements with *if-elfi-else*)
- **Repeated execution:** Perform some set of statements repeatedly, usually with some variation. (Ex: statements with *for, while* loop)
- **Reuse:** Write a set of instructions once and give them a name and then reuse those instructions as needed throughout your program. (Ex: statements in *functions*)

Identifiers

- A Python identifier is a name used to identify a variable, function, class, module or other object. An identifier starts with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores and digits (0 to 9).
- Python does not allow punctuation characters such as @, $, and % within identifiers. Python is a case sensitive programming language. Thus, Manpower and manpower are two different identifiers in Python.
- Here are naming conventions for Python identifiers
  - Class names start with an uppercase letter. All other identifiers start with a lowercase letter.
  - Starting an identifier with a single leading underscore indicates that the identifier is private.

— Starting an identifier with two leading underscores indicates a strongly private identifier.
— If the identifier also ends with two trailing underscores, the identifier is a language- defined special name.

Variables

- A variable is a named place in the memory where a programmer can store data and later retrieve the data using the variable "name"
- An assignment statement creates new variables and gives them values.
- In python, a variable need not be declared with a specific type before its usage. The type of it will be decided by the value assigned to it.

Values and types

A value is one of the basic things a program works with, like a letter or a number. Consider an example, It consists of integers and strings, floats, etc.,

```
>>>
print(4) 4
>>> message = 'And now for something completely different'
>>> n = 17
>>> pi = 3.1415926535897931
```

- The above example makes three assignments.
  — The first assigns a string to a new variable named message.
  — The second assigns the integer 17 to n
  — The third assigns the (approximate) value of π to pi.

- To display the value of a variable, you can use a print statement:

```
>>> print(n)
17
>>> print(pi)
3.141592653589793
```

- The type of a variable is the type of the value it refers to above example

```
>>>
type(message)                  #type refers to
<class 'str'>                  string
>>> type(n)                    #type refers to
<class 'int'>                  integer
>>> type(pi)
<class 'float'>                #type refers to float
```

- **Rules to follow when naming the variables**.
    - — Variable names can contain letters, numbers, and the underscore.
    - — Variable names cannot contain spaces and other special characters.
    - — Variable names cannot start with a number.
    - — Case matters—for instance, temp and Temp are different.
    - — Keywords cannot be used as a variable name.

- Example of valid variable names are: Spam, eggs, spam23 , _speed
- Variable names can be arbitrarily long. In this case ,underscore character (_) can appear in a name. Ex: *my_first_variable* .
- As Python is case-sensitive, variable name *sample* is different from *SAMPLE* .
- Variable names can start with an underscore character, but we generally avoid doing this unless we are writing library code for others to use.
- Examples for invalid variable names are list below

```
>>> 76trombones = 'big parade'  # illegal because it begins with a number
SyntaxError: invalid syntax


>>> more@ = 1000000             # illegal because it contains an illegal character, @.
SyntaxError: invalid
syntax

>>> class = 'Advanced'          # class is one of Python's keywords
SyntaxError: invalid
syntax
```

Keywords

Keywords are a list of reserved words that have predefined meaning. Keywords are special vocabulary and cannot be used by programmers as identifiers for variables, functions, constants or with any identifier name. Attempting to use a keyword as an identifier name will cause an error. The following table shows the Python keywords.

and	del	from	None	True	def	lambda
as	elif	global	nonlocal	Try	raise	return
assert	else	if	not	While	finally	
break	except	import	or	with	is	
class	False	in	pass	yield	continue	

Statement

- A statement is a unit of code that the Python interpreter can execute.
- We have seen two kinds of statements:
  — *assignment statement*: We assign a value to a variable using the assignment statement (=). An assignment statement consists of an expression on the right-hand side and a variable to store the result. In python ,there is special feature for

  multiple assignments, where more than one variable can be initialized in single statement.

```
Ex: str="google"
    x = 20+y
    a, b, c = 2, "B", 3.5
```

print statement *:* print is a function which takes string or variable as a argument to display it on the screen.

Following are the examples of statements –

```
>>> x=5                          #assignment statement
>>> x=5+3                             #assignment
>>>                                   statement
print(x)                        #printing statement
```

Optional arguments with print statement:

sep : Python will insert a space between each of the arguments of the print function. There is an optional argument called *sep*, short for separator, that you can use to change that space to something else. For example, using *sep*=':' would separate the arguments by a colon and *sep*='##' would separate the arguments by two pound signs.

```
>>>
print("a","b","c","d",sep=";")
a;b;c;d
```

end : The print function will automatically advance to the next line. For instance, the following will print on two lines:

code	output

`print("A")`
`print("B")`
`print("C", end=" ")`
`print("E")`
	A
	B
	C E

Expressions

An expression is a combination of values, variables, and operators. A value all by itself is considered an expression, and so is a variable, so the following are all legal expressions. If you type an expression in interactive mode, the interpreter evaluates it and displays the result:

```
>>> x=5
>>>  x+1
6
```

Operators are special symbols that represent computations like addition and multiplication. The values the operator is applied to are called operands.

Here is list of *arithmetic operators*

Operator	Meaning	Example
+	Addition	Sum= a+b
-	Subtraction	Diff= a-b
/	Division	a=2 b=3 div=a/ b (div will get a value 1.3333333)
//	Floor Division – returns only integral part of qotient after division	F = a//b A= 4//3 (X will get a value 1)
%	Modulus – remainder after Division	A= a %b (Remainder after dividing a by b)
**	Exponent	E = x** y (means x to the power of y)

Relational or *Comparison* Operators: are used to check the relationship (like less than, greater than etc) between two operands. These operators return a Boolean value either True or False.

Assignment Operators: Apart from simple assignment operator = which is used for assigning values to variables, Python provides compound assignment operators.

For example,

statements	Compound statement
x=x+y	x+=y
y=y//2	y//=2

Logical Operators: The logical operators *and, or, not* are used for comparing or negating the logical values of their operands and to return the resulting logical value. The values of the operands on which the logical operators operate evaluate to either True or False. The result of the logical operator is always a Boolean value, True or False.

Operators	Statement	Comments	Example
And	x> 0 and x<10	Is true only if x is greater than 0 and less than10	**Ex1 :**>>> x=5 >>> x>0 and x<10 True **Ex2:** >>> x= -5 >>> x>0 and x<10 False
Or	n%2==0 or n%3==0	Is true only if either condition is true	>>> n=2 >>>n%2==or n%3==0 True

Not	not(x>y)	negates Boolean expression	>>> x=5
			>>> x> 0 and x<10
			True
			>>> not x
			False

Precedence and Associativity (Order of operations)

- When an expression contains more than one operator, the evaluation of operators depends on the precedence of operators.
- The Python operators follow the precedence rule (which can be remembered as PEMDAS) as given below :

— *Parenthesis* have the highest precedence in any expression. The operations within parenthesis will be evaluated first.

— *Exponentiation* has the 2nd precedence. But, it is right associative. That is, if there are two exponentiation operations continuously, it will be evaluated from right to left (unlike most of other operators which are evaluated from left to right). For example

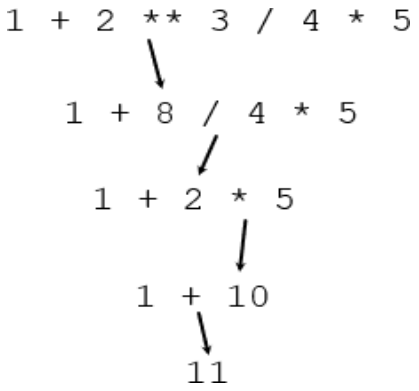
```
>>> print(2**3**2)          #It is 512 i.e., 2^(3^2)
```

— *Multiplication* and *Division* are the next priority. Out of these two operations, whichever comes first in the expression is evaluated.

```
>>> print(5*2/4) #multiplication and then division
2.5
>>> print(5/4*2) #division and then multiplication 2.5
```

— *Addition* and *Subtraction* are the least priority. Out of these two operations, whichever appears first in the expression is evaluated i.e., they are evaluated from left to right .

Example : **x = 1 + 2 ** 3 / 4 * 5**



Data Types

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

Basic data types of Python are

- Numbers
- Boolean
- Strings
- list
- tuple
- dictionary
- None

Numbers

Integers, floating point numbers and complex numbers fall under Python numbers category. They are defined as int, float and complex class in Python. Integers can be of any length; it is only limited by the memory available. A floating-point number is accurate up to 15 decimal places. Integer and floating points are separated by decimal points. 1 is an integer, 1.0 is floating point number. Complex numbers are written in the form, x + yj, where x is the real part and y is the imaginary part.

Boolean

Booleans may not seem very useful at first, but they are essential when you start using conditional statements. Boolean value is, either True or False. The Boolean values, True and False are treated as reserved words.

Strings

A string consists of a sequence of one or more characters, which can include letters, numbers, and other types of characters. A string can also contain spaces. You can use single quotes or double quotes to represent strings and it is also called a string literal. Multiline strings can be denoted using triple quotes, ''' or " " ". These are fixed values, not variables that you literally provide in your script.

For example,

1. >>> s = 'This is single quote string'
2. >>> s = "This is double quote string"
3. >>> s = '''This
   is Multiline
   string'''

List

A list is formed(or created) by placing all the items (elements) inside square brackets [ ], separated by commas.It can have any number of items and they may or may not be of different types (integer, float, string, etc.).

Example : List1 = [3,8,7.2,"Hello"]

Tuple

A tuple is defined as an ordered collection of Python objects. The only difference between tuple and list is that tuples are immutable i.e. tuples can't be modified after it's created. It is represented by tuple class. we can represent tuples using parentheses ( ).

Example: Tuple = (25,10,12.5,"Hello")

Dictionary

Dictionary is an unordered collection of data values, which is used to store data values like a map, which, unlike other Data Types that hold only a single value as an element, a Dictionary consists of key-value pair. Key-value is provided within the dictionary to form it more optimized. In the representation of a dictionary data type, each key-value pair during a Dictionary is separated by a colon: whereas each key's separated by a 'comma'.

Example: Dict1 = {1 : 'Hello' , 2 : 5.5, 3 : 'World' }

None

None is another special data type in Python. None is frequently used to represent the absence of a value. For example, >>> money = None

Indentation

In Python, Programs get structured through indentation (FIGURE below)

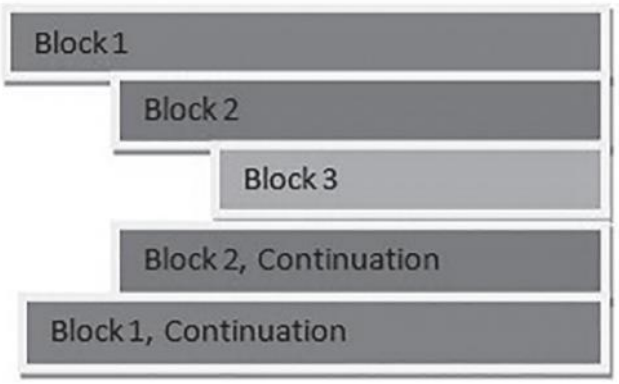


Figure : Code blocks and indentation in Python.

Usually, we expect indentation from any program code, but in Python it is a requirement and not a matter of style. This principle makes the code look cleaner and easier to understand and read.

Any statements written under another statement with the same indentation is interpreted to belong to the same code block. If there is a next statement with less indentation to the left, then it just means the end of the previous code block.

In other words, if a code block has to be deeply nested, then the nested statements need to be indented further to the right. In the above diagram, Block 2 and Block 3 are nested under Block 1. Usually, four whitespaces are used for indentation and are preferred over tabs. Incorrect indentation will result in Indentation Error.

Comments

As programs get bigger and more complicated, they get more difficult to read. Formal programming languages are many, and it is often difficult to look at a piece of code and figure out what it is doing, or why.

For this reason, it is a good idea to add notes to your programs to explain in natural language what the program is doing. These notes are called comments, and in Python they start with the # symbol:

```
Ex1. #This is a single-line comment
Ex2. ''' This is a
multiline
comment '''
```

Comments are most useful when they document non-obvious features of the code. It is reasonable to assume that the reader can figure out *what* the code does; it is much more useful to explain *why*.

Reading Input, Print Output

Reading Input

Python provides a built-in function called *input* that gets input from the keyboard. When this function is called, the program waits for the user input. When the user press the Enter key, the program resumes and input returns user value as a string.

For example

```
>>> inp = input()
Welcome to world of python
>>> print(inp)
Welcome to world of python
```

It is a good idea to have a prompt message telling the user about what to enter as a value. You can pass that prompt message as an argument to input function.

```
>>>x=input('Please enter some
text:\n') Please enter some text:
Roopa
>>>
print(x)
Roopa
```

The sequence \n at the end of the prompt represents a newline, which is a special character that causes a line break. That's why the user's input appears below the prompt.

If you expect the user to type an integer, you can try to convert the return value to int using the int() function:

Example1:

```
>>> prompt = 'How many days in a week?\n'
>>> days = input(prompt)
 How many days in a week?
 7
>>> type(days)
<class 'str'>

#by default value is treated as string
```

Example 2:

```
>>> x=int(input('enter number\n'))
 enter number
 12
>>> type(x)
     <class 'int'>
```

Print Output

Format operator

- The *format operator*, % allows us to construct strings, replacing parts of the strings with the data stored in variables.
- When applied to integers, % is the modulus operator. But when the first operand is a string, % is the format operator.
- For example, the format sequence "%d" means that the operand should be formatted as an integer (d stands for "decimal"):

  Example 1:

```
>>> camels = 42
>>>'%d' % camels
'42'
```

- A format sequence can appear anywhere in the string, so you can embed a value in a sentence:

  Example 2 :

```
>>> camels = 42
>>> 'I have spotted %d camels.' % camels
'I have spotted 42 camels.'
```

- If there is more than one format sequence in the string, the second argument has to be a tuple. Each format sequence is matched with an element of the tuple, in order.
- The following example uses "%d" to format an integer, "%g" to format a floating point number , and "%s" to format a string:

  Example 3:

```
>>> 'In %d years I have spotted %g %s.' % (3, 0.1, 'camels')
'In 3 years I have spotted 0.1 camels.'
```

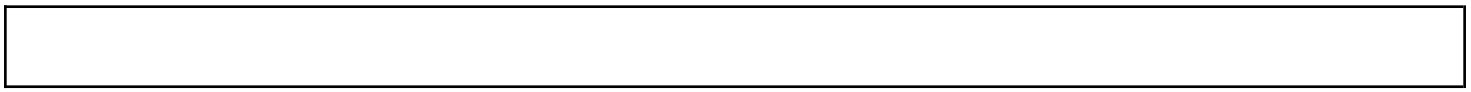
Format function

format() : is one of the string formatting methods in Python3, which allows multiple substitutions and value formatting. This method lets us concatenate elements within a string through positional formatting.

Two types of Parameters:

- positional_argument
- keyword_argument

- Positional argument: It can be integers, floating point numeric constants, strings, characters and even variables.



- Keyword argument : They is essentially a variable storing some value, which is passed as parameter.

 # To demonstrate the use of formatters with positional key arguments.

Positional arguments are placed in order	>>>print("{0} department college{1} ".format("RNSIT","EC"))
	RNSIT college EC department
Reverse the index numbers with the parameters of the placeholders	>>>print("{1} department {0} college ".format("RNSIT","EC"))
	EC department RNSIT college
Positional arguments are not specified. By default it starts positioning from zero	>>>print("Every {} should know the use of {} {} python programming and {}".format("programmer", "Open", "Source", "Operating Systems"))
	Every programmer should know the use of Open Source programming and Operating Systems
Use the index numbers of the values to change the order that they appear in the string	>>>print("Every {3} should know the use of {2} {1} programming and {0}" .format("programmer", "Open", "Source", "Operating Systems"))
	Every Operating Systems should know the use of Source Open programming and programmer
Keyword arguments are called by their keyword name	print("EC department {0} 'D' section {college}" .format("6", college="RNSIT"))
	EC department 6 'D' section RNSIT

f-strings

Formatted strings or f-strings were introduced in Python 3.6. A f-string is a string literal that is prefixed with "f". These strings may contain replacement fields, which are expressions enclosed within curly braces {}. The expressions are replaced with their values.

Example : >>>a=10

```
>>>print(f"the value is {a}")
the value is 10
```

Type conversion functions

Python also provides built-in functions that convert values from one type to another.

Table : Type conversion functions

DataType	Example
int()	**Ex:1**>>> int('32') 32 **Ex:2**>>> int('Hello') ValueError: invalid literal **for** int() **with** base 10: 'Hello' **Ex:3**>>>int(3.9999) 3 **Ex:4**>>> int(-2.3) -2
float()	Ex1 : >>>float(32) 32.0 Ex2 : float('3.124') 3.124
str()	Ex1 : >>>str(32) '32' Ex2 : >>> str(3.124) '13.124'

- int can convert floating-point values to integers, but it doesn't round off; it chops off the fraction part.
- float converts integers and strings to floating-point numbers.
- str converts its argument to a string.

The type( ) Function

type function is called to know the datatype of the value. The expression in parenthesis is called the argument of the function. The argument is a value or variable that we are passing into the function as input to the function.

Example: >>> type(33)

<class 'int'>

Is Operator

- If we run these assignment statements:

```
a =
'banana' b
= 'banana'
```

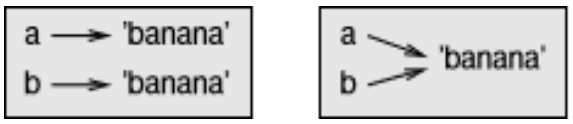


Figure: (a)

- We know that a and b both refer to a string, but we don't know whether they refer to the same string. There are two possible states, shown in Figure (a).
- In one case, a and b refer to two different objects that have the same value. In the second case, they refer to the same object. That is, a is an alias name for b and viceversa. In other words, these two are referring to same memory location.
- To check whether two variables refer to the same object, you can use the ***is* operator.**

```
>>> a = 'banana'
>>> b = 'banana'
>>> a is b
True
```

- When two variables are referring to same object, they are called as ***identical*** objects.
- When two variables are referring to different objects, but contain a same value, they are known as ***equivalent*** objects.

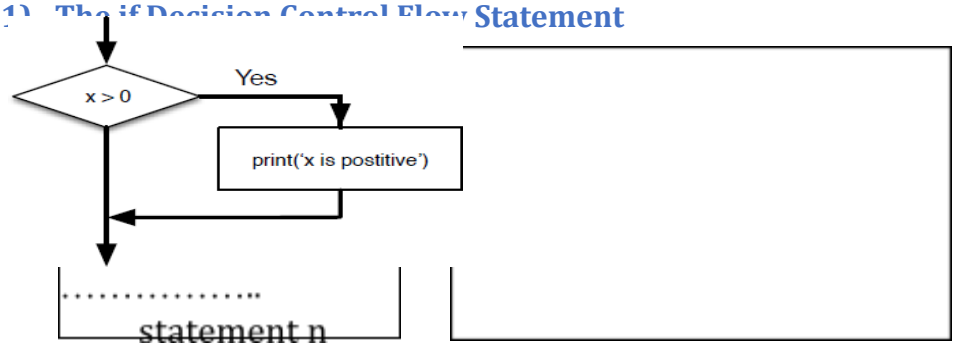
```
>>>s1=input("Enter a
string:")
>>>s2= input("Enter a
string:")
>>>s1 is s2                    #check s1 and s2 are identical
False
>>>s1 == s2                    #check s1 and s2 are equivalent
True
```

- Here s1 and s2 are equivalent, but not identical

- If two objects are identical, they are also equivalent, but if they are equivalent, they are not necessarily identical.

A conditional statement gives the developer to ability to check conditions and change the behaviour of the program accordingly. The simplest form is the if statement:

1) The if Decision Control Flow Statement



- The Boolean expression after the if keyword is called the *condition*.
- The if statement consists of a header line that ends with the colon character (:) followed by an indented block. Statements like this are called compound statements because they stretch across more than one line.
- If the logical condition is true, then the block of statements get executed. If the logical condition is false, the indented block is skipped.

2) The if…else Decision Control Flow Statement (*alternative execution)*

A second form of the if statement is *alternative execution*, in which there are two possibilities and the condition determines which one gets executed is as shown in flowchart below.

The syntax looks like this:

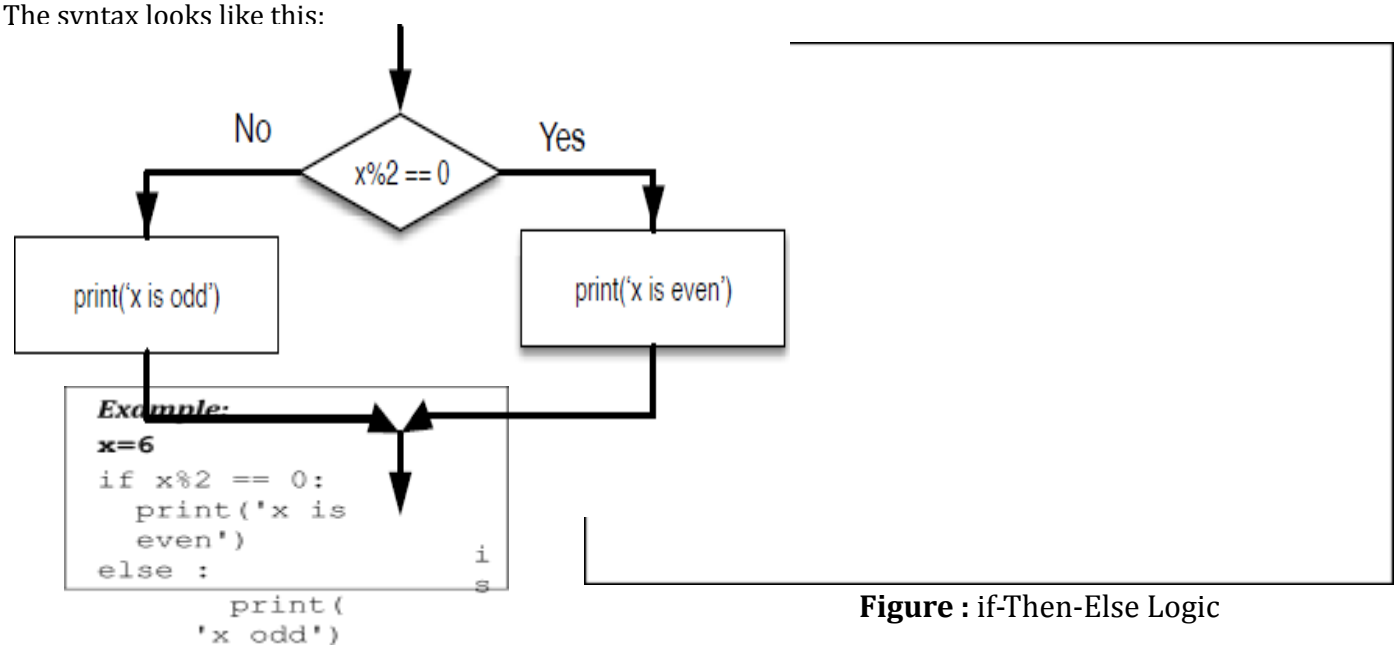


Figure : if-Then-Else Logic

3) The if…elif…else Decision Control Statement (*Chained conditionals*)

If there are more than two possibilities and we need more than two branches. One way to express a computation like that is a *chained conditional*. elif is an abbreviation of "else if." Again, exactly one branch will be executed is as shown in flowchart below.

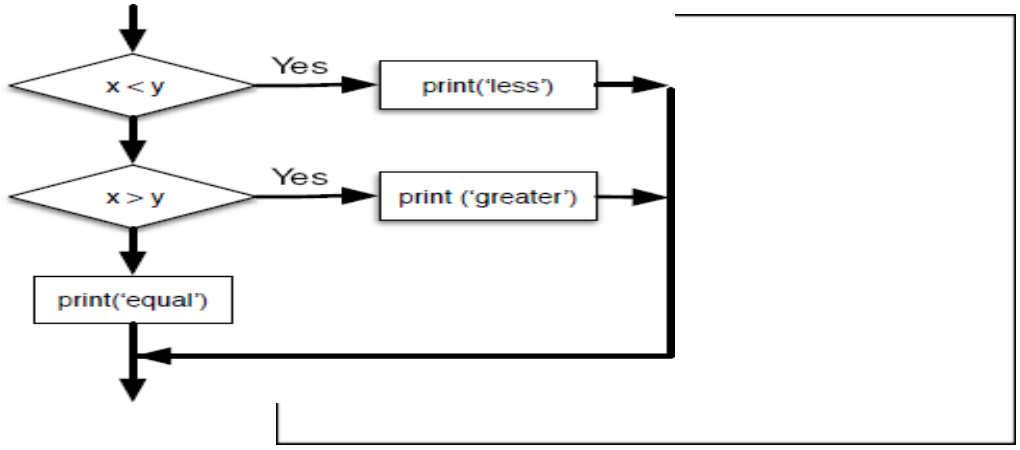


Figure : If-Then-ElseIf Logic

Syntax	**Example**

if *condition1:*
Statement elif
condition2:
Statement
.....
elif *condition_n:*
Statement
else:
Statement
	`x=1`
	`y=6`
	`if x < y:`
	`print('x is less than y')`
	`elif x > y:`
	`print('x is greater than y')`
	`else:`
	`print('x and y are equal')`
	
	`Output:`
	`X is less than y`

4) Nested if Statement

- The conditional statements can be nested. That is, one set of conditional statements can be nested inside the other.
- Let us consider an example, the outer conditional statement contains two branches.

Example	**Output**:

x=3
y=4
if x == y:
print('x and y are equal')
else:
if x < y:
print('x is less than y')
else:
print('x is greater than y')
	x is less than y

— The first branch contains a simple statement.
— The second branch contains another if statement, which has two branches of its own.
Those two branches are both simple statements, although they could have been conditional statements as well is as shown in flowchart below.

- Nested conditionals make the code difficult to read, even though there are proper indentations. Hence, it is advised to use logical operators like and to simplify the nested conditionals.

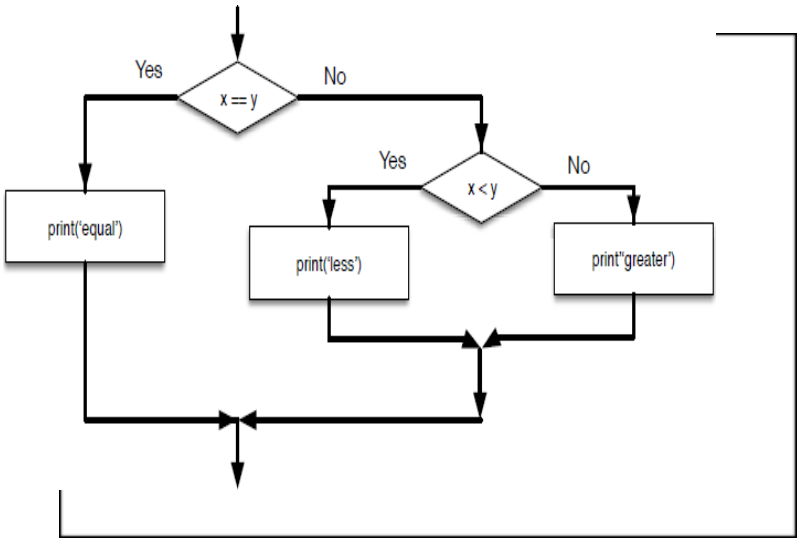


Figure : Nested If Statements

Short-circuit evaluation of logical expressions

- When Python is processing a logical expression such as

  x >= 2 and (x/y) > 2

  it evaluates the expression from left to right. Let's assume x=1. Because of the definition of *and* operator, if x is less than 2, the expression x >= 2 is False and so the whole expression is False regardless of whether (x/y) > 2 evaluates to True or False.

- If Python detects that there is nothing to be gained by evaluating the rest of a logical expression, it stops its evaluation and does not do the computations in the rest of the logical expression. When the evaluation of a logical expression stops because the overall value is already known, it is called ***short-circuiting the evaluation***.

- However ,if the first part of logical expression results in True, then the second part has to be evaluated to know the overall result. The short-circuiting not only saves the computational time, but it also leads to a technique known as ***guardian pattern***.

Consider the below examples:

Example1	Example 2	Example 3

>>> x = 6
>>> y = 2
>>> x >= 2 and (x/y) > 2
True
	>>> x = 1
	>>> y = 0
	>>> x >= 2 and (x/y) > 2
	False
		>>> x = 6
		>>> y = 0
		>>> x >= 2 and (x/y) > 2
		Traceback (most recent call last):
		ZeroDivisionError: division by zero

The first example is true because both conditions are true.

But the second example did *not* fail because the first part of the expression

x >= 2 evaluated to False so the (x/y) was not ever executed due to the *short-circuit rule* and there was no error .

The third calculation failed because Python was evaluating (x/y) and y was zero, which causes a runtime error.

We can construct the logical expression to strategically place a *guard* evaluation just before the evaluation that might cause an error as follows:

Consider an example,

Example 1:

>>> x = 1
>>> y = 0
>>> x >= 2 and y != 0 and (x/y) > 2
False

Example 2

>>> x = 6
>>> y = 0
>>> x >= 2 and y != 0 and (x/y) > 2
False

Example 3
>>> x >= 2 and (x/y) > 2 and y != 0 Traceback (most recent call last): File "<stdin>", line 1, in <module> ZeroDivisionError: division by zero

In the first logical expression, x >= 2 is False so the evaluation stops at first condition itself.

In the second logical expression, x >= 2 is True but y != 0 is False so it never reach the condition (x/y).

In the third logical expression, the y!= 0 is placed *after* the (x/y)>2 condition so the expression fails with an error.

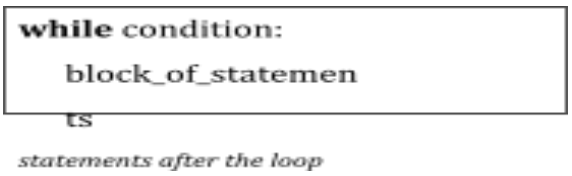
In the second expression, we say that y != 0 acts as a ***guard*** to insure that we only execute (x/y) if y is non-zero.

ITERATION (looping statements)

Iteration repeats the execution of a sequence of code. Iteration is useful for solving many programming problems. Iteration and conditional execution form the basis for algorithm construction.

The while statement

Sometimes, though, we need to repeat something, but we don't know ahead of time exactly how many times it has to be repeated i.e., indefinite loop. This is a situation when 'while loop' is used.



The syntax of while loop :

Here, while is a keyword

The flow of execution for a *while* statement is as below :

- The condition is evaluated first, yielding True or False
- If the condition is false, the loop is terminated and statements after the loop will be executed.
- If the condition is true, the body of the loop (indented block of statements)will be executed and then goes back to condition evaluation.

Example: program to print values 1 to 5

`i = 1`
`while i<=5:`
`  print(i)`
`  i= i+1`
`print(" printing is done")`
	Output:
	1
	2
	3
	4
	5
	printing is done

In the above example, variable *i* is initialized to 1. Then the condition i <=5 is being checked. If the condition is true, the block of code containing print statement print(i) and increment statement (n=n+1) are executed. After these two lines, condition is checked again. The procedure continues till condition becomes false, that is when n becomes 6. Now, the while- loop is terminated and next statement after the loop will be executed. Thus, in this example,

the loop is iterated for 5 times.

Also notice that, variable *i* is initialized before starting the loop and it is incremented inside the loop. Such a variable that changes its value for every iteration and controls the total execution of the loop is called as *iteration variable* or *counter variable*. If the count variable is not updated properly within the loop, then the loop may enter into infinite loop.

Infinite loops with *break*

Infinite loops are the looping statements which iterates infinite number of times where the condition remains true always.

```
n = 10
while True:
    print(n, end=' ')
    n = n - 1
print('Done!')
```

Let us consider an example:

Here, the condition is always True, which will never terminate the loop. Sometimes, the condition is given such a way that it will never become false and hence by restricting the program control to go out of the loop. This situation may happen either due to wrong condition or due to not updating the *counter* variable.

Hence to overcome this situation, ***break*** statement is used. The *break* statement can be used to break out of a *for* or *while* loop before the loop is finished.

Here is a program that allows the user to enter up to 10 numbers. The user can stop early by entering a negative number.

```
while True:
    num = eval(input('Enter a number: '))
    if num<0:
        break
    print(num)
```

Output:

```
Enter a number: 23 23
Enter a number: 34 34
Enter a number: 56 56
Enter a number: -12
```

In the above example, observe that the condition is kept inside the loop such a way that, if the user input is a negative number, the loop terminates. This indicates that, the loop may terminate with just one iteration (if user gives negative number for the very first time) or it may take thousands of iteration (if user keeps on giving only positive numbers as input). Hence, the number of iterations here is unpredictable. But we are

making sure that it will not

be an infinite-loop, instead, the user has control on the loop.

Another example for usage of while with *break* statement: program to take the input from the user and echo it until they type done:

```
while True:
    line = input('>')
    if line == 'done':
        break
    print(line)
print('Done!')
```

Output:

```
>       hello
 there hello
 there
>       finis
 hed
 finished
>       d
one
Done!
```

Finishing iterations with *continue*

Sometimes the user may want to skip few tasks in the loop based on the condition. To do so *continue* statement can be used.

the continue statement skips to the next iteration without finishing the body of the loop for the current iteration.

Here is an example of a loop that copies its input until the user types “done” but treats lines that start with the hash character as lines not to be printed (kind of like Python comments).

```
while True:
    line = input('>')
    if line[0] == '#':
        continue
    if line == 'done':
        break
   print(line)
print('Done!')
```

Output:

```
>       hello
there hello
there
> # don't print this
>       print
this! print
this!
> done
Done!
```

All the lines are printed except the one that starts with the hash sign because when the continue is executed, it ends the current iteration and jumps back to the while statement to start the next iteration, thus skipping the print statement.

Definite loops using for

Sometimes we want to loop through a set of things such as a list of words, the lines in a file, or a list of numbers. When we have a list of things to loop through, we can construct a definite loop using a *for* statement.

for statement loops through a known set of items so it runs through as many iterations as there are items in the set.

There are two versions in for loop:

- *for loop with sequence*
- *for loop with range( ) function*

```
for var in list/sequence :
    statements to be repeated
```

Syntax of ***for* loop with sequence**:

Here,

for and *in* -> are keywords

list/sequence -> is a set of elements on which the loop is iterated. That is, the loop will be executed till there is an element in list/sequence

statements -> constitutes body of the loop.

- Example :

```
friends = ['Roopa', 'Smaya', 'Vikas']
for name in friends:
    print('Happy New Year:', name)
print('Done!')
```

Output:

```
Happy New Year: Roopa
Happy New Year: Smaya
Happy New Year: Vikas
Done!
```

—In the example, the variable *friends* is a list of three strings and the for loop goes through the list and executes the body once for each of the three strings in the list.

—*name* is the iteration variable for the *for* loop. The variable *name* changes for each iteration of the loop and controls when the *for* loop completes. The iteration variable steps successively through the three strings stored in the *friends* variable.

- The *for* loop can be used to print (or extract) all the characters in a string as shown below :

```
for i in "Hello":
    print(i, end="\t")
```

Output:

```
H   e   l   l   o
```

- Syntax of ***for* loop with range( ) function:**

```
for variable in range( start, end,
                steps): statements to be
                repeated
```

The start and end indicates starting and ending values in the sequence, where end is excluded in the sequence (That is, sequence is up to end-1). The default value of start is 0.

The argument steps indicates the increment/decrement in the values of sequence with the default value as 1. Hence, the argument steps is optional. Let us consider few examples on usage of range() function.

EX:1 Program code to print the message multiple times

```
for i in range(3):
     print('Hello’)
```

```
Hello
Hello
Hello
```

EX:2 Program code to print the numbers in sequence. Here iteration variable *i* takes the value from 0 to 4 excluding 5. In each iteration value of *i* is printed.

```
for i in range(5):
    print(i, end= “\t”)
```

```
0    1    2    3    4
```

EX:3 Program to allow the user to find squares of any three number

```
for i in range(3):
   num = int(input('Enter a number:‘))
('The square of your number is',
num*num)
print('The loop is now done.')
```

```
Enter a number: 3
The square of your number is 9
Enter a number: 4
The square of your number is 16
Enter a number: 56
The square of your number is 3136
The loop is now done.
```

EX:4 Program that counts down from 5 and then prints a message.

```
for i in range(5,0,-1):
    print(i)
print('Blast off!!')
```

```
5
4
3
2
1
Blast off !!
```

Functions

- A sequence of instructions intended to perform a specific independent task is known as a *function*.
- You can pass data to be processed, as parameters to the function. Some functions can return data as a result.
- In Python, all functions are treated as objects, so it is more flexible compared to other high- level languages.
- In this section, we will discuss various types of built-in functions, user-defined functions, applications/uses of functions etc.

Function types

Built in functions

User defined functions

Built-in functions

Python provides a number of important built in functions that we can use without needing to provide the function definition.

Built-in functions are ready to use functions.

The general form of built-in functions: **function_name(arguments)**

An argument is an expression that appears between the parentheses of a function call and each argument is separated by comma .

Table : Built in functions

Built–in- functions	**Syntax/Examples**	**Comments**
max( )	>>>max('hello world') 'w'	display character having maximum ASCII code
min( )	>>>min('hello world') ' '	display least character having minimum ASCII code
len( )	>>>len('hello world') 11	display length of string
round( )	>>>round(3.8) 4 >>>round(3.3) 3 >>>round(3.5) 4 >>>round(3.141592653, 2) 3.14	round the value with single argument. round the value with 2 arguments.
pow()	>>>pow(2,4)	2^4
abs()	>>>abs(-3.2)	returns the absolute value of object

Commonly Used Modules

Math functions

Python has a math module that provides most of the frequently used mathematical functions. Before we use those functions , we need to import the module as below:

```
>>> import math
```

This statement creates a module object named math. If we pass module object as an argument to print, information about that object is displayed:

```
>>> print(math)
<module 'math' (built-in)>
```

Table : Math Functions

Examples	Comments

`>>> import math`
`>>> math.sqrt(3)`
`1.7320508075688772`
	`# Finds the square root of number`

`>>> math.sin(30)` `-0.9880316240928618`	`# Finds the sin of 30`
`>>> math.cos(30)` `0.15425144988758405`	`# Finds the cos of 30`
`>>> print(math.pi)` `3.141592653589793`	`# print the value of pi`
`>>> math.sqrt(2)` `1.4142135623730951`	`# finds the square root of 2`
`>>> math.log(2)` `0.6931471805599453`	`#finds the log base e`

Random numbers

- Most of the programs that we write take predefined input values and produces expected output values. such programs are said to be *deterministic*. Determinism is usually a good thing, since we expect the same calculation to yield the same result.
- But it is not case always, for some applications, we want the computer to be unpredictable. Games are an obvious example, but there are many applications.
- Making a program truly nondeterministic turns out to be not so easy, but there are ways to make it at least seem nondeterministic. One of them is to use *algorithms* that generate *pseudorandom* numbers.
- The function random returns a random float between 0.0 and 1.0 and for integer between (1 and 100 etc) .
- Python has a module called *random*, in which functions related to random numbers are available.

- To generate random numbers. Consider an example program to use **random()** function which generates random number between 0.0 and 1.0 ,but not including 1.0. In the below program, it generates 5 random numbers

import random **for** i in range(5): x = random.random() print(x)	**Output:** 0.11132867921152356 0.5950949227890241 0.04820265884996877 0.841003109276478 0.997914947094958

- The function **randint()** takes the parameters low and high, and returns an integer between low and high (including both).

```
>>> import random
>>>
random.randint(5,10)
10
>>>
random.randint(5,10) 6
>>>
random.randint(5,10) 7
```

To choose an element from a sequence at random, you can use **choice()**:

```
>>> t = [1, 2, 3]
>>> random.choice(t)
 2
>>> random.choice(t)
3
```

Function Definition and Calling the Function

- Python facilitates programmer to define his/her own functions.
- The function written once can be used wherever and whenever required.
- The syntax of user-defined function would be:

def fname(arg_list): statement_1 statement_2 …… statement_n return value	Here, ***def*** : is a keyword indicating it as a function definition. ***fname*** **:** is any valid name given to the function. ***arg_list*** **:** is list of arguments taken by a function. These are treated as inputs to the function from the position of function call. There may be zero or more arguments to a function. ***statements*** : are the list of instructions to perform required task. ***return*** **:** is a keyword used to return the output value. This statement is optional

- The first line in the function ***def fname(arg_list)*** is known as function header/definition. The remaining lines constitute function body.
- The function header is terminated by a colon and the function body must be indented.
- To come out of the function, indentation must be terminated.
- Unlike few other programming languages like C, C++ etc, there is no main() function or specific location where a user-defined function has to be called.
- The programmer has to invoke (call) the function wherever required.
- Consider a simple example of user-defined function –

```
def myfun():
    print("Hello everyone")
    print("this is my own function")

print("before calling the
function") myfun()#fuction call
print("after calling the function")
```

Output:

```
before calling the function
Hello everyone
this is my own function
after calling the function
```

Function calls

- A function is a named sequence of instructions for performing a task.
- When we define a function we will give a valid name to it, and then specify the instructions for performing required task. Then, whenever we want to do that task, a function is called by its name.

  Consider an example,

```
>>> type(33)
<class 'int'>
```

- Here, *type* function is called to know the datatype of the value. The expression in parenthesis is called the argument of the function. The argument is a value or variable that we are passing into the function as input to the function.
- It is common to say that a function "takes" an argument and "returns" a result. The result is called the return value.

The return Statement and void Function

A function that performs some task, but do not return any value to the calling function is known as ***void*** function. The examples of user-defined functions considered till now are void functions.

The function which returns some result to the calling function after performing a task is known as ***fruitful*** function. The built-in functions like mathematical functions, random

number generating functions etc. that have been considered earlier are examples for fruitful functions.

One can write a user-defined function so as to return a value to the calling function as shown in the following example.

```
def addition(a,b):          #function
      definition sum=a + b
      return sum

x=addition(3, 3)       #function
call print("addition of 2
numbers:",x)
```

Output:

```
addition of 2 numbers:6
```

— In the above example, The function addition() take two arguments and returns their sum to the receiving variable x.

— When a function returns something and if it is not received using a some variable, the return value will not be available later.

- When we are using built –in functions, that yield results are fruitful functions.

```
>>>math.sqrt(2)
1.7320508075688772
```

- The void function might display something on the screen or has some other effect. they perform an action but they don't have return value. Consider an example

```
>>> result=print('python')
python
>>> print(result)
None
>>> print(type(None))
<class 'NoneType'>
```

Scope and Lifetime 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.

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

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

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

Default Parameters

- For some functions, you may want to make some parameters optional and use default values in case the user does not want to provide values for them. This is done with the help of default argument values.

- Default argument values can be specified for parameters by appending to the parameter name in the function definition the assignment operator (=) followed by the default value.

- Note that the default argument value should be a constant.
- Only those parameters which are at the end of the parameter list can be given default argument values i.e. you cannot have a parameter with a default argument value preceding a parameter without a default argument value in the function's parameter list. This is because the values are assigned to the parameters by position.

  For example, **def func(a, b=5)** is valid, but **def func(a=5, b)** is not valid.

```
def say(message, times=1):
            print(message * times)

say('Hello')
say('World', 5)
```

Output :

```
Hello
WorldWorldWorldWorldWorld
```

How It Works

The function named *say* is used to print a string as many times as specified. If we don't supply a value, then by default, the string is printed just once. We achieve this by specifying a default argument value of 1 to the parameter times . In the first usage of *say* , we supply only the string and it prints the string once. In the second usage of *say* , we supply both the string and an argument 5 stating that we want to *say* the string message 5 times.

Keyword Arguments

If you have some functions with many parameters and you want to specify only some of them, then you can give values for such parameters by naming them - this is called ***keyword arguments*** - we use the name (keyword) instead of the position to specify the arguments to the function.

There are two advantages

— one, using the function is easier since we do not need to worry about the order of the arguments.

— Two, we can give values to only those parameters to which we want to, provided that the other parameters have default argument values.

```
def func(a, b=5, c=10):
    print('a is', a, 'and b is', b, 'and c is', c)

func(3, 7)
func(25, c=24)
func(c=50,
a=100)
```

Output:

a is 3 and b is 7 and c is 10 a is 25

and b is 5 and c is 24

a is 100 and b is 5 and c is 50

How It Works

The function named func has one parameter without a default argument value, followed by two parameters with default argument values. In the first usage, func(3, 7) , the parameter a gets the value 3 , the parameter b gets the value 7 and c gets the default value of 10 . In the second usage func(25, c=24) , the variable a gets the value of 25 due to the position of the argument. Then, the parameter c gets the value of 24 due to naming i.e. keyword arguments. The variable b gets the default value of 5 . In the third usage func(c=50, a=100) , we use keyword arguments for all specified values. Notice that we are specifying the value for parameter c before that for a even though a is defined before c in the function definition.

*args and **kwargs, Command Line Arguments

Sometimes you might want to define a function that can take any number of parameters, i.e. variable number of arguments, this can be achieved by using the stars.

*args and **kwargs are mostly used as parameters in function definitions. *args and **kwargs allows you to pass a variable number of arguments to the calling function. Here variable number of arguments means that the user does not know in advance about how many arguments will be passed to the called function.

***args** as parameter in function definition allows you to pass a non-keyworded, variable length tuple argument list to the called function.

****kwargs** as parameter in function definition allows you to pass keyworded, variable length dictionary argument list to the called function. *args must come after all the positional parameters and **kwargs must come right at the end

•

```
def total(a=5, *numbers, **phonebook):
   print('a', a)
#iterate through all the items in tuple
for single_item in numbers:
print('single_item', single_item)
#iterate through all the items in dictionary
for first_part, second_part in phonebook.items():
  print(first_part,second_part)
  total(10,1,2,3,Jack=1123,John=2231,Inge=1560)
```

Output:

```
a 10
single_item 1
single_item 2
single_item 3
Inge 1560
John 2231
Jack 1123
```

Note: statement blocks of the function definition * and ** are not used with args and kwargs.

Command Line Arguments

- The **sys** module provides a global variable named *argv* that is a list of extra text that the user can supply when launching an application from command prompt in Windows and terminal in OS X and Linux.
- Some programs expect or allow the user to provide extra information.
- Example : let us consider the user supplies a range of integers on the command line. The program then prints all the integers in that range along with their square roots.

sqrtcmdline.py

```
from sys import argv
from math import sqrt

if len(argv) < 3:
  print('Supply range of values')
else:
  for n in range(int(argv[1]), int(argv[2]) + 1):
    print(n, sqrt(n))
```

Output:

C:\Code>python **sqrtcmdline.py 2 5**

2 1.4142135623730951

3 1.7320508075688772

4 2.0

5 2.23606797749979

Question Bank

Q. No.	*Questions*
1	Explain the features of python.
2	Give the comparison between Interpreter and compiler
3	Define python? List the standard data types of python?
4	Explain Type conversion in Python with examples.
5	Write a short note on data types in Python.
6	Differentiate between local and global variables with suitable examples.
7	Write short notes on : i)Variables and statements ii)Expressions iii) String and Modules operator
8	Explain with example how to read input from user in python
9	Explain the different types of arithmetic operators with example.
10	Discuss operator precedence used in evaluating the expression
11	Briefly explain the conditional statements available in Python.
12	Explain the syntax of for loop with an example
13	When to use nested condition in programming? Discuss with an example.
14	Explain the fruitful and void functions? Give examples.
15	With an example, Demonstrate effective coding with functions in Python.
16	Explain the working of python user defined functions along with its syntax.
17	Explain the following terms i)Boolean and logical expression ii) Condition execution and alternative execution with syntax iii)Chained conditions iv)Short circuit evaluation of logical expression
18	Illustrate the three functions of random numbers with example
19	List and explain all built in math functions with example
20	Write a note on short circuit evaluation for logical expression
21	Predict the output for following expression: i)-11%9 ii) 7.7//7 iii)(200-7)*10/ 5 iv) 5*2**1
22	What is the purpose of using break and continue?
23	Differentiate the syntax of if...else and if...elif...else with an example.

22	Create a python program for calculator application using functions
23	Define function. What are the advantages of using a function?
24	Differentiate between user-defined function and built-in functions.
25	Explain the built-in functions with examples in Python.
26	Explain the advantages of *args and **kwargs with examples.