Sree Siddaganga Education Society ®Estd: 1966-67

Sree Siddaganga College of Arts Science & Commerce

Affiliated to Tumkur University

Admission for Both Boys & Girls

A College with a difference-NAAC B ++

B.H.Road, Tumkur-572102.Ph:0816-2278569, 8277338148

Website: - www.sscasc.in

e-mail:-principal.sscasc@gmail.com

STUDY MATERIAL

Subject:-Python Programming

DEPARTMENT OF COMPUTER SCIENCE

6th Semester BSc

Chapter-1
Python Programming Basics

Introduction to Python: Python is an easy to learn, powerful programming language. It combines the features of C and JAVA. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. Python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms.

Python is open source software, which means anybody can freely download it from www.python.org and use it to develop programs. Its source code can be accessed and modified as required in the projects.

Features of python:

1. Simple: Python is a simple and minimalistic language. Reading a good Python program feels almost like reading English language. It means more clarity and less stress on understanding the syntax of the language. It allows you to concentrate on the solution to the problem rather than the language itself.

2. Easy to learn: Python uses very few keywords. Python has an extraordinarily simple syntax and simple program structure.

3. Open Source: There is no need to pay for Python software. Python is *FLOSS* (Free/Library and Open Source Software). It can be free downloaded from www.python.org website. Its source can be read, modified and used in programs as desired by the programmers.

4. High level language: When you write programs in Python, you never need to bother about the low-level details such as managing the memory used by your program, etc.

5. Dynamically typed: Python provides *IntelliSense. IntelliSense* to make writing your code easier and more error-free. IntelliSense option includes statement completion, which provides quick access to valid member function or variables, including global, via the member list. Selecting from the list inserts the member into your code.

6. Portable: Due to its open-source nature, Python has been ported to (i.e. changed to make it work on) many platforms. All your Python programs can work on any of these platforms without requiring any changes at all if you are careful enough to avoid any system-dependent features.

7. Platform independent: When a Python program is compiled using a Python compiler, it generates byte code. Python's byte code represents a fixed set of instructions that run on all operating systems and hardware. Using a Python Virtual Machine (PVM), anybody can run these byte code instructions on any computer system. Hence, Python programs are not dependent on any specific operating system.

8. Procedure and Object Oriented: Python supports procedure-oriented programming as well as object-oriented programming. In *procedure-oriented* languages, the program is built around procedures or functions which are nothing but reusable pieces of programs. In *object-oriented* languages, the program is built around objects which combine data and functionality.

Python has a very powerful but simplistic way of doing OOP, especially when compared to big languages like C++ or Java.

9. Interpreted: Python converts the source code into an intermediate form called byte codes and then translates this into the native language of your computer using PVM(Is s interpreter) and then runs it.

10. Extensible: The programs or function written in C / C++ can be integrated into Python an executed using PVM. There is another flavor of Python where programs from other languages can be integrated into Python.

11. Embeddable: You can embed Python within your C/C++ programs to give *scripting* capabilities for your program's users.

12. Extensive Libraries: The Python Standard Library is huge indeed. It can help you do various things involving regular expressions, documentation generation, unit testing, threading, databases, web browsers, CGI, FTP, email, XML, XML-RPC, HTML, WAV files, cryptography, GUI (graphical user interfaces), and other system-dependent stuff. Remember, all this is always available wherever Python is installed. This is called the ***Batteries*** *Included* philosophy of Python.

Some interesting batteries or packages are:

- **orgparse** is a package that represents command-line parsing library
- **botois** Amazon web services library
- **cherryPhy** is an object-oriented HTTP frame work
- **cryptography** offers cryptographic techniques for the programmers.
- **fiona** reads and writes big data files.
- **numpy** is package for processing arrays of single or multidimensional type.
- **w3lib** is a library of web related functions.
- **mysql-connector-pythonis** is a driver written in Python to connect to MySQL data base.

13. Scripting language: Python is considered as scripting language as it is interpreted and it is used on the Internet to support other software.

14. Database connectivity: Python provides interface to connect its programs to all major databases like Oracle, Sybase or MySQL.

15. Scalable: Python programs are scalable since they can run on any platform and use the features of the new platform effectively.

Comparison between C and Python

C-Language	**Python**
Procedure Oriented Programming Language	Object Oriented Programming Language
Program execute faster	Program execute slower compare to C
Declaration of variable is compulsory	Type declaration is NOT required.
Type discipline is static and weak	Type discipline is dynamic and string
Pointer is available	No pointer
Does not have exception handling	Handles exceptions
It has while, for and do-while loops	It has while and for loops

It has switch-case statement	It does not have switch-case statement
The variable in for loop does not incremented automatically.	The variable in the for loop incremented automatically.
Memory allocation and de-allocation is not automatic	Memory allocation and de-allocation is done automatically by PVM.
It does not contain a garbage collection	Automatic garbage collection
It supports single and multi dimensional arrays	It supports only single dimensional array. Implement multi dimensional array we should use third party application like numpy.
The array index should be positive integer.	Array index can be positive and negative integer. Negative index represents location from the end of the array.
Indentation of statements in not necessary	Indentation is required to represents a block of statements.
A semicolon is used to terminate the statements and comma is used to separate expressions / variables.	New line indicates end of the statements and semicolon is used as an expression separator.
It supports in-line assignment	It does not supports in-line assignment.

Comparison between Java and Python

Java	**Python**
Pure Object-Oriented Programming Language	Both Object-Oriented and Procedure-Oriented programming language
Java programs are verbose.	Python programs are concise and compact.
Declaration of variable is compulsory	Type declaration is NOT required.
Type discipline is static and weak	Type discipline is dynamic and string
It has while, for and do-while loops	It has while and for loops
It has switch-case statement	It does not have switch-case statement
The variable in for loop does not incremented automatically.	The variable in the for loop incremented automatically.
Memory allocation and de-allocation is automatically by JVM	Memory allocation and de-allocation is done automatically by PVM.
It supports single and multi dimensional arrays	It supports only single dimensional array. Implement multi dimensional array we should use third party application like numpy.
The array index should be positive integer.	Array index can be positive and negative integer. Negative index represents location from the end of the array.
Indentation of statements in not necessary	Indentation is required to represents a block of statements.
A semicolon is used to terminate the statements and comma is used to separate expressions / variables.	New line indicates end of the statements and semicolon is used as an expression separator.
The collection objects like stack, linked list or vector but not primitive data types like int, float, char etc.,	The collection objects like lists and dictionaries can store objects of any type including numbers and lists.

Python Virtual Machine (PVM) or Interpreter

Python converts the source code into byte code. Byte code represents the fixed set of instructions created by Python developers representing all types of operations. The size of each byte code instruction is 1 byte.

The role of PVM is to convert the byte code instructions into machine code. So that the computer can execute those machine code instruction and display the final output. The PVM is also called as interpreter.

Python Shell

Python Interpreter is a program which translates your code into machine language and then executes it line by line.
We can use Python Interpreter in two modes:

1. Interactive Mode.
2. Script Mode.

In Interactive Mode, Python interpreter waits for you to enter command. When you type the command, Python interpreter goes ahead and executes the command, then it waits again for your next command.
In Script mode, Python Interpreter runs a program from the source file.

Interactive Mode

Python interpreter in interactive mode is commonly known as Python Shell. To start the Python Shell enter the following command in terminal or command prompt:

To start the Python 3 Shell enter `python3` instead of just `python`.

```
q@vm:~$ python3
Python 3.5.2 (default, Nov 17 2016, 17:05:23)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

What you are seeing is called Python Shell. `>>>` is known as prompt string, it simply means that Python shell is ready to accept you commands. Python shell allows you type Python code and see the result immediately.
In Python shell, enter the following calculations one by one and hit enter to get the result.

```
>>>
>>> 88 + 4
92
>>> 45 * 4
180
```

Script Mode

Python Shell is great for testing small chunks of code but there is one problem – the statements you enter in the Python shell are not saved anywhere.

In case, you want to execute same set of statements multiple times you would be better off to save the entire code in a file. Then, use the Python interpreter in script mode to execute the code from a file.

Create a new file named `example.py` and following code to it:

```
print("Welcome to Python Program")
print("BCA 6th Sem")
print("SSCASC Tumkur")
```

By convention, all Python programs have `.py` extension. The file `example.py` is called source code or source file or script file or module. Execute by typing the following command an obtained out as follows,

```
$ python example.py
Welcome to Python Program
BCA 6th Sem
SSCASC Tumkur
```

Indentation: Whitespace is important in Python. Actually, **whitespace at the beginning of the line is important**. This is called *indentation*. Leading whitespace (spaces and tabs) at the beginning of the logical line is used to determine the indentation level of the logical line, which in turn is used to determine the grouping of statements.

This means that statements which go together *must* have the same indentation. Each such set of statements is called a **block**. One thing you should remember is that wrong indentation can give rise to errors

Basic elements of Python

1. **Comments :** Comments are non-executable statements. It means neither compiler nor PVM will not execute them. *Comments* are any text to the right of the # symbol and is mainly useful as notes for the reader of the program. There are two types of comments in Python, Single line comments and Multiline comments
   a. **Single line Comments:** this comment starts with a hash symbol (#) and are useful to mention that the entire line till the end should be treated as comments.

      Eg.
      ```
      # To find the sum of two number
      k=5     # assign 5 to variable k.
      ```
      In the above example, first line starts with # and hence the total line treated as comments. In second line part of this line starting from # to the end of the line treated as comments.
   b. **Multi line Comments:** The triple double quotes (""") or triple single quotes (''') are called multi line comments or block comments. They are used to enclose a block of lines as comments.

      Eg-1.
      ```
      """ This is illustrated as multi line comments
      To find the sum of two number
      Using Triple double quotes
      """
      ```
2. **Identifiers:** Identifier is the name given to various program elements like variables, function, arrays, classes, strings etc.,

The Python identifiers follow the following rules:

i. The Name should begin with an alphabet.

ii. Only alphabets, digits and underscores are permitted.
iii. Distinguish between uppercase and lowercase alphabets.
iv. Keywords should not be used as identifiers.
v. No blank space between the identifiers.

e.g. **Valid Identifiers :**

Area
area_tri
num1

3. **Keywords:** The keywords have predefined meaning assigned by the Python Complier. The keywords are also called as **reserved word.** All keywords are written in lower case alphabets. The Python keywords are:

and	del	for	lambda	true
as	elif	from	not	try
assert	else	global	or	while
break	except	if	pass	with
class	exec	import	print	yield
continue	false	in	raise	
def	finally	is	return	

4. **Variable:** Is a program element, whose value changes during the execution of the program. Unlike other programming languages, Python has no command for declaring a variable.
A variable is created the moment you first assign a value to it.
Eg. x=5;
y="kvn"
Variables do not need to be declared with any particular type and can even change type after they have been set.

5. **Constants:** Constant is a program element, while execution of a program the value does not change. A constant gets stored in a memory location but the address of the location is not accessible to the programmer

Assigning value to a constant in Python

In Python, constants are usually declared and assigned on a module. Here, the module means a new file containing variables, functions etc which is imported to main file.

Inside the module, constants are written in all capital letters and underscores separating the words.

Eg.:

Create a constant.py
PI = 3.14

6. **Literals :** Literal is a raw data given in a variable or constant. In Python, there are various types of literals they are as follows:

a. **Numeric Literals :** Numeric Literals are immutable (unchangeable). Numeric literals can belong to 3 different numerical types Integer, Float and Complex.

```
E.g.  a=5       # integer literal
      b=2.5     # float literal
      c=3.5j    # complex literal
```

b. **String literals :** A string literal is a sequence of characters surrounded by quotes. We can use both - single, double or triple quotes for a string. Character literal is a single character surrounded by single or double quotes.

```
E.g.  str="SSCASCT"
```

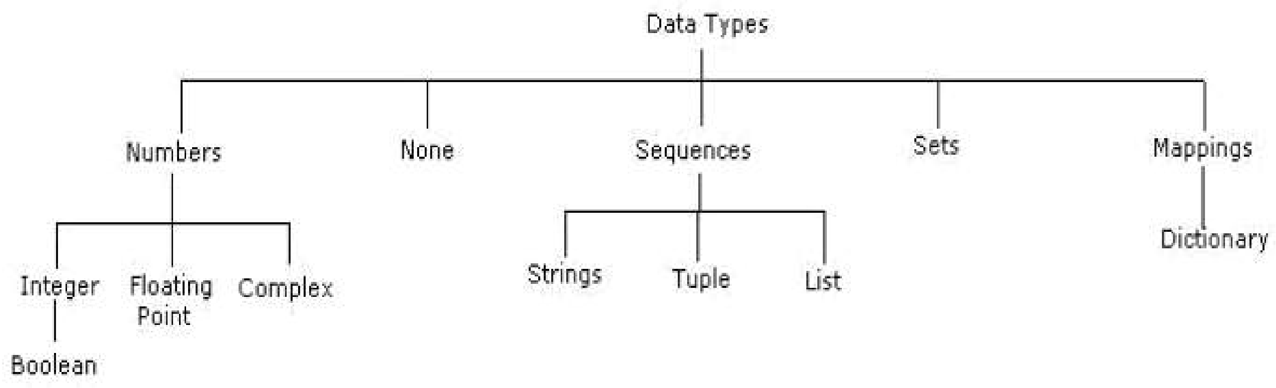
c. **Boolean literals :** A Boolean literal can have any of the two values: **True** or **False**.

```
E.g.  x=true
Y=false
```

d. **Special literals:** Python contains one special literal i.e. **None**. We use it to specify to that field that is not created.

```
E.g.  k=none
```

7. **Data Types:** A data type represents the type of data stored into a variable or memory. There are 5 different data types are:
   - None type
   - Numeric type
   - Sequences
   - Sets
   - Dictionary



i. **None data type :** The none data type represents an object that does not contain any value. In java language it is called "NULL" object. But in Python it is called as "none". In Python maximum of only one 'none' object is provided. If no value is passed to the function, then the default value will be taken as 'none'.

ii. **Numeric data type:** The numeric type represents numbers. There are 3 sub types:
- int
- float
- complex

int data type: The int data type represents integer number (Whole number). An integer number is number without fraction. Integers can be of any length, it is only limited by the memory available.

E.g. a=10
b=-29

float data type: The float data type represents floating point number. A floating point number is a number with fraction. Floating point numbers can also be written in scientific notation using exponentiation format.

A floating point number is accurate up to 15 decimal places. Integer and floating points are separated by decimal points.

complex data type: A complex number is number is written in the form of x +yj or x+yJ. Here x is the real part and y is the imaginary part.

We can use the **type()** function to know which class a variable or a value belongs to and the **isinstance()** function to check if an object belongs to a particular class.

E.g.
```
a = 5
print(a, "is of type", type(a))
b = 2.0
print(a, "is of type", type(b))
```

iii. **Sequences:** A sequence represents a group of items or elements. There are six types of sequences in Python. Important sequences as follows,
- str
- list
- tuple

str data type : The str represents string data type. A string is a collection of character enclosed in single or double quotes. Both are valid.

```
E.g.        str="kvn"           # str is name of string variable
            str='vedish'        # str is name of string variable
```

Triple double quote or triple single quotes are used to embed a string in a another string (Nested string).

```
str="""This is 'str data type' example"""
        print(str)          # output is :       This is 'str data type' example
```

The [] operator used to retrieve specified character from the string. The string index starts from 0. Hence, str[0] indicates the 0th character in the string.

```
e.g     str=" SSCASC Tumkur"
        print(str)          # it display -  SSCASC Tumkur
        print(str[0])       # it display -    G
```

list data type: A List is a collection which is ordered and changeable. It allows duplicate members. A list is similar to array. Lists are represented by square brackets [] and the elements are separated by comma.

The main difference between a list and an array is that a list can store different data type elements, but an array can store only one type of elements. List can grow dynamically in memory but the size of array is fixed and they cannot grow dynamically.

```
e.g.    list=[10,3.5,-20, "SSCASCT",'TUMKUR']          # create a list
        print(list)   # it display all elements in the list :   10,3.5,-20,
"SSCASCT",'TUMKUR'
```

tuple data type: A tuple is similar to list. A tuple contains group of elements which can be different types. The elements in the tuple are separated by commas and enclosed in parentheses (). The only difference is that tuples are immutable. Tuples once created cannot be modified. The tuple cannot change dynamically. That means a tuple can be treated as read-only list.

```
e.g.    tpl=(10,3.5,-20, "SSCASCT",'TUMKUR')          # create a tuple
        print(tpl)   # it display all elements in the tuple :   10,3.5,-20,
"SSCASCT",'TUMKUR'
```

iv. **Sets:** Set is an unordered collection of unique items and un-indexed. The order of elements is not maintained in the sets. A set does not accept duplicate elements. Set is defined by values separated by comma inside braces { }.
There are two sub types in sets:

- Set data type

- Frozen set data type

Set data type: To create a set, we should enter the elements separated by comma inside a curly brace.

e.g. s = {10,30, 5, 30,50}
print(s) # it display : {10,5,30,50}

In the above example, it displays un-orderly and repeated elements only once, because set is unordered collection and unique items.

We can use set() to create a set as
K=set("kvn")
Print(K) # it display : "kvn"

Frozen set data type: Frozen set is just an immutable version of a Python set object. While elements of a set can be modified at any time, an element of frozen set remains the same after creation. Due to this, frozen sets can be used as key in Dictionary or as element of another set.

v. **Dictionary:** A dictionary is an unordered collection, changeable and indexed. In Python dictionaries are written with curly brackets, and they have keys and values. That means dictionary contains pair of elements such that first element represents the key and the next one becomes its value. The key and value should be separated by a colon(:) and every pair should be separated by comma. All the elements should be enclosed inside curly brackets.

e.g.
d={3: 'sscasc', 4:'tumkur', 5:'kvn', 6: 'vedish'}

Here, d is the name of dictionary. 3 is the key and its associated value is 'sscasc'. The next is 4 and its value is 'tumkur' and so on.

Print(d) # it display : ={3: 'sscasc', 4:'tumkur', 5:'kvn', 6: 'vedish'}
Print(d[5]) # it display : kvn

String operations:

1. Extract specified character from the string. Get the character at position 1 (remember that the first character has the position 0):

   a= "Hello,Python!"
   print(a[6]) # it display : P
2. Substring: Extract number of character from the specified position. Get the characters from position 2 to position 5 (not included):
   ```
   b= "Hello,Python!"
   print(b[6:8])     # it display :    Ph
   ```
3. The strip() method removes any whitespace from the beginning or the end of the given string.
   ```
   a= "                    Hello,Python!                    "
   print(a.strip())        # it display :  "Hello,Python!"
   ```

4. The len() method returns the length of a given string

```
a= "                                    Python"
print(a.len())          # it display : 6
```

5. The lower() method returns the given string in lower case.

```
a= "                                    PYTHON"
print(a.lower())              # it display : python
```

6. The upper() method returns the given string in upper case.

```
a= "                                    python"
print(a.upper())              # it display : PYTHON
```

7. The replace() method replaces a given string with another string

```
a= "FOR"
print(a.replace('O' , 'A'))            # it display : FAR
```

8. The split() method splits the string into substrings if it finds instances of the separator

```
a= "Hello,Python!"
print(a.split(','))      # it display   :      [        'Hello'         ,          'Python'}
```

Operator and Operand:
Operators are special symbols which represents computation. They are applied on operand(s), which can be values or variables. Same operator can behave differently on different data types. Operators when applied on operands form an expression. Operators are categorized as Arithmetic, Relational, Logical and Assignment. Value and variables when used with operator are known as operands.

1. Arithmetic Operators :

Symbol	Description	Example-1	Example-2
+	Addition	>>> 5 + 6 11	>>>'SSCASCT'+'BCA' SSCASCTBCA
-	Subtraction	>>>10-5 5	>>>5 – 6 -1
*	Multiplication	>>> 5*6 30	>>>'SSCASCT' * 2 SSCASCTSSCASCT
/	Division	>>> 10 / 5 2	>>>5 /2.0 2.5
%	Remainder / Modulo	>>> 5 % 2 1	>>>15%5 0
**	Exponentiation	>>> 2**3 8	>>>2**8 256
//	Integer Division	>>> 7.0 // 2 3.0	>>>3//2 1

2. Relational Operators :

Symbol	Description	Example-1	Example-2
<	Less than	>>> 7<10 True	>>> 'SSCASCT' <'BCA' False

>	Greater Than	>>> 7 >10 False	>>>'SSCASCT' > 'BCA' True
<=	Less than or equal to	>>> 7<=10 True	>>>'SSCASCT' <='BCA' False
>=	Greater than or equal to	>>> 7>=10 False	>>>'SSCASCT'>='BCA' True
!= , <>	Not equal to	>>> 7!=10 True	>>>'SSCASCT'!= 'sscasct' True
==	Equal to	>>> 7==10 False	>>>'SSCASC' =='SSCASC' True

3. Logical Operators :

Symbol	**Description**	**Example-2**
or	If any one of the operand is true, then condition becomes TRUE	>>> 7<=10 or 7 ==10 True
and	If both the operands are true, then the condition becomes TRUE	>>>7<10 and 7 >20 False
not	Reverse the state of operand / condition	>>> not 7<10 False

4. Assignment Operator:

Symbol	**Description**	**Example-1**
=	Assigned values from right side operands to left variable.	>>> x=10 10

Variations of Assignment Operators:

Compound Assignment Operator combines the effect of arithmetic and assignment operator, the original value of x =5

Symbol	**Description**	**Example-1**
+=	added and assign back the result to left operand	>>> x+=2 7
-=	subtracted and assign back the result to left operand	>>> x-=2 3
*=	multiplied and assign back the result to left operand	>>> x*=2 10
/=	divided and assign back the result to left operand	>>> x/=2 2
%=	taken modulus using two operands and assign the result to left operand	>>> x%=2 1
**=	performed exponential (power) calculation on operators and assign value to the left operand	>>> x**=2 25
//=	performed floor division on operators and assign value to	>>> x//=2 2.5

	the left operand	

5. **Bitwise Operator:** a bit is the smallest unit of data storage and it can have only one of the two values, 0 and 1. Bitwise operators works on bits and perform bit-by-bit operation.

Symbol	**Description**	**Example**
|	Performs binary *OR* operation	5 **|** 3 gives 7
&	Performs binary *AND* operation	5 & 3 gives 1
~	Performs binary *XOR* operation	5 ^ 3 gives 6
^	Performs binary one's complement operation	~5 gives -6
<<	**Left shift operator**: The left-hand side operand bit is moved left by the number specified on the right-hand side (Multiply by 2)	0010 << 2 gives 8
>>	**Left shift operator**: The left-hand side operand bit is moved left by the number specified on the right-hand side (Divided by 2)	0100 << 2 gives 1

6. **Membership operators:** Python has membership operators, which test for membership in a sequence, such as strings, lists or tuples. There are two membership operators are:

Symbol	**Description**	**Example**

in	Returns True if the specified operand is found in the sequence	>>> x = [1,2,4,6,8]
		>>> 3 in x
		false

Not in	Returns True if the specified operand is found in the sequence	>>> x = [1,2,4,6,8]
		>>> 3 not in x
		true

7. **Identity operator:** Identity operators compare the memory locations of two objects. There are two Identity operators are:

Symbol	**Description**	**Example**	**Example**

is	Returns True if two variables point to the same object and False, otherwise	>>>X=10
		>>>Y=10
		>>> X is Y
		true
			>>> x=[1,2,3]
			>>> y=[1,2,3]
			>>> x is y
			false

is not	Returns False if two variables point to the same object and True, otherwise	>>>X=10
		>>>Y=10
		>>> X is not Y
		false
			>>> x=[1,2,3]
			>>> y=[1,2,3]
			>>> x is not y
			true

CHAPTER 2
CREATING PYTHON PROGRAM

Input function

The print function enables a Python program to display textual information to the user. Programs may use the input function to obtain information from the user. The simplest use of the input function assigns a string to a variable:

x = input()

The parentheses are empty because the input function does not require any information to do its job. usinginput.py demonstrates that the input function produces a string value.

Exaple: Demonstrates that the input function

```
print('Please enter some text:')
x = input()
print('Text entered:', x)
print('Type:', type(x))
```

Since user input always requires a message to the user about the expected input, the input function optionally accepts a string and prints just before the program stops to wait for the user to respond.

The statement **variable = input("Enter a value: ")**

The value entered is a string. You can use the function **eval** to evaluate and convert it to a numeric value.

Example: `eval("34.5")` returns 34.5, `eval("345")` returns 345.

Example: Compute area with console input

```
# Prompt the user to enter a radius
  radius = eval(input("Enter a value for radius: "))
# Compute area
  area = radius * radius * 3.14159
#Display results
  print("The area for the circle of radius",radius,"is",area)
```

Compute Average

```
# Prompt the user to enter three numbers
number1 = eval(input("Enter the first number: "))
number2 = eval(input("Enter the second number: "))
number3 = eval(input("Enter the third number: "))
# Compute average
average = (number1 + number2 + number3) / 3
# Display result
print("The average of", number1, number2, number3,"is",average)
```

Print Function

The print a line of text, and then the cursor moves down to the next line so any future printing appears on the next line.

```
    print('Please enter an integer value:')
```

The print statement accepts an additional argument that allows the cursor to remain on the same line as the printed text:

```
    print('Please enter an integer value:', end=' ')
```

The expression end=' ' is known as a keyword will cause the cursor to remain on the same line as the printed text. Without this keyword argument, the cursor moves down to the next line after printing the text.
Another way to achieve the same result is

```
print(end='Please enter an integer value: ')
```

This statement means "Print nothing, and then terminate the line with the string 'Please enter an integer value:' rather than the normal \n newline code.
The statement

```
print('Please enter an integer value:', end='\n')
```

that is, the default ending for a line of printed text is the string '\n', the newline control code. Similarly, the statement

```
print( )
```

is a shorter way to express

```
print(end='\n')
```

By default, the print function places a single space in between the items it prints. Print uses a keyword argument named **sep** to specify the string to use insert between items. The name **sep** stands for separator. The default value of sep is the string ' ', a string containing a single space. The program printsep.py shows the **sep** keyword customizes print's behaviour.

Program to illustrate sep:

```
w, x, y, z = 10, 15, 20, 25
print(w, x, y, z)
print(w, x, y, z, sep=' , ')
print(w, x, y, z, sep=' ')
print(w, x, y, z, sep=' : ')
print(w, x, y, z, sep='-----')
```

The output

```
10 15 20 25
10,15,20,25
10 15 20 25
10:15:20:25
10-----15-----20-----25
```

Formatting Numbers and Strings

Format function is used to return a formatted string for displaying numbers in a certain desirable format. For example, the following code computes interest, given the amount and the annual interest rate.

```
>>> amount = 12618.98
>>> interestRate = 0.0013
>>> interest = amount * interestRate
>>> print("Interest is", format(interest, ".2f "))
Interest is 16.40
>>>
```

The syntax to invoke this function is

```
format(item, format-specifier)
```

Where item is a number or a string and format-specifier is a string that specifies how the
item is formatted. The function returns a string.

Frequently Used Specifiers

Specifier	***Format***
"10.2f"	Format the float item with width 10 and precision 2.
"10.2e"	Format the float item in scientific notation with width 10 and precision 2.
"5d"	Format the integer item in decimal with width 5.
"5x"	Format the integer item in hexadecimal with width 5.
"5o"	Format the integer item in octal with width 5.
"5b"	Format the integer item in binary with width 5.
"10.2%"	Format the number in decimal.
"50s"	Format the string item with width 50.
"<10.2f"	Left-justify the formatted item.
">10.2f"	Right-justify the formatted item.

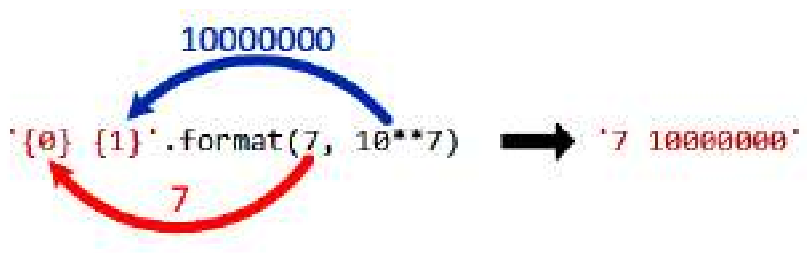
Placeholder substitution within a formatting string

```
print('{0} {1}'.format(2, 10**2))
```

This expression has two main parts:

• '{0} {1}': This is known as the formatting string. It is a Python string because it is a sequence of characters enclosed with quotes. Notice that the program at no time prints the literal string {0} {1}. This formatting string serves as a pattern that the second part of the expression will use. {0} and {1} are placeholders, known as positional parameters, to be replaced by other objects. This formatting string, therefore, represents two objects separated by a single space.

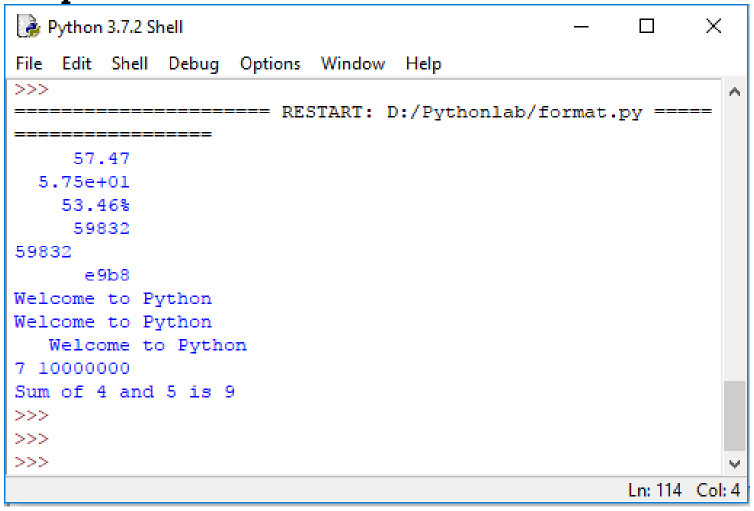
• format(2, 10**2): This part provides arguments to be substituted into the formatting string. The first argument, 2, will take the position of the {0} positional parameter in the formatting string. The value of the second argument, 10**2, which is 100, will replace the {1} positional parameter.



Program to Illustrate Format Specifier

```
    print(format(57.467657, "10.2f"))
    print(format(57.467657, "10.2e"))
    print(format(0.53457, "10.2%"))
    print(format(59832, "10d"))
    print(format(59832, "<10d"))#Left Justfy
    print(format(59832, "10x"))#Converts to HexaDecimal
    print(format("Welcome to Python", "20s"))
    print(format("Welcome to Python", "<20s"))
    print(format("Welcome to Python", ">20s"))#Right Justify
    print('{0} {1}'.format(7,10**7))
    print('Sum of {0} and {1} is {2}'.format(4,5,4+5))
```

Output



```
>>>
====================== RESTART: D:/Pythonlab/format.py =====
=================
     57.47
  5.75e+01
    53.46%
     59832
59832
      e9b8
Welcome to Python
Welcome to Python
   Welcome to Python
7 10000000
Sum of 4 and 5 is 9
>>>
>>>
>>>
```

CONTROL STATEMENTS

If Statements

A one-way if statement executes the statements if the condition is true. The syntax for a one-way if statement is:

```
    if boolean-expression:
        statement(s) # Note that the statement(s) must be indented
```

• The reserved word if begins a if statement.
• The condition is a Boolean expression that determines whether or not the body will be
  executed. A colon (:) must follow the condition.
• The block is a block of one or more statements to be executed if the condition is true.
  The statements within the block must all be indented the same number of spaces from
  the left. The block within an

Example:To demonstrate simple if

```
#Get two integers from the user
dividend = int(input('Please enter the number to divide: '))
```

```
divisor = int(input('Please enter dividend: '))
# If possible, divide them and report the result
if divisor != 0:
    quotient = dividend/divisor
    print(dividend, '/', divisor, "=", quotient)
print('Program finished')
```

Output

```
Please enter the number to divide: 4
Please enter dividend: 5
4 / 5 = 0.8
Program finished
>>>
```

If-else statements

A two-way **if-else** statement decides which statements to execute based on whether the condition is true or false.

The syntax for a two-way **if-else** statement:

```
if boolean-expression:
   statement(s)     #for-the-true-case , the statement(s) must be indented
else:
   statement(s)     #for-the-false-case
```

Example: To demonstrate if else

```
percent=float(input("Enter Percentage"))
if percent >= 90.0:
    print ("congratulations, you got an A")
    print ("you are doing well in this class")
else:
    print ("you did not get an A")
    print ("see you in class next week")
```

Nested if statements.

A series of tests can written using nested if statements.

Example: Nestedif

```
percent=float(input("Enter Percentage"))
if (percent >= 90.00):
    print ('congratuations, you got an A')
else:
    if (percent >= 80.0):
        print ('you got a B')
    else:
        if (percent >= 70.0):
            print ('you got a C')
        else:
            print ('your grade is less than a C')
```

If_elif_else Statement

In Python we can define a series of conditionals (multiple alternatives) using if for the first one, elif for the rest, up until the final (optional) else for anything not caught by the other conditionals.

Example:If_elif_else

```
score=int(input("Enter Score"))
if score >= 90.0:
    grade = 'A'
elif score >= 80.0:
    grade = 'B'
elif score >= 70.0:
    grade = 'C'
elif score >= 60.0:
    grade = 'D'
else:
    grade = 'F'
print("Grade=",grade)
```

Using **else if** instead of **elif** will trigger a syntax error and is not allowed.

Loops

It is one of the most basic functions in programming; loops are an important in every programming language. Loops enable is to execute a statement repeatedly which are referred to as iterations. (*A loop is used to tell a program to execute statements repeatedly*).
The simplest type of loop is a **while** loop.

The syntax for the while loop is:

```
while loop-continuation-condition:
     # Loop body
     Statement(s) # Note that the statement(s) must be indented
```

```
i = initialValue # Initialize loop-control variable
while i < endValue:
     # Loop body
     ...
     i += 1 # Adjust loop-control variable
```

Example1: To demonstrate while

```
count = 0#Program to print "Programming is fun!" for 10 times
while count < 10:
    print("Programming is fun!")
    count = count + 1
```

Example2:

```
#program to read a series of values from the user, count the
#number of items, and print their count, sum and average.
#User indicates the end of the input by typing the special value -
1.
```

```
sum = 0
count = 0
num=int(input("Enter your number:"))
while num != -1:
    sum = sum + num
    count = count + 1
    num =int(input("enter your number:"))
print ("Count is :", count)
print ("Sum is :", sum)
print ("Average is :", sum / count)
```

The for Loop

A for loop iterates through each statements in a sequence for exactly know many times the loop body needs to be executed, so a control variable can be used to count the executions. A loop of this type is called a counter-controlled loop. In general, the loop can be written as follows:

```
for i in range(initialValue, endValue):
     # Loop body # Note that the statement(s) must be indented
```

In general, the syntax of a for loop is:

```
for var in sequence:
     # Loop body
```

The function **range(a, b)** returns the sequence of integers **a**, **a + 1**, ..., **b-2**, and **b- 1.** The **range** function has two more versions. You can also use **range(a)** or **range(a, b, k)**. **range(a)** is the same as **range(0, a). k** is used as ***step value*** in **range(a, b, k)**. The first number in the sequence is **a**. Each successive number in the sequence will increase by the step value **k**. **b** is the limit. The last number in the sequence must be less than **b**.

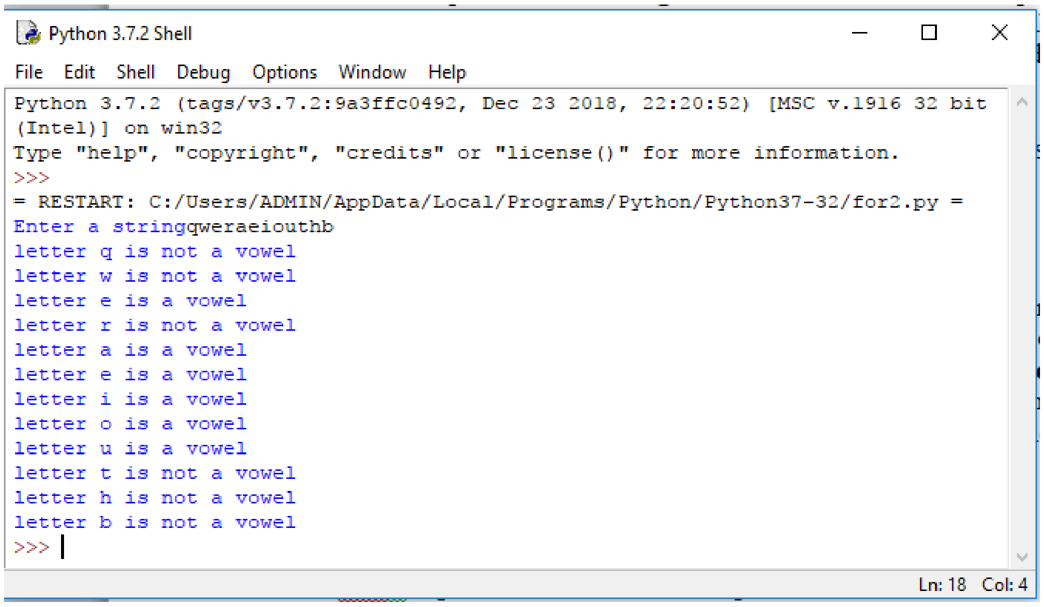
Example1:To demonstrate For

```
for i in range(5):
    print (i)
```

Example2:

```
str=input("Enter a string")
for ch in str:
    if ch in 'aeiou':
        print ('letter', ch, 'is a vowel')
    else:
        print ('letter ', ch, 'is not a vowel')
```

Output of example2



```
Python 3.7.2 (tags/v3.7.2:9a3ffc0492, Dec 23 2018, 22:20:52) [MSC v.1916 32 bit
(Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: C:/Users/ADMIN/AppData/Local/Programs/Python/Python37-32/for2.py =
Enter a stringqweraeiouthb
letter q is not a vowel
letter w is not a vowel
letter e is a vowel
letter r is not a vowel
letter a is a vowel
letter e is a vowel
letter i is a vowel
letter o is a vowel
letter u is a vowel
letter t is not a vowel
letter h is not a vowel
letter b is not a vowel
>>> |
```

We can iterate through a list by using for:

Example3

```
for x in ['one', 'two', 'three', 'four']:
      print(x)
This will print out the elements of the list:
one
two
three
four
```

Iterating over dictionaries:

```
Considering the following dictionary:
d = {"a": 1, "b": 2, "c": 3}
#To iterate through its keys, we can use:
for key in d:
    print(key)Output:
Output:
"a"
"b"
"c"
```

Break and Continue in Loops

break statement:

When a break statement executes inside a loop, control flow comes out of the loop immediately:

```
Example:to demonstrate break
i = 0
while i < 7:
    print(i)
    if i == 4:
        print("Breaking from loop")
        break
    i += 1
```

The loop conditional will not be evaluated after the break statement is executed. Note that break statements are only allowed inside loops. A break statement inside a function cannot be used to terminate loops that called that function.

Executing the following prints every digit until number 4 when the break statement is met and the loop stops:

Output

```
01234
Breaking from loop
```

Break statements can also be used inside for loops, the other looping construct provided by Python:

Example:

```
for i in (0, 1, 2, 3, 4):
    print(i)
    if i == 2:
        break
```

Executing this loop now prints:

```
012
```

Note that 3 and 4 are not printed since the loop has ended.

Continue statement

A continue statement will skip to the next iteration of the loop bypassing the rest of the current block but continuing the loop. Continue can only used inside loops:

Example to demonstrate continue

```
for i in (0, 1, 2, 3, 4, 5):
    if i == 2 or i == 4:
        continue
    print(i)
```

Note that 2 and 4 aren't printed, this is because continue goes to the next iteration instead of continuing on to print(i) when i == 2 or i == 4.

Executing this loop now prints:

```
0135
```

The Pass

The pass statement is used in code in places where the language requires a statement to appear but we wish the program to take no action. We can make the code fragment legal by adding a pass statement:

```
if x < 0:
pass # Do nothing
else:
print(x)
```

Pass is a null statement, when a statement is required by Python syntax (such as within the body of a for or while loop), but no action is required or desired by the programmer. This can be useful as a placeholder for code that is yet to be written.

Example: to demonstrate pass

```
for x in range(10):
    pass #we don't want to do anything, so we'll pass
```

In this example, nothing will happen. The for loop will complete without error, but no commands or code will be actioned. Pass allows us to run our code successfully without having all commands and action fully implemented. Similarly, pass can be used in while loops, as well as in selections and function definitions etc.

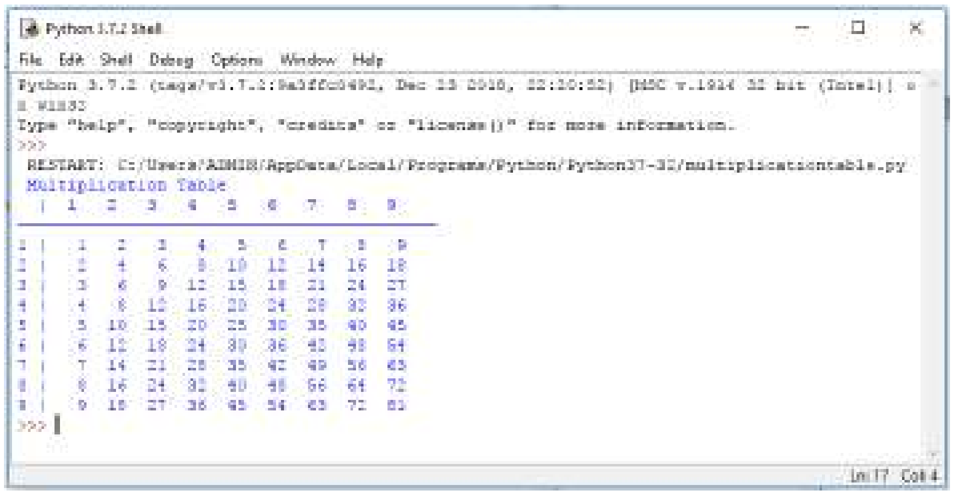
Nested Loops

A loop can be nested inside another loop. Nested loops consist of an outer loop and one or more inner loops. Each time the outer loop is repeated; the inner loops are re-entered and started a new. Below is classical example for nesting of loops.

Example: Multiplcationtable

```
print(" Multiplication Table")
# Display the number title
print("  |", end = '')
for j in range(1, 10):
    print(" ", j, end = ' ')
print() # Jump to the new line
print("——————————————————————————————————————————")
# Display table body
for i in range(1, 10):
    print(i, "|", end = '')
# Display the product and align properly
    for j in range(1, 10):
        print(format(i * j, "4d"), end = '')
    print() # Jump to the new line
```

Output



```
Python 3.7.2 (tags/v3.7.2:9a3ffc0492, Dec 23 2018, 22:20:52) [MSC v.1916 32 bit (Intel)] o
n win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
 RESTART: C:/Users/ADMIN/AppData/Local/Programs/Python/Python37-32/multiplicationtable.py
 Multiplication Table
  |  1   2   3   4   5   6   7   8   9
_____________________________________
1 |  1   2   3   4   5   6   7   8   9
2 |  2   4   6   8  10  12  14  16  18
3 |  3   6   9  12  15  18  21  24  27
4 |  4   8  12  16  20  24  28  32  36
5 |  5  10  15  20  25  30  35  40  45
6 |  6  12  18  24  30  36  42  48  54
7 |  7  14  21  28  35  42  49  56  63
8 |  8  16  24  32  40  48  56  64  72
9 |  9  18  27  36  45  54  63  72  81
>>> 
```

FUNCTIONS

- *A function is a collection of statements grouped together that performs an operation.*
- *A function is a way of packaging a group of statements for later execution.*

The function is given a name. The name then becomes a short-hand to describe the process. Once defined, the user can use it by the name, and not by the steps involved. Once again, we have separated the "what" from the "how", i.e. *abstraction*.

Functions in any programming language can fall into two broad categories:

- **Built-in functions**

They are predefined and customized, by programming languages and each serves a specific purpose.

- **User-defined functions**

They are defined by users as per their programming requirement.

There are two sides to every Python function:

• **Function definition**. The definition of a function contains the code that determines the function's behaviour.

• **Function call**. A function is used within a program via a function invocation.

Defining a Function

A function definition consists of the function's name, parameters, and body. The syntax for defining a function is as follows:

```
def functionName(list of parameters):
    Statements # Note that the statement(s) must be indented
    return
```

- A function contains a header and body. The header begins with the *def* keyword, followed by the function's name known as the identifier of the function and parameters, and ends with a colon.
- The variables in the function header are known as formal parameters or simply parameters. When a function is invoked, you pass a value to the parameter. This

value is referred to as an actual parameter or argument. Parameters are optional; that is, a function may not have any parameters.

- Statement(s) – also known as the function body – are a nonempty sequence of statements executed each time the function is called. This means a function body cannot be empty, just like any indented block.
- Some functions return a value, while other functions perform desired operations without returning a value. If a function returns a value, it is called a value-returning function.

Calling a Function

Calling a function executes the code in the function. In a function's definition, you define what it is to do. To use a function, you have to call or invoke it. The program that calls the function is called a caller. There are two ways to call a function, depending on whether or not it returns a value. If the function returns a value, a call to that function is usually treated as a value.

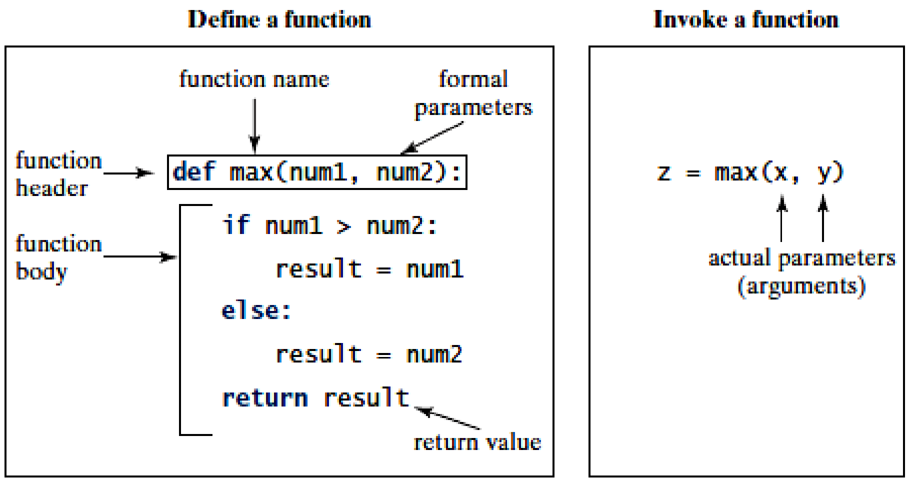
For example,

```
larger = max(3, 4)
```

Calls max(3, 4) and assigns the result of the function to the variable larger.
Another example of a call that is treated as a value is

```
print(max(3, 4))
```

This prints the return value of the function call max (3, 4).



```
def max(num1, num2):
    if num1 > num2:
        result = num1
    else:
        result = num2
    return result
```

```
z = max(x, y)
```

Example:

```
# Return the max of two numbers
# Function with arguments and return
def max(num1, num2):
    if num1 > num2:
       result = num1
    else:
        result = num2
    return result
def main():
    i = 5
    j = 2
    k = max(i,j)# Call the max function
    print("The larger number of", i, "and", j, "is", k)
main() # Call the main function
```

Output

```
The larger number of 5 and 2 is 5
```

Categories of User-defined functions

1. Function with arguments
2. Function with an argument and return type
3. Function with default argument
4. Function with variable length argument
5. Pass by reference.

Function with arguments

A function can contain any number of arguments depending on the requirement.

Example:

```
def func(passArgument):#function definition
     print passArgument

str = "Hello all"
func(str)# function call
```

In this example, the ***func*** function accepts one argument which has a data type string. We create a variable str with a certain string statement assigned and then we call
the ***func*** function and thereby pass the value of str.
Finally, the output will be:

```
Hello all
```

Function with an argument and return type

This type of function takes any arbitrary number of arguments and return specific data type or value from it.

Example: Refer above program Fun_max.py

Function with default argument

In this type of function, the formal parameter assigned with some value, represents a default parameter or default argument. If the caller does not supply an actual parameter, the formal parameter value is assigned.

Example:

```
def info(name, age=50):
     print("Name:", name)
     print("Age:", age)

info("John", age=28)
info("James")
     Output: Name: John
              Age: 28
             Name: James
             Age: 50
```

Function with variable length argument

There might be a scenario where you need to pass more arguments than specified during the function definition. In this case, variable length arguments can be passed:

Syntax

```
     def function_name(arg, *var):
          code block
          return
```

Here, arg means normal argument which is passed to the function. The *var refers to the
variable length argument.

Example:

```
     def variable_argument( arg, *vari):
         print ("Out-put is",arg)
         for var in vari:
             print (var)

     variable_argument(60)
     variable_argument("Hari",100,90,40,50,60)
```

Output:

```
Out-put is 60
Out-put is Hari
100
90
40
50
60
>>>
```

Pass by reference versus pass by value

In pass by reference values to the argument of the function are passed as reference, that is, the address of the variable is passed and then the operation is done on the value stored at these addresses.

Pass by value means that the value is directly passed as the value to the argument of the function. In this case, the operation is done on the value and then the value is stored at the address.

In Python arguments, the values are passed by reference. During the function call, the called function uses the value stored at the address passed to it and any changes to it also affect the source variable:

Example: call by reference

```
     def pass_ref(list1):#Call by ref
         list1.extend([23,89])
         print ("list inside the function: ",list1)

     list1 = [12,67,90]
     print ("list before pass", list1)
     pass_ref(list1)
     print ("list outside the function", list1)
```

Output:

```
     list before pass [12, 67, 90]
     list inside the function: [12, 67, 90, 23, 89]
     list outside the function [12, 67, 90, 23, 89]
```

Here, in the function definition, we pass the list to the pass_ref function and then we extend the list to add two more numbers to the list and then print its value. The list extends inside the function, but the change is also reflected back in the calling function. We finally get the output by printing out different representations of the list:
Let's look at another Example:

```
def func(a):#call by value function
a=a+4
print ("Inside the function", a)

a= 10
func(a)
print ("Outside the function", a)
```

The preceding example call by value, as the change happening inside the Python function does not get reflected back in the calling function. It is still a pass by reference, as, in this situation inside the function, we made new assignment, that is, a= a+4. Although you might think that a = a + 4 is changing the number stored in a, but it is actually reassigning a to point to a new value:

Returning Multiple Values from a Function
In python function, we can return multiple values. We can use return statements as

```
return a,b,c
```

Here a,b,c values returned by function as tuples and we have to use three variables to receive these values in called function as shown below

X,Y,Z=function()

Example:sum_sub_mul

```
def Sum_Sub_Mul(a,b):
      sum=a+b
      sub=a-b
      mul=a*b
      return sum,sub,mul
def main():
    a=10
    b=5
    X,Y,Z=Sum_Sub_Mul(a,b)
    print("Sum of two no=",X)
    print("Sub of two no=",Y)
    print("Mul of two no=",Z)
    print("Sum, Sub and Mul of two no:")
    T=Sum_Sub_Mul(a,b)#Retrieving multiple value using Tuples
    for i in T:
        print(i,end='\n ')
main()
```

Output

```
Sum of two no= 15
Sub of two no= 5
Mul of two no= 50
Sum, Sub and Mul of two no:
15
5
50
>>>
```

Local and Global Variables

When we declare variable inside a function it becomes a local variable and its scope is limited to that function where it is created and it is available in that function and not available outside the function.

When a variable declared outside the function, it is available to all function which are using these variables.

Sometimes local variable and global variable have the same name, in this case the function ignores global variable and uses local variable only.

When the programmer wants to use the global variable inside a function, we can use the keyword ***global*** before the variable in the beginning of the function body.

Example:

```
b=20#global variable
c=30
def myfunction():
    a=10#Local variable
    print("Global B:",b)
    print("Local A:",a)
    global c#This is global c
    print("Global C:",c)
    c=50#this is local c
    print("Local C:",c)
myfunction()
```

Output

```
Global B: 20
Local A: 10
Global C: 30
Local C: 50
>>>
```

RECURSION

A recursive function is one that invokes itself. Or A recursive function is a function that calls itself in its definition.
For example the mathematical function, factorial, defined by factorial(n) = n*(n-1)*(n-2)*...*3*2*1. can be programmed as

```
def factorial(n):
#n here should be an integer
if n == 0:
return 1
else:
     return n*factorial(n-1)
```

Any recursive function can be divided into two parts.

First, there must be one or more **base cases**, to solve the simplest case, which is referred to as the *base case* or the *stopping condition*

Next, **recursive cases**, here function is called with different arguments, which are referred to as a recursive call. These are values that are handled by "reducing" the problem to a "simpler" problem of the same form.

Example: To find factorial using Recursion

```
def main():
    n=int(input("Enter a nonnegative integer: "))
    print("Factorial of", n, "is",factorial(n))
    print(" 0! = ", factorial(0))
    print(" 1! = ", factorial(1))
    print(" 5! = ", factorial(6))
  # Return the factorial for the specified number
def factorial(n):
    if n == 0: # Base case
        return 1
    else:
        return n*factorial(n-1) # Recursive call
main()# call the main
```

Output:

```
Enter a nonnegative integer: 5
Factorial of 5 is 120
0! =  1
1! =  1
5! =  720
```

PROFILING

Profiling means having the application run while keeping track of several different parameters, like the number of times a function is called, the amount of time spent inside it, and so on. Profiling can help us find the bottlenecks in our application, so that we can improve only what is really slowing us down. Python includes a profiler called cProfile. It breaks down your entire script and for each method in your script it tells you:

ncalls: The number of times a method was called

tottime: Total time spent in the given function (excluding time made in calls to sub-functions)

percall: Time spent per call. Or the quotient of tottime divided by ncalls

cumtime: The cumulative time spent in this and all subfunctions (from invocation till exit). This figure is accurate even for recursive functions.

percall: is the quotient of cumtime divided by primitive calls

filename: lineno(function): provides the respective data of each function

MODULES

A module is a library of functions. Modules are code files meant to be used by other programs. Normally files that are used as modules contain only class and function definitions. Modularizing makes code easy to maintain and debug, and enables the code to be reused. To use a module, we use the ***import*** statement. A module can also import other modules.

Example:GCD_LCM

```
# File name GCD_LCM_Module.py
# Return the gcd of two integers
def gcd(n1, n2):
    p = n1 # Initial gcd is 1
    q= n2 # Possible gcd
    while n2!=0:
          r=n1%n2
          n1=n2
          n2=r
    gcd=n1
    lcm=int(p*q/gcd)
    return gcd,lcm # Return gcd and lcm
```

Now we write a separate program to use the above module function, as shown below

File name Test_module

```
# file name Test_module.py
from GCD_LCM_Module import gcd # Import the module
# Prompt the user to enter two integers
n1 = eval(input("Enter the first integer: "))
n2 = eval(input("Enter the second integer: "))
print("The GCD and LCM for", n1,"and", n2, "is", gcd(n1, n2))
```

Output:

```
Enter the first integer: 6
Enter the second integer: 3
The GCD and LCM for 6 and 3 is (3, 6)
>>>
```

The import statement

In order to use the functions and variables of the module1.py program, we will use the import statement. The syntax of the import statement is shown here:

```
import module1, module2, module
```

The statements and definitions of modules are executed for the first time when the interpreter encounters the module name in the import statement.

In preceding code, in order to use the module variables and functions, use the ***module_name.variable*** and ***module_name.function()*** notations.

In order to write module1 with every function of module1.py, Python allows use the ***as*** statement as shown. The syntax is given as follows:

```
import module_name as new_name
```

Example:module

```
def sum1(a,b):
     c = a+b
```

```
         return c
     def mul1(a,b):
         c = a*b
         return c
```
Filename:Test_module
```
     import module1 #import module_name
     x = 12
     y = 34
     print ("Sum is ", module1.sum1(x,y))
     print ("Multiple is ", module1.mul1(x,y))
```

Filename:Test_module2.py
```
     import module1 as md #import module_name as new_name
     x = 12
     y = 34
     print ("Sum is ", md.sum1(x,y))
     print ("Multiple is ", md.mul1(x,y))
```

ARRAY, LISTS, SETS AND DICTIONARY
Arrays

An array is a data structure that stores values of same data type. In Python, this is the main difference between arrays and lists.

While python lists can contain values corresponding to different data types, arrays in python can only contain values corresponding to same data type.

To use arrays in python language, you need to import the standard array module. This is because array is not a fundamental data type like strings, integer etc. Here is how you can import array module in python:

`from array import *`

Once you have imported the array module, you can declare an array. Here is how you do it:

`arrayIdentifierName = array(typecode, [Initializers])`

Typecode	Details
B	Represents signed integer of size 1 byte
B	Represents unsigned integer of size 1 byte
C	Represents character of size 1 byte
u	Represents unicode character of size 2 bytes
h	Represents signed integer of size 2 bytes
H	Represents unsigned integer of size 2 bytes

i	Represents signed integer of size 2 bytes
I	Represents unsigned integer of size 2 bytes
w	Represents unicode character of size 4 bytes
1	Represents signed integer of size 4 bytes
L	Represents unsigned integer of size 4 bytes
f	Represents floating point of size 4 bytes
D	Represents floating point of size 8 bytes

Example of an array containing 5 integers:

```
from array import *
my_array = array('i', [1,2,3,4,5])
for i in my_array:
print(i)
#output:1,2,3,4,5
```

Some built-in array methods:

Append any value to the array using append() method

```
my_array = array('i', [1,2,3,4,5])
my_array.append(6)
# array('i', [1, 2, 3, 4, 5, 6])
```

Note that the value 6 was appended to the existing array values.

Insert value in an array using insert() method

```
my_array = array('i', [1,2,3,4,5])
my_array.insert(0,0)
#array('i', [0, 1, 2, 3, 4, 5])
```

In the above example, the value 0 was inserted at index 0. Note that the first argument is the index while second argument is the value.

Extend python array using extend() method

```
my_array = array('i', [1,2,3,4,5])
my_extnd_array = array('i', [7,8,9,10])
my_array.extend(my_extnd_array)
# array('i', [1, 2, 3, 4, 5, 7, 8, 9, 10])
```

We see that the array my_array was extended with values from my_extnd_array.

Remove any array element using remove()

```
my_array = array('i', [1,2,3,4,5])
my_array.remove(4)
# array('i', [1, 2, 3, 5])
```

We see that the element 4 was removed from the array.

Remove last array element using pop() method

pop removes the last element from the array.

```
my_array = array('i', [1,2,3,4,5])
```

```
my_array.pop()
# array('i', [1, 2, 3, 4])
```

So we see that the last element (5) was popped out of array.

Fetch any element through its index using index()

index() returns first index of the matching value. Remember that arrays are zero-indexed.

```
my_array = array('i', [1,2,3,4,5])
print(my_array.index(5))
#output: 5
my_array = array('i', [1,2,3,3,5])
print(my_array.index(3))
#output: 3
```

Note in that second example that only one index was returned, even though the value exists twice in the array

Reverse a python array using reverse() method

The reverse() method reverses the array.

```
my_array = array('i', [1,2,3,4,5])
my_array.reverse()
# array('i', [5, 4, 3, 2, 1])
```

Sort a python array using sort() method

```
from array import *
my_array = [1,20,13,4,5]
my_array.sort()
print(my_array)
#output:1,4,5,13,20
```

Multi-Dimensional Array

An array containing more than one row and column is called multidimensional array. It is also called combination of several 1D arrays.2D array is also considered as matrix.

```
A=array([1,2,3,4])# create 1D array with 1 row
B=array([1,2,3,4],[5,6,7,8]) create 2D array with 2 row
```

Example:2D_array

```
from numpy import*
a=array([[1,2,3],[4,5,6],[7,8,9]])
print(a)#Prints 2D array as rows
print("2D Array Element wise Printing")
for i in range(len(a)):
    for j in range(len(a[i])):
        print(a[i][j],end=' ')#Prints array element wise
    print(end='\n')
print(end='\n' )
#2D array As matrix by using matrix fun
print("Matrix printing")
a=matrix('1 2 3; 4 5 6 ; 7 8 9')
print(a)
```

Output

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
2D Array Element wise Printing
1 2 3
4 5 6
7 8 9
Matrix printing
[[1 2 3]
 [4 5 6]
 [7 8 9]]
>>>
```

Matrix in Numpy

In python we can show matrices as 2D array. In numpy, a matrix is considered as specialized 2D array. It has lot of built in operators on 2D matrices. In numpy, matrix is created using the following syntax.

`Matrix_name=matrix(2D array or string)`

Eg. `a=matrix('1 2 3;4 5 6;7 8 8')`

Matrix addition, multiplication and division.

We can use arithmetic operators like +, -,* ,/ to perform different operations on matrices.

Example: Matrix_Operation

```
from numpy import*
a=matrix('4 4 4;4 4 4;4 4 4')
b=matrix('2 2 2;2 2 2;2 2 2')
print("Printing A matrix")
print(a)
print("Printing B matrix")
print(b)
print("Printing Addition of two matrix")
c=a+b #matrix addition
print(c)
print("Printing Multplication of two matrix")
c=a*b #matrix addition
print(c)
print("Printing Division of two matrix")
c=a/b #matrix addition
print(c)
```

Output

```
Printing A matrix
[[4 4 4]
 [4 4 4]
 [4 4 4]]
Printing B matrix
[[2 2 2]
 [2 2 2]
 [2 2 2]]
Printing Addition of two matrix
```

```
[[6 6 6]
 [6 6 6]
 [6 6 6]]
Printing Multplication of two matrix
[[24 24 24]
 [24 24 24]
 [24 24 24]]
Printing Division of two matrix
[[2. 2. 2.]
 [2. 2. 2.]
 [2. 2. 2.]]
>>>
```

Example: #prog to accept matrix from key board and display its transpose.

```
from numpy import*
# Accept Rows and Column of matrix")
r,c=[int(a) for a in input("Enter rows,col:").split()]
str=input("Enter matrix elements:")
#convert the string into amatrix with size r*c
x=reshape(matrix(str),(r,c))
print("The original  matrix")
print(x)
print("Printing Transpose of matrix")
y=x.transpose()
print(y)
```

Output

```
Enter rows,col: 2 2
Enter matrix elements:1 2 3 4
The original matrix
[[1 2]
 [3 4]]
Printing Transpose of matrix
[[1 3]
 [2 4]]
>>>
```

LISTS

A list is an object that holds a collection of objects; it represents a sequence of data. A list can hold any Python object. A list need not be homogeneous; that is, the elements of a list do not all have to be of the same type.

Like any other variable, a list variable can be local or global, and it must be defined (assigned) before its use.

The following are the characteristics of a Python list:

- Values are ordered
- Mutable
- A list can hold any number of values
- A list can add, remove, and alter the values

Creating a list

Let's see how we can create an empty list:

```
<Variable name > = [ ]
```
E.g. `List1 = [ ]`

Creating a list with values

A list contains comma-separated values. For example:

```
a = [1, 2, 3, 4, 5]
Avengers = ['hulk', 'iron-man', 'Captain', 'Thor']
```

List methods and supported operators

Starting with a given list a:

a = [1, 2, 3, 4, 5]

1. append(value) – appends a new element to the end of the list.

```
# Append values 6, 7, and 7 to the list
a.append(6)
a.append(7)
a.append(7)
# a: [1, 2, 3, 4, 5, 6, 7, 7]
# Append an element of a different type, as list elements do not need
to have the same type
my_string = "hello world"
a.append(my_string)
# a: [1, 2, 3, 4, 5, 6, 7, 7, [8, 9], "hello world"]
```

Note that the append() method only appends one new element to the end of the list. If you append a list to another list, the list that you append becomes a single element at the end of the first list.

```
# Appending a list to another list
a = [1, 2, 3, 4, 5, 6, 7, 7]
b = [8, 9]
a.append(b)
# a: [1, 2, 3, 4, 5, 6, 7, 7, [8, 9]]
a[8]
# Returns: [8, 9]
```

2. extend(enumerable) – extends the list by appending elements from another enumerable.

```
a = [1, 2, 3, 4, 5, 6, 7, 7]
b = [8, 9, 10]
# Extend list by appending all elements from b
a.extend(b)
# a: [1, 2, 3, 4, 5, 6, 7, 7, 8, 9, 10]
# Extend list with elements from a non-list enumerable:
a.extend(range(3))
# a: [1, 2, 3, 4, 5, 6, 7, 7, 8, 9, 10, 0, 1, 2]
```

Lists can also be concatenated with the + operator. Note that this does not modify any of the original lists:

```
a = [1, 2, 3, 4, 5, 6] + [7, 7] + b
# a: [1, 2, 3, 4, 5, 6, 7, 7, 8, 9, 10]
```

3. index(value, [startIndex]) – gets the index of the first occurrence of the input value. If the input value is not in the list a ValueError exception is raised. If a second argument is provided, the search is started at that specified index.

```
a.index(7)
```

```
# Returns: 6
a.index(49) # ValueError, because 49 is not in a.
a.index(7, 7)
# Returns: 7
a.index(7, 8) # ValueError, because there is no 7 starting at index 8
```

4. insert(index, value) – inserts value just before the specified index. Thus after the insertion the new element occupies position index.

a.insert(0, 0) # insert 0 at position 0

```
a.insert(2, 5) # insert 5 at position 2
# a: [0, 1, 5, 2, 3, 4, 5, 6, 7, 7, 8, 9, 10]
```

5. pop([index]) – removes and returns the item at index. With no argument it removes and returns the last element of the list.

```
a.pop(2)
# Returns: 5
# a: [0, 1, 2, 3, 4, 5, 6, 7, 7, 8, 9, 10]
a.pop(8)
# Returns: 7
# a: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
# With no argument:
a.pop()
# Returns: 10
# a: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

6. remove(value) – removes the first occurrence of the specified value. If the provided value cannot be found, a ValueError is raised.

```
a.remove(0)
a.remove(9)
# a: [1, 2, 3, 4, 5, 6, 7, 8]
a.remove(10)
# ValueError, because 10 is not in a
```

7. reverse() – reverses the list in-place and returns None.

```
a.reverse()
# a: [8, 7, 6, 5, 4, 3, 2, 1]
```

8. count(value) – counts the number of occurrences of some value in the list.

```
a.count(7)
# Returns: 2
```

9. sort() – sorts the list in numerical and lexicographical order and returns None.

```
a.sort()
# a = [1, 2, 3, 4, 5, 6, 7, 8]
# Sorts the list in numerical order
```

10. The slicing of a list. Syntax: <list-name> [start: stop: step]

```
     a = [1, 2, 3, 4, 5, 6, 7, 8]
     print(a[2:5]) #Output: [3, 4, 5]
```

Example:#to demonstrate lists

```
     NUMBER_OF_ELEMENTS = 5
     # Create an empty list
     numbers=[]
     sum = 0
     for i in range(NUMBER_OF_ELEMENTS):
         value = eval(input("Enter a new number: "))
         numbers.append(value)
```

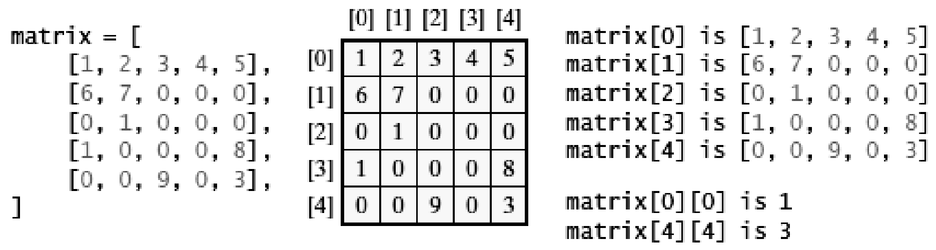
```
        sum += value
    average = sum / NUMBER_OF_ELEMENTS
    count = 0 # The number of elements above average
    for i in range(NUMBER_OF_ELEMENTS):
        if numbers[i] > average:
            count += 1
    print("Average is", average)
    print("Number of elements above the average is", count)
    numbers.append(value)
```

Multidimensional Lists

Data in a table or a matrix can be stored in a two-dimensional list. ***A two-dimensional list is a list that contains other lists as its elements.***

A value in a two-dimensional list can be accessed through a row and column index. You can think of a two-dimensional list as a list that consists of rows. Each row is a list that contains the values. The rows can be accessed using the index, conveniently called a row index. The values in each row can be accessed through another index, called a column index.

A two-dimensional list named matrix is illustrated in Figure.



```
matrix = [
    [1, 2, 3, 4, 5],
    [6, 7, 0, 0, 0],
    [0, 1, 0, 0, 0],
    [1, 0, 0, 0, 8],
    [0, 0, 9, 0, 3],
]
```

	[0]	[1]	[2]	[3]	[4]
[0]	1	2	3	4	5
[1]	6	7	0	0	0
[2]	0	1	0	0	0
[3]	1	0	0	0	8
[4]	0	0	9	0	3

```
matrix[0] is [1, 2, 3, 4, 5]
matrix[1] is [6, 7, 0, 0, 0]
matrix[2] is [0, 1, 0, 0, 0]
matrix[3] is [1, 0, 0, 0, 8]
matrix[4] is [0, 0, 9, 0, 3]

matrix[0][0] is 1
matrix[4][4] is 3
```

Example: Initializing Lists with Input Values:

```
    #The following loop initializes the matrix with user input values:
    matrix = [] # Create an empty list
    numberOfRows = eval(input("Enter the number of rows: "))
    numberOfColumns = eval(input("Enter the number of columns: "))
    for row in range(numberOfRows):
        matrix.append([]) # Add an empty new row
        for column in range(numberOfColumns):
            value = eval(input("Enter an element and press Enter: "))
            matrix[row].append(value)
    print("Printing Matrix....")
    for row in range(len(matrix)):
        for column in range(len(matrix[row])):
            print(matrix[row][column], end = " ")
        print() # Print a new line
```

Output:

```
    Enter the number of rows: 2
    Enter the number of columns: 2
    Enter an element and press Enter: 1
    Enter an element and press Enter: 2
    Enter an element and press Enter: 3
    Enter an element and press Enter: 4
    Printing Matrix....
    1 2
    3 4
    >>>
```

SETS

Sets are like lists in that you use them for storing a collection of elements. Unlike lists,
the elements in a set are non-duplicates and are not placed in any particular order.

Creating Sets

Python provides a data structure that represents a mathematical set. As with mathematical sets, elements of Set are enclosed inside a pair of curly braces ({ }). The elements are non-duplicate, separated by commas. You can create an empty set, or you can create a set from a list or a tuple, as shown in the following examples:
s1 = set( ) # Create an empty set
s2 = {1, 3, 5} # Create a set with three elements
s3 = set([1, 3, 5]) # Create a set from a tuple

Example: Program to convert Tuple into Set:

```
#We can make a set out of a list using the set conversion
function:
L = [10, 13, 10, 5, 6, 13, 2, 10, 5]
S=set(L)
print(L)
print(S)
{10, 2, 13, 5, 6}
```

Operation	Mathematical Notation	Python Syntax	Result Type	Meaning
Union	$A \cup B$	`A | B`	set	Elements in A or B or both
Intersection	$A \cap B$	`A & B`	set	Elements common to both A and B
Set Difference	$A - B$	`A - B`	set	Elements in A but not in B
Symmetric Difference	$A \oplus B$	`A ^ B`	set	Elements in A or B, but not both
Set Membership	$x \in A$	`x in A`	bool	x is a member of A
Set Membership	$x \notin A$	`x not in A`	bool	x is not a member of A
Set Equality	$A = B$	`A == B`	bool	Sets A and B contain exactly the same elements
Subset	$A \subseteq B$	`A <= B`	bool	Every element in set A also is a member of set B
Proper Subset	$A \subset B$	`A < B`	bool	A is a subset B, but B contains at least one element not in A

Example: Operations on Sets:

```
s1 = {1, 2, 4}
s1.add(6)
print(s1)#prints {1, 2, 4, 6}
print(len(s1))
#prints 4
print(max(s1))
#prints 6
print(min(s1))
#prints 1
print(sum(s1))
```

```
#prints 13
print(3 in s1)
#prints False
s1.remove(4)
print(s1)
#prints {1,2,6)
s1 = {1, 2, 4}
s2 = {1, 4, 5, 2, 6}
print(s1.issubset(s2)) # s1 is a subset of s2
#prints True
s1 = {1, 2, 4}
s2 = {1, 4, 5, 2, 6}
print(s2.issuperset(s1)) # s2 is a superset of s1
#prints True
s1 = {1, 2, 4}
s2 = {1, 4, 2}
print(s1 == s2)#prints True
print(s1 != s2)#prints False
```

Set Operations

Python provides the methods for performing set **union, intersection, difference, and symmetric difference operations.**

Example:

```
s1 = {1, 4, 5, 6}
s2 = {1, 3, 6, 7}
print(s1.union(s2))
print(s1 | s2)
print(s1.intersection(s2))
print(s1 & s2)
print(s1.difference(s2))
print(s1 - s2)
print(s1.symmetric_difference(s2))
print(s1 ^ s2)
```

Output:

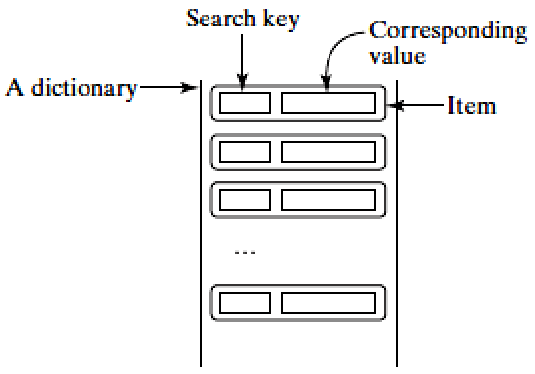
```
{1, 3, 4, 5, 6, 7}
{1, 3, 4, 5, 6, 7}
{1, 6}
{1, 6}
{4, 5}
{4, 5}
{3, 4, 5, 7}
{3, 4, 5, 7}
>>>
```

DICTIONARIES

A dictionary is a container object that stores a collection of key/value pairs. It enables fast retrieval, deletion, and updating of the value by using the key.
A dictionary is a collection that stores the values along with the keys. The keys are like an index operator. In a list, the indexes are integers. A dictionary cannot contain duplicate keys. Each key maps to one value. A key and its corresponding value form an item (or entry) stored in a dictionary, as shown in Figure. The data structure is a called a "dictionary" because it resembles a word dictionary, where the words are the keys and the words' definitions is the values. A dictionary is also known as a map, which maps each key to a value.

The syntax of a dictionary is as follows:

```
Dictionary_name = {key: value}
```



Example:
port = {22: "SSH", 23: "Telnet" , 53: "DNS", 80: HTTP" }
The **port** variable refers to a dictionary that contains port numbers as keys and its protocol names as values.
Consider the following example:
Companies = {"IBM": "International Business Machines", "L&T" :"Larsen & Toubro"}

Characteristics of dictionary:

- The keys in a dictionary may have different types(string, int, or float)
- The values in a dictionary may have different types
- The values in a dictionary may be mutable objects
- The key of the dictionary cannot be changed i.e. Keys are unique
- Values can be anything, for example, list, string, int, and so on
- Values can be repeated
- Values can be changed
- A dictionary is an unordered collection, which means that the order in which you have entered the items in a dictionary may not be retained and you may get the items in a different order i.e. the order of key: value pairs in a dictionary are independent of the order of their insertion into the dictionary

Operations on the dictionary

A dictionary is mutable; we can add new values, and delete and update old values.

Accessing the values of dictionary

The dictionary's values are accessed by using key. Consider a dictionary of networking ports: In order to access the dictionary's values, the key is considered.

```
Port = {80: "HTTP", 23: "Telnet", 443: "HTTPS"}
print(port[80])
>>'HTTP'
print(port[443])
>>'HTTPS'
```

If the key is not found, then the interpreter shows the preceding error.

Deleting an item from the dictionary

del keyword is used to delete the entire dictionary or the dictionary's items. Syntax to delete dictionary's item:

`del dict[key]`

Considering the following code snippet for example:

```
>>> port = {80: "HTTP", 23 : "Telnet", 443 : "HTTPS"}
>>> del port[23]
>>> print(port)
      {80: 'HTTP', 443: 'HTTPS'}
>>>
```

Syntax to delete the entire dictionary:

`del dict_name`

Consider the following example:

```
>>> port = {80: "HTTP", 23 : "Telnet", 443 : "HTTPS"}
>>> del port
>>> print(port)
     Traceback (most recent call last):
     File "<pyshell#12>", line 1, in <module>
     port
     NameError: name 'port' is not defined
>>>
```

The preceding error shows that the port dictionary has been deleted.

Updating the values of the dictionary

To update the dictionary, just specify the key in the square bracket along with the dictionary name and assigning new value. The syntax is as follows:

`dict[key] = new_value`

Example:

port = {80: "HTTP", 23 : "SMTP", 443 : "HTTPS"}

In the preceding dictionary, the value of port 23 is "SMTP", but in reality, port number 23 is for telnet protocol. Let's update the preceding dictionary with the following code:

```
>>> port = {80: "HTTP", 23 : "SMTP", 443 : "HTTPS"}
>>>print( port)
     {80: 'HTTP', 443: 'HTTPS', 23: 'SMTP'}
>>> port[23] = "Telnet"
>>> print(port)
      {80: 'HTTP', 443: 'HTTPS', 23: 'Telnet'}
>>>
```

Adding an item to the dictionary

Item can be added to the dictionary just by specifying a new key in the square brackets along with the dictionary. The syntax is as follows:

`dict[new_key] = value`

Example:

```
>>> port = {80: "HTTP", 23 : "Telnet"}
>>> port[110]="POP"
>>> print(port)
   {80: 'HTTP', 110: 'POP', 23: 'Telnet'}
>>>
```

Other dictionary functions:
Similar to lists and tuples, built-in functions available for dictionary.
len()- to find the number of items that are present in a dictionary.
Example:
```
>>> port = {80: "http", 443: "https", 23:"telnet"}
>>> print(len(port))
    3
>>>
```
max()-It returns the key with the maximum worth.
Example:
```
>>>  dict1  =  {1:"abc",5:"hj",  43:"Dhoni",  ("a","b"):"game",
"hj":56}
>>> max(dict1)
     ('a', 'b')
```
min()- It returns the dictionary's key with the lowest worth. The syntax of the method is as follows:
Example:
```
>>> dict1={1: 'abc', (1, 3): 'kl', 5: 'hj', 43: 'Dhoni', 'hj':
56}
>>> min(dict1)
   1
>>>
```

FILES

Files are used to store data permanently. Data used in a program is temporary; unless the data is specifically saved, it is lost when the program terminates. To permanently store the data created in a program, you need to save it in a file on a disk or some other permanent storage device. The file can be transported and can be read later by other programs.

Python’s standard library has a file class that makes it easy for programmers to make objects that can store data to, and retrieves data from, disk

In order to read and write into a file, we will use the open() built-in function to open the file. The open() function creates an file_object object.
The Syntax is:

```
file_object = open(file_name ,access_mode)
```

The first argument,file_name, specifies the filename that is to be opened. The second argument, access_mode, determines in which mode the file has to be opened, that is, read, write and append.

Mode	Description
"r"	Opens a file for reading.
"w"	Opens a new file for writing. If the file already exists, its old contents are destroyed.
"a"	Opens a file for appending data from the end of the file.
"rb"	Opens a file for reading binary data.
"wb"	Opens a file for writing binary data.
"r+"	Opens a file for reading and writing.
"w+"	Opens a file for reading and writing. If the file doesn't exist, then a new file is created.

Examples:

```
f = open('myfile.txt', 'r')
```

Creates a file object named f capable of reading the contents of the text file named myfile.txt. in current directory.

```
f = open('myfile.txt', 'w')
```

Creates and returns a file object named f capable of writing data to the text file named myfile.txt.

We can also use the absolute path to filename to open the file in Windows, as follows:

```
f = open("c:\\pybook\\myfile.txt", "r")
```

Creates and returns a file object named f capable of reading data to the text file named myfile.txt in folder name python in c drive

```
f = open('myfile.txt', 'a')
```

New data will be appended after the pre-existing data in that file.

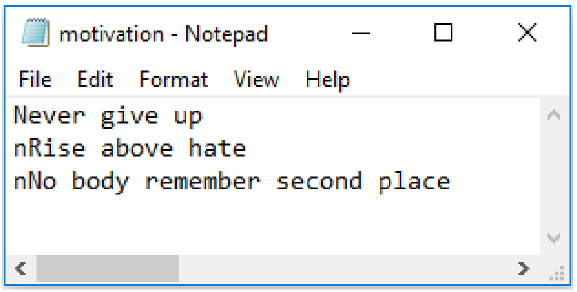
Writing text to a file

The 'w' mode creates a new file. If the file already exists, then the file would be overwritten. We will use the **write()** function.

Example:

```
file_input = open("motivation.txt",'w')
file_input.write("Never give up\n")
file_input.write("Rise above hate\n")
file_input.write("No body remember second place")
file_input.close()
```

Output:

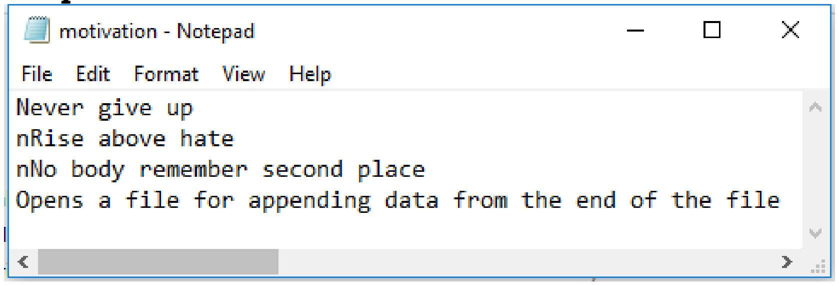


```
Never give up
nRise above hate
nNo body remember second place
```

Example: with access mode 'a'.

```
file_input = open("motivation.txt",'a')
file_input.write("Opens a file for appending data from the end of the
file")
file_input.close()
```

Output:



```
Never give up
nRise above hate
nNo body remember second place
Opens a file for appending data from the end of the file
```

Reading Data

After a file is opened for reading data, you can use the **read()** method to read a specified number of characters or all characters from the file and return them as a string, the **readline()** method to read the next line, and the **readlines()** method to read all the lines into a list of strings.

Example to read data

```
def  main():
#Open file for input
infile = open("motivation.txt", "r")
print("(1) Using read():")
print(infile.read())#read all data
infile.close() # Close the input file
# Open file for input
infile = open("motivation.txt", "r")
print("\n(2) Using read(number): ")
s1 =infile.read(6)#reads character
print(s1)
infile.close() # Close the input file
#Open file for input
infile = open("motivation.txt", "r")
print("\n(3) Using readline(): ")
line1 =infile.readline()#reads line
print(repr(line1))
infile.close() # Close the input file
```

```
        infile = open("motivation.txt", "r")
        # Open file for input
        infile = open("motivation.txt", "r")
        print("\n(4) Using readlines(): ")
        print(infile.readlines() )#prints lines to list
        infile.close() # Close the input file
        main() # Call the main functionample:Read_file.py
```

EXCEPTION HANDLING

Exception handling enables a program to deal with exceptions and continue its normal execution**.**

The run-time exceptions immediately terminate a running program. Rather than terminating the program's execution, an executing program can detect the problem when it arises and possibly execute code to correct the issue or soften it in some way. This chapter explores handling exceptions in Python.

An error that occurs at runtime is also called an exception. The run-time exceptions immediately terminate a running program. Python provides a standard mechanism called exception handling that allows the program to catch the error and prompt the user to correct the issue or soften it in some way

This can be done using Python's exception handling syntax.

The syntax for exception handling is to wrap the code that might raise (or throw) an exception in a try clause, as follows:

```
try:
<body>
except <ExceptionType>:
<handler>
```

Here, <body> contains the code that may raise an exception. When an exception occurs, the rest of the code in <body> is skipped. If the exception matches an exception type, the corresponding handler is executed. <handler> is the code that processes the exception.

Example:

```
 def divide(a,b):
     try:
     c = a/b
     return c
     except :
     print ("Error in divide function")
print(divide(10,0))#function call
```

Output

```
Error in divide function
None
>>>
```

Common standard exception classes

Class	**Meaning**
AttributeError	Object does not contain the specified instance variable or method
ImportError	When import statement fails to find a specified module or name
IndexError	A sequence (list, string, tuple) index is out of range
KeyError	Specified key does not appear in a dictionary
NameError	Specified local or global name does not exist
TypeError	Operation or function applied to an inappropriate type
ValueError	Operation or function applied to correct type but inappropriate value
ZeroDivisionError	Second operand of divison or modulus operation is zero

A try with multiple except clause

A try statement can have more than one except clause to handle different exceptions.

The statement can also have an optional else and/or finally statement, in a syntax like this:

```
try:
<body>
except <ExceptionType1>:
<handler1>
 ...
except <ExceptionTypeN>:
<handlerN>
except:
<handlerExcept>
 else:
<process_else>
finally:
<process_finally>
```

The multiple excepts are similar to elifs. When an exception occurs, it is checked to match an exception in an except clause after the try clause sequentially. If a match is found, the handler for the matching case is executed and the rest of the except clauses are skipped.

Note that the <ExceptionType> in the last except clause may be omitted. If the exception does not match any of the exception types before the last except clause, the

<handlerExcept> for the last except clause is executed.

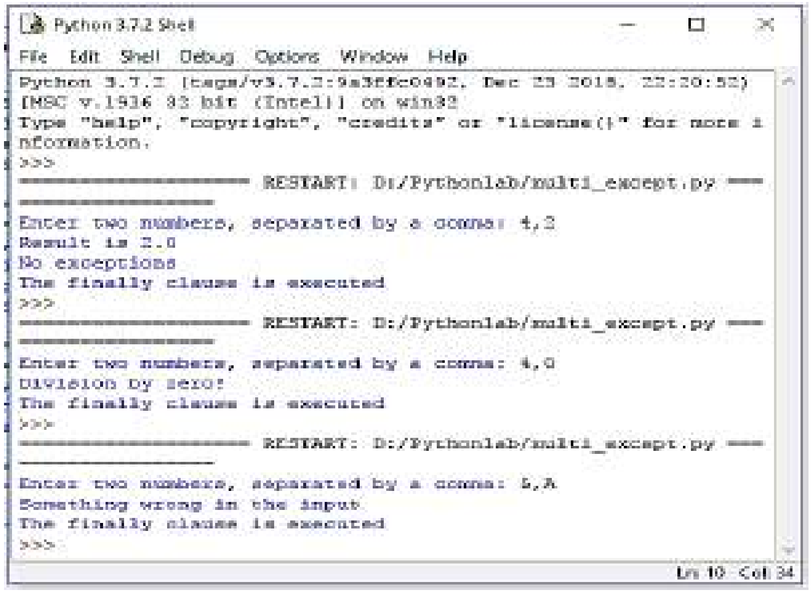
A try statement may have an optional else clause, which is executed if no exception is raised in the try body.

A try statement may have an optional finally clause, which is intended to define clean-up actions that must be performed under all circumstances.

Ex : Multiple excepts

```
     def main():
          try:
          number1, number2 = eval(input("Enter two numbers, separated
          by
                    a comma: "))
          result = number1 / number2
          print("Result is", result)
          except ZeroDivisionError:
               print("Division by zero!")
          except SyntaxError:
               print("A comma may be missing in the input")
          except:
               print("Something wrong in the input")
          else:
               print("No exceptions")
          finally:
               print("The finally clause is executed")
     main() # Call the main function
```

Output



```
Python 3.7.2 (tags/v3.7.2:9a3ffc0492, Dec 23 2018, 22:20:52)
[MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more i
nformation.
>>>
==================== RESTART: D:/Pythonlab/multi_except.py ===
==================
Enter two numbers, separated by a comma: 4,2
Result is 2.0
No exceptions
The finally clause is executed
>>>
==================== RESTART: D:/Pythonlab/multi_except.py ===
==================
Enter two numbers, separated by a comma: 4,0
Division by zero!
The finally clause is executed
>>>
==================== RESTART: D:/Pythonlab/multi_except.py ===
==================
Enter two numbers, separated by a comma: 5,A
Something wrong in the input
The finally clause is executed
>>>
```

The try...finally statement

A try statement may include an optional finally block. Code within a finally block always executes whether the try block raises an exception or not. A finally block usually contains “clean-up code” that must execute due to activity initiated in the try block.

The syntax is as follows:

```
try:
#run this action first
except:
# Run if exception occurs
Finally :
#Always run this code
```

The order of the statement should be:

```
try -> except -> else -> finally
```

Example: to demonstrate finally

```
def main():
    try:
        num = int(input("Enter the number "))
        re = 100/num
    except:
        print ("Something is wrong")
    else:
        print ("result is ",re)
    finally :
        print ("finally program ends")
main()
```

Output

```
Enter the number 15
result is  6.666666666666667
finally program ends
>>>
```

MULTITHREADED PROGRAMMING

Computer programs are purely executable, binary (or otherwise), which reside on disk. They do not take on a life of their own until loaded into memory and invoked by the operating system. A process (sometimes called a heavyweight process) is a program in execution. Each process has its own address space, memory, a data stack, and other auxiliary data to keep track of execution. Process switching needs interaction with operating system. The operating system manages the execution of all processes on the system, dividing the time fairly between all processes.

Processes can also fork or spawn(give birth) new processes to perform other tasks, but each new process has its own memory, data stack, etc., and cannot generally share information unless inter process communication (IPC) is employed.

What Are Threads?

Threads (sometimes called lightweight processes) are similar to processes except that they all execute within the same process, thus all share the same context. They can be thought of as "mini-processes" running in parallel within a main process or "main thread."

A thread has a beginning, an execution sequence, and a conclusion. It has an instruction pointer that keeps track of where within its context it is currently running. It can be pre-empted (interrupted) and temporarily put on hold (also known as sleeping) while other threads are running—this is called yielding.

Multiple threads within a process share the same data space with the main thread and can therefore share information or communicate with each other more easily than if they were separate processes. Threads are generally executed in a concurrent fashion, and it is this parallelism and data sharing that enable the coordination of multiple tasks.

Naturally, threads are scheduled in such a way that they run for a little bit, then yield to other threads. Throughout the execution of the entire process, each thread performs its own, separate tasks, and communicates the results with other threads as necessary.

If two or more threads access the same piece of data, inconsistent results may arise because of the ordering of data access. This is commonly known as a race condition. The thread libraries come with synchronization functions which allow the thread manager to control execution and access.

Some threads may not be given equal and fair execution time. This is because some functions block until they have completed.

Multithreading, is the ability of a CPU to manage the use of operating system by executing multiple threads concurrently. The main idea of multithreading is to achieve parallelism by dividing a process into multiple threads.

States of Thread

To understand the functionality of threads in depth, we need to learn about the lifecycle of the threads or the different thread states. Typically, a thread can exist in five distinct states. The different states are shown below:

New Thread

A new thread begins its life cycle in the new state. But, at this stage, it has not yet started and it has not been allocated any resources. We can say that it is just an instance of an object.

Runnable

As the newly born thread is started, the thread becomes runnable i.e. waiting to run. In this state, it has all the resources but still task scheduler have not scheduled it to run.

Running

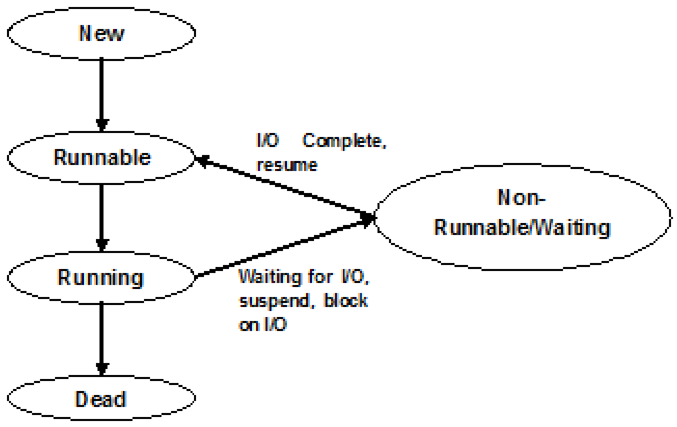
In this state, the thread makes progress and executes the task, which has been chosen by task scheduler to run. Now, the thread can go to either the dead state or the non-runnable/ waiting state.

Non-running/waiting

In this state, the thread is paused because it is either waiting for the response of some I/O request or waiting for the completion of the execution of other thread.

Dead

A runnable thread enters the terminated state when it completes its task or otherwise terminates.



There are two different kinds of threads:

- Kernel threads
- User-space Threads or user threads

Kernel Threads are part of the operating system, while User-space threads are not implemented in the kernel, can be seen as an extension of the function concept of a programming language.

Advantages of Threading:

- Multithreading improve the speed of computation on multiprocessor or multi-core systems because each processor or core handles a separate thread concurrently.
- Multithreading allows a program to remain responsive while one thread waits for input and another runs a GUI at the same time. This statement holds true for both multiprocessor or single processor systems.
- All the threads of a process have access to its global variables. If a global variable changes in one thread, it is visible to other threads as well. A thread can also have its own local variables.

Disadvantages of Threading:

- On a single processor system, multithreading wouldn't impact the speed of computation. In fact, the system's performance may downgrade due to the overhead of managing threads.
- Synchronization is required to avoid mutual exclusion while accessing shared resources of the process. It directly leads to more memory and CPU utilization.

- Multithreading increases the complexity of the program thus also making it difficult to debug.
- It raises the possibility of potential deadlocks.
- It may cause starvation when a thread doesn't get regular access to shared resources. It would then fail to resume its work.

Creating threads in python

Python provides "Thread" class of threading module to create threads. We can create threads in the following different ways.

- Creating threads without using a class
- Creating a thread by creating sub class to thread class
- Creating a thread without creating sub class to Thread class

Creating threads without using a class

In this method,a function is created and its name is passed as target for the thread as

t=Thread(target=function_name,[args=(arg1,arg2,..)

Here **t** is the object of thread class; the target represents the function on which the thread will act.the args represents a tuple of arguments which are passed to the function. the tread is started by calling the start() method as

t.start().

The thread t will execute the function.

Example:

```
      import time
from threading import Thread

def sleepMe(i):
    print("\nThread %i going to sleep for 5 seconds." % i)
    time.sleep(5)
    print("\nThread %i is awake now." % i)

for i in range(5):
    th = Thread(target=sleepMe, args=(i, ))
    th.start()
```

Output:

```
Thread 0 going to sleep for 5 seconds.
Thread 1 going to sleep for 5 seconds.
Thread 2 going to sleep for 5 seconds.
Thread 3 going to sleep for 5 seconds.
Thread 4 going to sleep for 5 seconds.
>>>
Thread 4 is awake now.
Thread 2 is awake now.
Thread 1 is awake now.
Thread 3 is awake now.
Thread 0 is awake now.
```

Creating a thread by creating sub class to thread class

By using Thread class from threading module, we can create our own class as sub class to Thread class, so that we can inherit the functionality of Thread class. By inheriting Thread class we can make use all methods of Thread class into sub class. The Thread class has run() method which is also available to sub class. Every thread will run this method when it is started. By overriding this run we can make threads run our own run() method.

Example:

```
      from threading import Thread
      import time
      class MyThread(Thread):
          def run(self):
              for i in range(1,6):
                  print("\nThread %i going to sleep for 5 seconds." %
      i)
                  time.sleep(5)
                  print("\nThread %i is awake now." % i)
      t1=MyThread()
      t1.start()
      t1.join()
```

Output:

```
      Thread 1 going to sleep for 5 seconds.
      Thread 1 is awake now.
      Thread 2 going to sleep for 5 seconds.
      Thread 2 is awake now.
      Thread 3 going to sleep for 5 seconds.
      Thread 3 is awake now.
      Thread 4 going to sleep for 5 seconds.
      Thread 4 is awake now.
      Thread 5 going to sleep for 5 seconds.
      Thread 5 is awake now.
```

Creating a thread without creating sub class to Thread class

We can ceate an independent class say MYThread that does not inherit from Thread class. Then create object of MyThread .Next step is to create a thread by creating object to Thread class and specifying the method of the My thread class as its target as

T1=Thread(target=obj.Sleep,args( ))

Example:

```
from threading import Thread
      import time
      class MyThread:
          def Sleep(self):
              for i in range(1,6):
                  print("Thread %i going to sleep for 5 seconds." % i)
                  time.sleep(5)
                  print("Thread %i is awake now." % i)
      obj=MyThread()
      t1=Thread(target=obj.Sleep())
      t1.start()
```

Output:

```
      Thread 1 going to sleep for 5 seconds.
      Thread 1 is awake now.
      Thread 2 going to sleep for 5 seconds.
```

```
 Thread 2 is awake now.
 Thread 3 going to sleep for 5 seconds.
 Thread 3 is awake now.
 Thread 4 going to sleep for 5 seconds.
 Thread 4 is awake now.
 Thread 5 going to sleep for 5 seconds.
Thread 5 is awake now
```

Thread synchronization

It is defined as a mechanism which ensures that two or more concurrent threads do not simultaneously execute some particular program segment known as critical section.
Critical section refers to the parts of the program where the shared resource is accessed.
When a thread is already acting on an object, preventing any other thread from acting on the same object is called is called thread synchronization. It is implemented using the following techniques:

- Locks
- Semaphores.

Locks are used to lock objects on which the thread is acting, after completion of execution, it will unlock the object and comes out.
Locks is created by creating object of class Lock as shown below.
L=Lock().
To lock the object,we should use the acquire() as L.axquire().
To unlock or release the object,release() method is used as L.release( )

*****

Chapter-3
Object Oriented Programming

The concept of object-oriented programming was seen to solve many problems, which procedural programming did not solve. In object-oriented programming, everything is just like a real-world object. In the real world, everything is an object. An object can have state and behavior. An object in the real world can communicate with another object.
For example, a dog object in the real world has state and behavior. OOPS is based on four pillars. They are:
Polymorphism
Inheritance
Abstraction
Encapsulation
Class and Objects

Key concepts

Class: A class defines the properties and behaviors for objects. Class is considered as a blueprint for object creation. It provides a template for creating an object and specifying its behavior through means of methods and state through means of variable instance name.
Objects: An object represents an entity in the real world that can be distinctly identified. An object has a unique identity, state, and behavior or attributes. They can be considered as an instance of a class. For example, a student, a desk, a circle, a button, and even a loan can all be viewed as objects.
Inheritance: In object-oriented programming, the child class can inherit many properties from the parent class. Here, the child class can use an existing method or behavior, which the parent class has defined and use them accordingly in their class. Inheritance can be a single inheritance or multiple inheritance. Single inheritance, as the name suggests, refers to only one parent, while multiple inheritance refers to inheriting the property from multiple parents.
Polymorphism: In OOPs, an object can have many forms through means of different attributes. To simplify, in our case, we can understand it by methods with the same name but having different outputs.
Abstraction: Here, we hide the necessary details and are only interested in showing the relevant details to the other proposed user.
Encapsulation: This refers to hiding the necessary methods and their relevant details from the outside world. A class can be treated as a best example, which provides encapsulation to the methods and relevant instances.

Creating a class

Creating a class in Python is quite easy. Refer to the following syntax:

```
class <class name >(<parent class name>):#statements must be indented
      <method definition-1>
      <method definition-n>
```

Methods

The class functions are known by common name, *methods.* In Python, methods are defined as part of the class definition and are invoked only by an instance. To call a method we have to : (1) define the class (and the methods), (2) create an instance, and finally, (3) invoke the method from that instance. Here is an example class with a method:

```
class MyMethod:#define the class
    def display(self): # define the method
        print ('Welcome! to class MyMethod')
obj=MyMethod()
obj.display()
```

self is a parameter that references the object itself. Using self, you can access object's members in a class definition.

The **self** is a parameter that references the object itself. Using self, you can access object's members in a class definition. The ***self*** argument, which must be present in all method invocations. That argument, represents the instance object, is passed to the method implicitly by the interpreter when you invoke a method via an instance.

For example, you can use the syntax self.x to access the instance variable x and syntax self.m1() to invoke the instance method m1 for the object self in a class .

To invoke the method we have to instantiate the class and call the method as follows.

```
>>> myObj = MyMethod() # create the instance
>>> myObj.display() # now invoke the method
```

Constructor (The __init__ method)

Here __init__ works as the class's constructor. When a user instantiates the class, it runs automatically. A class provides a special method, **__init__**. This method, known as an *initializer*, is invoked to initialize a new object's state when it is created. An initializer can perform any action, but initializers are designed to perform initializing actions, such as creating an object's data fields with initial values.

Python uses the following syntax to define a class:

```
class ClassName:
initializer
methods
```

You can access the object's data fields and invoke its methods by using the *dot operator* (**.**), also known as the *object member access operator*.

Example: Construct1.py

```
import math
class Circle:
     # Construct a circle object
    def __init__(self, radius = 1):
        self.radius = radius
    def getPerimeter(self):
        return 2 * self.radius * math.pi
    def getArea(self):
        return self.radius * self.radius * math.pi
    def setRadius(self, radius):
        self.radius = radius
c = Circle(5)
print(c.radius)
print(c.getPerimeter())
print(c.getArea())
c.setRadius(6)
print(c.radius)
print(c.getPerimeter())
print(c.getArea())
```

Output:

```
5
31.41592653589793
78.53981633974483
6
37.69911184307752
113.09733552923255
```

Example: Construct2.py

```
class Car():
    """A simple attempt to represent a car."""
    def __init__(self, make, model, year):
        """Initialize attributes to describe a car."""
        self.make = make
        self.model = model
        self.year = year
    def get_name(self):
        """Return a neatly formatted descriptive name."""
        name = str(self.year) + ' ' + self.make + ' ' + self.model
        return name
my_new_car = Car('audi', 'a4', 2016)
print(my_new_car.get_name())
```

Output:

```
2016 audi a4
```

Destructor (__del__() Method)

Like constructor, there is an equivalent destructor special method called **__del__()**. However, due to the way Python manages garbage collection of objects, this function is not executed until all references to an instance object have been removed. Destructors in Python are methods which provide special processing before instances are de_allocated and are not commonly implemented since instances are rarely de_allocated explicitly.

Example:Destructor.py

```
class myClass:
    count = 0 # use static data for count
    def __init__(self): # constructor, incr. count
        myClass.count = myClass.count + 1
    def __del__(self): # destructor, decr. count
        myClass.count = myClass.count - 1
    def howMany(self): # return count
        return myClass.count
```

```
a = myClass()
b = myClass()
print(b.howMany())
print(a.howMany())
del b
print(a.howMany())
#print(b.howMany())Prints error after deletion of object b
```

Output:

```
2
2
1
```

Class variables

Class variables are the ones, which are sharable among all the instances of the class. The class variable must be the same for all the instances.

Class Attributes

An attribute is a data or functional element which belongs to another object and is accessed via the familiar dotted-attribute notation. Some Python types such as complex numbers have data attributes (real and imag), while others such as lists and dictionaries have methods (functional attributes).

Class Data Attributes

Data attributes are simply variables of the class we are defining. They can be used like any other variable in that they are set when the class is created and can be updated either by methods within the class or elsewhere in the main part of the program.
Such attributes are better known to OO programmers as static members, class variables, or static data. They represent data that is tied to the class object they belong to and are
independent of any class instances.

Class Methods Attributes

A method, in the class is simply a function defined as part of a class definition (thus making methods class attributes). This means that Method applies only to objects (instances) of Class type. Note is tied to its instance because invocation requires both names in the dotted notation:

Example:Class_var.py

```
class Tumkur_org():
    mul_num = 1.20  #class variable
    def __init__(self,first,last,pay):
        self.f_name = first # Class Data Attributes
        self.l_name = last
        self.pay_amt = pay
        self.full_name = first+" "+last
    def make_email(self): # Class Method Attributes
        return self.f_name+ "."+self.l_name+"@gmail.com"
    def increment(self):
        self.pay_amt = int(self.pay_amt*self.mul_num)
        return self.pay_amt
obj1 = Tumkur_org('Hari', 'Das', 60000)
```

```
obj2 = Tumkur_org('Mahesh', 'Span',70000)
print(obj1.full_name)# dotted notation
print(obj1.make_email())
print(obj1.increment())
print(obj2.full_name)
print(obj2.make_email())
print(obj2.increment())
```

Output:

```
Hari Das
Hari.Das@gmail.com
72000
Mahesh Span
Mahesh.Span@gmail.com
84000
```

Inheritance

Inheritance in Python is based on similar ideas used in other object oriented languages like Java, C++ etc. Inheritance allows us to inherit methods and attributes of the parent class. By inheritance, a new child class automatically gets all of the methods and attributes of the existing parent class. The syntax is given as follows:

```
class BaseClass(object):
pass

class DerivedClass(BaseClass):
<statement-1>
.
. .
<statement-N>
```

The BaseClass is the already existing (parent) class, and the DerivedClass is the new (child) class that inherits (or subclasses) attributes from BaseClass.

OR

Inheritance enables you to define a general class (a superclass) and later extend it to more specialized classes (subclasses).

Example:Simple_Inheritance.py

```
class Rectangle():
    def __init__(self, w, h):
        self.w = w
        self.h = h
    def area(self):
        return self.w * self.h

class Square(Rectangle):
    def __init__(self, l,b):
    # call parent constructor.
        super().__init__(l, b)
        self.L=l
        self.B=b
    def perimeter(self):
        return 2 * (self.L + self.B)
sqr=Square(5,6)
print(sqr.area())
print(sqr.perimeter())
```

The **Square** class will automatically inherit all attributes of the Rectangle class as well as the object class.

super() refers to the superclass. Using super() we avoid referring the superclass explicitly. **super()** is used to call the **__init__( )** method of Rectangle class, essentially calling any overridden method of the base class. When invoking a method using super(), don't pass self in the argument.

Types of Inheritance:

- Single Inheritance
- Multiple Inheritances

In Multiple inheritance a subclass inherited from more than one parent class and is able to access functionality from both of them.

Syntax:

```
class A:
   # variable of class A
   # functions of class A

class B:
   # variable of class A
   # functions of class A

class C(A, B):
   # class C inheriting property of both class A and B
   # add more properties to class C
```

- Multilevel Inheritance

```
class A:
   # properties of class A

class B(A):
   # class B inheriting property of class A
   # properties of class B

class C(B):
   # class C inheriting property of class B
   # class C also inherits properties of class A
   # properties of class C
```

Overriding methods

Overriding methods allows a user to override the parent class method. That is use the same method both in base and child class. The name of the method must be the same in the parent class and the child class with different implementations.

Example:classover1.py:

```
class A():
    def sum1(self,a,b):
        print("In class A")
        c = a+b
        return c
class B(A):
    def sum1(self,a,b):
        print("In class B")
        c= a*a+b*b
        return c
a_obj=A()
print(a_obj.sum1(4,5))
b_obj = B()
print(b_obj.sum1(4,5))
```

Output:

```
In class A
9
In class B
41
```

In the preceding example, classes A and B both have the same method sum1() with different implementations.

Namespace

A **namespace** is mapping from names to objects. Examples are the set of built-in names (containing functions that are always accessible for free in any Python program), the global names in a module, and the local names in a function. Even the set of attributes of an object can be considered a namespace.

The namespaces allow defining and organizing names with clarity, without overlapping or interference.

For example, the namespace associated with that book we were looking for in the library can be used to import the book itself, like this:

from library.second_floor.section_x.row_three import book

We start from the library namespace, and by means of the dot (.) operator, we navigate that namespace. Within this namespace, we look for second_floor, and again we navigate with dot(.) operator. We then walk into section_x, and finally within the last namespace, row_three, we find the name we were looking for: book.

Regular Expressions

● The Regular Expression Module A regular expression is a compact notation for representing a collection of strings. What makes regular expressions so powerful is that a single regular expression can represent an unlimited number of strings—

providing they meet the regular expression's requirements. Regular expressions (which we will mostly call "regexes" from now on) are defined using a mini-language that is completely different from Python—but Python includes the re module through which we can seamlessly create and use regexes.★Regexes are used for five main purposes:

• Parsing: identifying and extracting pieces of text that match certain criteria—regexes are used for creating ad hoc parsers and also by traditional parsing tools

• Searching: locating substrings that can have more than one form, for example, finding any of "pet.png", "pet.jpg", "pet.jpeg", or "pet.svg" while avoiding "carpet.png" and similar

• Searching and replacing: replacing everywhere the regex matches with a string, for example, finding "bicycle" or "human powered vehicle" and replacing either with "bike"

• Splitting strings: splitting a string at each place the regex matches, for example, splitting everywhere colon-space or equals (": " or "=") occurs

• Validation: checking whether a piece of text meets some criteria, for example, contains a currency symbol followed by digits

The Python package

Python modules are a single file, whereas a Python package is a collection of modules. A
package is a directory that contains Python modules. A *package* is a module that contains other modules. Some or all of the modules in a package may be sub packages, resulting in a hierarchical tree-like structure.

A **namespace** is mapping from names to objects. Examples are the set of built-in names (containing functions that are always accessible for free in any Python program), the global names in a module, and the local names in a function. Even the set of attributes of an object can be considered a namespace.

The namespaces allow defining and organizing names with clarity, without overlapping or interference.

For example, the namespace associated with that book we were looking for in the library can be used to import the book itself, like this:

from library.second_floor.section_x.row_three import book

We start from the library namespace, and by means of the dot (.) operator, we navigate that namespace. Within this namespace, we look for second_floor, and

again we navigate with dot(.) operator. We then walk into section_x, and finally

within the last namespace, row_three, we find the name we were looking for: book.

*****

Chapter 4
DATABASES

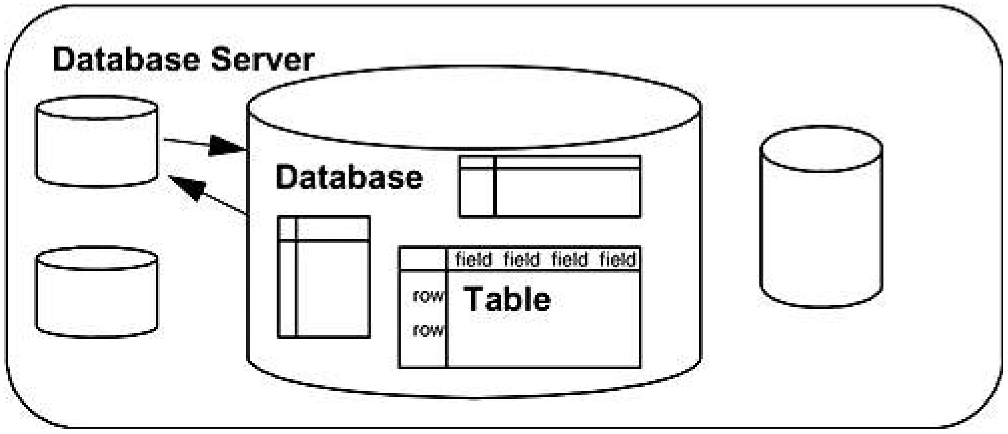
TALKING TO THE DATABASE WITH PYTHON

The RDBMS

RDBMS stands for Relational Database Management System. RDBMS is the basis for SQL, and for all modern database systems like MS SQL Server, IBM DB2, Oracle, MySQL, and Microsoft Access.

A (RDBMS) is a database management system that is based on the relational model as introduced by E. F. Codd. In Relational database model, a table is a collection of data elements organised in terms of rows and columns. Table is the simplest form of data storage. RDBMS is used to manage Relational database. Relational database is a collection of organized set of tables related to each other, and from which data can be accessed easily.

What makes up a database? The main components of an RDBMS are:
a. The database server
b. The database
c. Tables
d. Records and fields
e. Primary key
f. Schema



The database server, the database, and a table.

The Database Server

The database server is the actual server process running the databases. It controls the storage of the data, grants access to users, updates and deletes records, and communicates with other servers. The database server is normally on a dedicated host computer, serving and managing multiple clients over a network, but can also be used as a standalone server on the local host machine to serve a single client (e.g., you

might be the single client using MySQL on your local machine, often referred to as “local host” without any network connection at all).
The requests to the database server can also be made from a program that acts on behalf of a user making requests from a Web page. We can connect to the database server from a PHP program using PHP built-in functions to make requests to the MySQL database server.

The database
A database is a collection of related data elements, usually corresponding to a specific application. The databases are listed as “mysql,” “northwind,” “phpmyadmin,” and “test.”

The tables
Each database consists of two-dimensional tables. In fact, a relational database stores all of its data in tables. All operations are performed on the table, which can then produce other tables, and so on. During designing a database first we create tables and relate these tables to one another in some way. For example, a typical database for an organization might consist of tables for customers, orders, and products and all these tables are related to one another.

The records and fields
A table has a name and consists of a set of rows and columns. It resembles a spread-sheet. Where each row, also called a record. Each table in a database contains zero or more records. The vertical columns, also called fields or attributes and has names. All rows from the same table have the same set of columns. Remember, a relational database manipulates only tables and the result of all operations are also tables. You can view the database itself as a set of tables. You can also perform a number of other operations on tables and also between two tables by treating them as sets: You can join information from two tables, make Cartesian products of the tables, get the intersection between two tables, add one table to another, and so on. SQL allows you to “talk” to a database. A data type can be a number, a character, a date, a time stamp, and so on. *The terms “row” and “record” are often interchangeable, as are “column” and “field.*

The primary key and indexes
A primary key is a unique identifier for each record. These identifiers are unique. In the world of database tables, we call the unique identifier a primary key. Although it is a good idea to have a primary key, not every table has one. The primary key is determined when the table is created

The database schema
Database design is a science and requires understanding how the relational model is implemented. The term database schema, which refers to the structure of the database. It describes the design of the database similar to a template or blueprint; it describes all the tables, and their layout, but does not contain the actual data in the database.

Connecting to the database

To communicate with the MySQL server, you will need a language, and SQL (Structured Query Language) is the language of choice for most modern multiuser, relational databases. SQL provides the syntax and language constructs needed to talk to relational databases in a standardized, cross-platform structured way. We discuss how to use the SQL language.

The version of SQL used by MySQL follows the ANSI (American National Standards Institute) standard, meaning that it must support the major keywords (e.g., SELECT, UPDATE, DELETE, INSERT, WHERE, etc.) as defined in the standard. As you can see by the names of these keywords, SQL is the language that makes it possible to manipulate the data in a database.

First we have to install a database server and run it. There are a number of client applications available to connect to the database server, the most popular and most widely available being the mysql command-line client and also PhpMyAdmin in case of XAMP or WAMP. Regardless of the type of client you choose, you will always need to specify the username, and the host we are connecting to. To connect to a database using the client, we have to enter information similar to the following line
mysql →user→root→password =my_password →host=localhost.
Once you are successfully connected, you will get the mysql> prompt. This means we are connected to the MySQL database server and not to local computer's operating system.

MySQL data types

When creating a table in database, the type and size of each field must be defined. A field is similar to a PHP variable except that it can store only the specified type and size of data in a given field.
MySQL supports three main groups of data types: ***numeric, date/time, and string***. For full details see the MySQL manual at http://dev.mysql.com/doc/.

Numeric data types

You can store numbers in MySQL in many ways, as shown by the following table. Choose the data type most suited for the type of numbers needed to store in table.

Type	**Max value (signed/unsigned)**
TINYINT	– 127 / 255
SMALLINT	– 32,767 to 65,535 /0 to 65535 if UNSIGNED
MEDIUMINT	– 8,388,607 / 16,777,215 or 0 to 16777215 if UNSIGNED
INT	– 2,147,483,647 / 4,294,967,295
BIGINT	– 9,223,372,036,854,775,807 to 18,446,744,073,709,551,615
FLOAT	Smallest non - zero value: ± 1.176 × 10– 38 ; largest value: ± 3.403 × 10 38
FLOAT	Smallest non - zero value: ± 2.225 × 10 – 308 ;

	largest value: ± 1.798 × 10 308
DECIMAL	Same as DOUBLE , but fixed – point rather than floating - point.
BIT	0 or 1

Date and time data types
As with numbers, we can choose from a range of different data types to store dates and times, depending on whether we want to store a date only, a time only, or both:

Date/Time Data Type	Description & Allowed Range of Values
DATE	Date 1 Jan 1000 to 31 Dec 9999
DATETIME	Date and time Midnight, 1 Jan 1000 to 23:59:59, 31 Dec 9999
TIMESTAMP	Timestamp 00:00:01, 1 Jan 1970 to 03:14:07, 9 Jan 2038, UTC (Universal Coordinated Time)
TIME	Time – 838:59:59 to 838:59:59
YEAR	Year 1901 to 2155

String data types
MySQL allows storing text or binary strings of data in many different ways, as shown in the following table:

String Data	Type Description	Allowed Lengths
CHAR(n)	Fixed - length string of n characters	0 – 255 characters
VARCHAR(n)	Variable - length string of up to n characters	0 – 65535 characters
BINARY(n)	Fixed - length binary string of n bytes	0 – 255 bytes
VARBINARY(n)	Variable - length binary string of up to n bytes	(0 – 65535) bytes
MEDIUMTEXT	Medium - sized text field	0 – 16777215 characters

LONGTEXT	Large text field	0 – 4294967295 characters
BLOB	Normal - sized **B**inary **L**arge **OB**ject	0 – 65535 bytes
MEDIUMBLOB	Medium - sized BLOB	0 – 16777215 bytes (16MB)

Introduction to SQL statements

The standard language for communicating with relational databases is SQL, the Structured Query Language. SQL is an ANSI (American National Standards Institute) standard computer language, designed to be as close to the English language as possible, making it an easy language to learn. Popular database management systems such as Oracle, Sybase, and Microsoft SQL Server, all use SQL.To actually work with databases and tables, we SQL statements. Commonly used statements include:

These SQL commands are mainly categorized into three categories as discussed below:

Data Definition Language (DDL):
DDL or Data Definition Language actually consists of the SQL commands that can be used to define the database schema. It simply deals with descriptions of the database schema and is used to create and modify the structure of database objects in database.

Examples of DDL commands:

CREATE: is used to create the database or its objects (like table, index, function, views, store procedure and triggers).
DROP: is used to delete objects from the database.
ALTER: is used to alter the structure of the database.
TRUNCATE: is used to remove all records from a table, including all spaces allocated for the records are removed.
COMMENT: is used to add comments to the data dictionary.
RENAME: is used to rename an object existing in the database.

Data Manipulation Language (DML):
The SQL commands that deals with the manipulation of data present in database belong to DML or Data Manipulation Language and this includes most of the SQL statements.

Examples of DML:

SELECT: Is used to retrieve data from the database.
INSERT: Is used to insert data into a table.
UPDATE: Is used to update existing data within a table.
DELETE: Is used to delete records from a database table.

Data Control Language (DCL):
DCL includes commands such as GRANT and REVOKE which mainly deals with the rights, permissions and other controls of the database system.

Examples of DCL commands:

GRANT: Gives user's access privileges to database.
REVOKE: Withdraw user's access privileges given by using the GRANT command.
COMMIT: Commits a Transaction.
ROLLBACK: Rollbacks a transaction in case of any error occurs.
SAVEPOINT: Sets a save point within a transaction.
SET TRANSACTION: Specify characteristics for the transaction.

Creating a table
Tables are where we actually store your data. To start with, to create a very simple table, ***fruit***, containing three fields: id (the primary key), name (the name of the fruit), and color (the fruit's color).

The first thing to do is select the database just created. Once you a database are selected, any database manipulation commands entered works on that database. Type the following:
USE mydatabase;
Press Enter, and we see:
Database changed
mysql >
Now to create table. Type the following at the mysql > prompt:

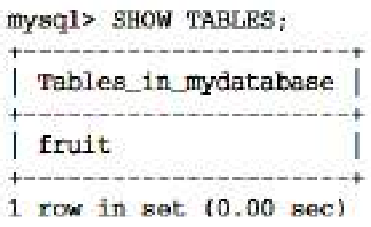
mysql **> CREATE TABLE fruit (id SMALLINT UNSIGNED NOT NULL**

AUTO_INCREMENT, name VARCHAR(30) NOT NULL,

color VARCHAR(30) NOT NULL, PRIMARY KEY (id));

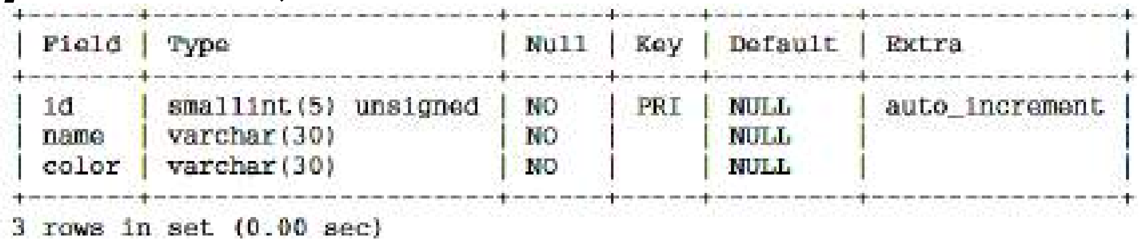
After successful execution of command we get message:
Query OK, 0 rows affected (0.06 sec)

To see a list of tables in the database, use the SHOW TABLES command:
mysql **> SHOW TABLES;**



```
mysql> SHOW TABLES;
+----------------------+
| Tables_in_mydatabase |
+----------------------+
| fruit                |
+----------------------+
1 row in set (0.00 sec)
```

To see the structure of newly created table use EXPLAIN command, as follows:
***mysql* > EXPLAIN fruit;**



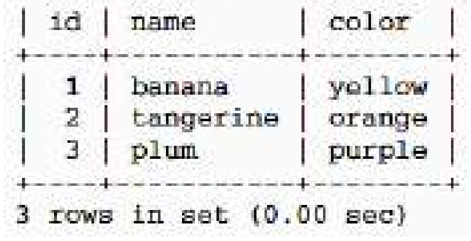
```
+-------+----------------------+------+-----+---------+----------------+
| Field | Type                 | Null | Key | Default | Extra          |
+-------+----------------------+------+-----+---------+----------------+
| id    | smallint(5) unsigned | NO   | PRI | NULL    | auto_increment |
| name  | varchar(30)          | NO   |     | NULL    |                |
| color | varchar(30)          | NO   |     | NULL    |                |
+-------+----------------------+------+-----+---------+----------------+
3 rows in set (0.00 sec)
```

Adding data to a table
Now try adding some data into fruit table. To add a new row to a table, you use the SQL INSERT statement. In its basic form, an INSERT statement looks like this:
INSERT INTO table VALUES (value1 , value2 , ...);
This inserts values into each of the fields of the table, in the order that the fields were created. To insert a row of partial data, use:
INSERT INTO table (field1 , field2 , ...) VALUES (value1 , value2 , ...);
So you can add three rows to the fruit table by inserting data into just the name and color fields (the id field will be filled automatically):

```
mysql >INSERT INTO fruit(name, color)
       VALUES('banana', 'yellow');
      Query OK, 1 row affected (0.06 sec)
mysql >INSERT INTO fruit (name, color)
       VALUES('tangerine', 'orange' );
     Query OK, 1 row affected (0.00 sec)
mysql > INSERT INTO fruit (name, color )
        VALUES ('plum','purple' );
    Query OK, 1 row affected (0.00 sec)
 mysql >
```

Reading data from a table

To read data in SQL, we create a query using the SELECT statement. To retrieve a list of all the data in your fruit table, you can use: Typical form of a MySQL SELECT statement, which retrieves records from a table. Operations performed with SELECT are known as queries (hence the name "Structured Query Language"):
SELECT field1 , field2 , ... , fieldn **FROM *table* WHERE** condition.
To retrieve a list of all the data in fruit table, we can use:
***mysql* > SELECT * from fruit;**



```
+----+-----------+--------+
| id | name      | color  |
+----+-----------+--------+
|  1 | banana    | yellow |
|  2 | tangerine | orange |
|  3 | plum      | purple |
+----+-----------+--------+
3 rows in set (0.00 sec)
```

To retrieve a selected row or rows, we need to introduce a WHERE clause at the end of the SELECT statement. A WHERE clause filters the results according to the condition in the clause. Here is simple WHERE clauses:

mysql **> SELECT * from fruit WHERE name = 'banana';**

```
+----+--------+--------+
| id | name   | color  |
+----+--------+--------+
|  1 | banana | yellow |
+----+--------+--------+
1 row in set (0.08 sec)
```

CONNECTING PYTHON WITH DATABASE

- Install MySQL Connector Python using pip.
- Use the **mysql.connector.connect()** method of MySQL Connector Python with required parameters to connect MySQL.
- Use the connection object returned by a **connect()** method to create a cursor object to perform Database Operations.
- The **cursor.execute()** to execute SQL queries from Python.
- Close the Cursor object using a **cursor.close()** and MySQL database connection using **connection.close()** after your work completes.
- Catch Exception if any that may occur during this process.

Creating a connection object

Now, we are ready to issue commands.

Interacting with the database

Many SQL commands can be issued using a single function as:
cursor.execute()

closing the database

After all changes have been committed, we can then close the database:
mydb.close()

Python MySQL Database connection program

import mysql.connector

- This line imports the MySQL Connector Python module in to program and we can use this module's API to connect MySQL.

from mysql.connector import Error

- **mysql connector Error object** is used to show us an error when we failed to connect Databases or if any other database error occurred while working with the database. Example ER_ACCESS_DENIED_ERROR when username or password is wrong.

mysql.connector.connect()

- Using this function we can connect the MySQL Database, this function accepts four required parameters: **Host, Database, User and Password** that we already discussed.

- **connect()** function establishes a connection to the MySQL database from python application and returns a MySQLConnection object. This process automates logging into the database and selecting a database to be used. Then we can use MySQLConnection object to perform various operation on the MySQL Database.
- **Connect ()** function can throw an exception, i.e. Database error if one of the required parameters are wrong. For example, if you provide a database name that is not present in MySQL, then Python application throws an exception. So check the arguments that you are passing to this function.

The syntax for calling the connect() function and assigning the results to a variable is as follows:

```
[variable] = MySQLdb.connect([hostname], [username], [password],
             [database name])
```

We can assign values these variables as shown but it is not required, but it is good practice until you get used to the format of the function call. So we can use the following format to call the connect() function:

```
[variable] = MySQLdb.connect(host="[hostname]", user="[username]",
       passwd="[password]", db="[database name]")

OR (in case of mysql DB)
[variable] = mysql.connector.connect (host="[hostname]",
       user="[username]", passwd="[password]", db="[database name]")
```

conn.is_connected()

- is_connected() is the function of the MySQLConnection class through which we can verify is our python application connected to MySQL.

Creating a cursor object:

Using a cursor object, we can execute SQL queries. The MySQL Cursor class instantiates objects that can execute operations such as SQL statements. Cursor objects interact with the MySQL server using a MySQLConnection object.

After the connection object is created, cursor object allows to interact with the database. The point of a cursor is to mark place and to allows to issue commands to the computer. A cursor in MySQL for Python serves as a Python-based proxy for the cursor in a MySQL shell session, where MySQL would create the real cursor for us if we logged into a MySQL database.

To create the cursor, we use the cursor( ) method of the MySQLdb.connections object we created for the connection. The syntax is as follows:

```
[cursor name] = [connection object name].cursor()
```

Example:

```
cursor = mydb.cursor()
```

cursor.close()

- Using cursor's close method we can close the cursor object. Once we close the cursor object, we can not execute any SQL statement.

connection.close()

- At last, we are closing the MySQL database connection using a close() function of MySQLConnection class.

Some of Python Database functions

db.close()	Closes the connection to the database (represented by the **db object** which is obtained by calling a connect() function)
db.commit()	Commits any pending transaction to the database; does
	nothing for databases that don't support transactions
db.cursor()	Returns a database cursor object through which queries can
	be executed
db.rollback()	Rolls back any pending transaction to the state that existed
	before the transaction began; does nothing for databases that don't support transactions
db.arraysize	The (readable/writable) number of rows that fetchmany() will return if no size is specified
db.close()	Closes the cursor, c; this is done automatically when the cursor goes out of scope
db.description()	A read-only sequence of 7-tuples (name, type_code, display_size, internal_size, precision, scale, null_ok), describing each successive column of cursor c
db.execute (sql,*params*)	Executes the SQL query in string sql, replacing each placeholder with the corresponding parameter from the *params* sequence or mapping if given
db.executemany (sql,*seq_of_params*)	Executes the SQL query once for each item in the *seq_of_params* sequence of sequences or mappings; this method should not be used for operations that create result sets (such as SELECT statements)
db.fetchall()	Returns a sequence of all the rows that have not yet been fetched (which could be all of them)
db.fetchmany(*size*)	Returns a sequence of rows (each row itself being a

	sequence); *size* defaults to c.arraysize
db.fetchone()	Returns the next row of the query result set as a sequence, or None when the results are exhausted. Raises an exception if there is no result set.
db.rowcount()	The read-only row count for the last operation (e.g.,SELECT, INSERT, UPDATE, or DELETE) or -1 if not available or not applicable

Example:

```
import mysql.connector
conn=mysql.connector.connect(host='localhost',database='test',user='ro
ot',password='')
cur=conn.cursor()
str='''CREATE TABLE IF NOT EXISTS examples( id int(11) NOT NULL
AUTO_INCREMENT, description varchar(45), PRIMARY KEY (id))'''
cur.execute(str)
str2='INSERT INTO examples(description) VALUES ("Hello World")'
cur.execute(str2)
str3='INSERT INTO examples(description) VALUES ("TUMKUR")'
cur.execute(str3)
conn.commit()
cur.execute("Select * from examples")
row=cur.fetchall()
print("total row=",cur.rowcount)
for r in row:
    print(r)
cur.close()
conn.close()
```

Output:

```
total row= 2
(1, 'Hello World')
(2, 'TUMKUR')
>>>
```

Chapter-5
Graphical User Interface

The user can interact with an application through graphics or an image is called GUI [Graphical User Interface]. Here the user need not remember any commands. User can perform task just by clicking on relevant images.

Advantages:

1. It is user friendly.
2. It adds attraction and beauty to any application by adding pictures, colors, menus, animation.
3. It is possible to simulate the real life objects using GUI.
4. GUI helps to create graphical components like push button, radio button, check box, text box, menus etc.,

GUI in Python:

Python offers ***tkinter*** module to create graphical programs. The ***tkinter*** represents 'toolkit interface' for GUI. This is a interface for Python programmers that enable them to the classes of TK module of TCL/TK [Tool Command Language]. TCL language use TK[Tool Kit] language to generate graphics.

General steps involved in basic GUI programs:

1. Create a root Window: The root window is the top level window that provides rectangular space on the screen where we can display text, color, images, etc.,

- First import tkinter module
  from tkinter import *
- root=Tk() # create root window object.

2. Create a Canvas / Frame : Canvas and frame are child windows in the root window.
C=Canvas(root, bg='blue', height=500,width=400,cursor='pencil')
Cursors are – circle, hand1, hand2, heart, pencil, plus, mouse, star, watch etc…

3. Create Widgets – Components :
For Canvas – create line, circle, rectangle, any geometric objects
For Frame – buttons, text box, list , check box etc…

4. Create a Event Handler
Write the function for corresponding event.

WAP to draw line, oval, polygon, rectangle and text

```
from tkinter import *
root=Tk()
root.title ("My Window")
#root.geometry ("400x300")
c=Canvas(root,bg="blue",height=700,width=1200,cursor='pencil')
id=c.create_line(50,50,200,50,200,150,width=4,fill="white")
id=c.create_oval(100,100,400,300,width=2,fill="yellow" , activefill="blue")
id=c.create_polygon(10,10,200,200,300,200,width=3,fill="green",outline="red",activefill="Pink")
id=c.create_rectangle(500,200,700,600,width=2,fill="gray",outline="yellow",activefill="green")
fnt=('Times',40,'bold italic underline')
id=c.create_text(500,100,text="Python Graphics", font=fnt, fill="White", activefill="red")
c.pack()
root.mainloop()
```

Frame: A frame is similar to canvas, but it can hold components of forms. To create a frame, we can create an object of Frame class as :

```
F= Frame(root, height=400,width=500,bg="yellow",cursor="cross"
```

Here, 'F' is an object of class Frame, the options height and width reprsents the area of frame in pixels, 'bg' represents the back ground color to be displayed and 'cursor' indicates the type of the cursor to be displayed in the frame.
Once the frame is created, it should be added to the root window using the **pack()** method.

Widgets/ form elements: Widgets is a GUI component that is displayed on the screen and can perform a task as designed by the user. The widgets are:

1. Button	2. Label	3. Text box	4. Message	
5. Checkbox	6. List box	7. Option button	8. Scroll bars	9. Menus

Steps to create a Widgets

1. **Create a widget object**
```
B=button(f, text="Submit")
```
2. **Define the event performed by the widget**
```
def buttonClick(self) :
        print("You have clicked submit button")
```
3. **Clicking event should be linked with the callback handler**
```
B.bind('<Button-1>', buttonClick)
```
4. **Call event loop**
```
root.mainloop()
```

1. **Button:** A push button is a component that performs some action when clicked. These buttons are created as objects of Button class as,
```
b=Button(f, text="Ok", width=15,height=3, bg="yellow", fg="blue")
```
Here, '**b**' is the object of Button class, '**f**' represents the frame for which the button is created as a child. The '**text**' option represents the text to be display on the button, **width** and **height** represents the size of the button, '**bg**' represents the background color and 'fg' represents the fore ground color of the button,
```
b.bind("<Button-1>", buttonClick)
```
We link the mouse left button with the buttonClick( )method using bind()method.

2. **Text:** A text widget is same as label. But text widget has several options and can display multiple lines of text in different colors and fonts. It is possible to insert text into a text widget, modify it or delete it. We can also display images in the text widget. A text is created as an object of Text class as,
```
t=Text(f,width=20, height=3, font=('Times',20,'italic'), fg='blue', bg="yellow",
wrap=WORD)
```
Here, '**t**' is the object of Text class, '**f**' represents the frame for which the text is created as a child. The **width** and **height** represents the size of the text box, '**bg**' represents the background color and 'fg' represents the fore ground color of the text, '**font** ' represents a tuple that contains font name, size and style. The '**wrap**' represents the text information can be align with in the text box.

3. **Label:** A label represents the constant text that is displayed in the frame or container. A label can display one or more lines of text that cannot be modified. A label is created as an object of Label class as,
```
l=Label(f, text=" Label Demo",width=25,height=3, font=('Times',14,'bold'),
fg='blue',bg='yellow')
```

Here, '**l'** is the object of Label class, '**f'** represents the frame for which the button is created as a child. The '**text'** option represents the text to be display on the Label box, **width** and **height** represents the size of the label, '**bg'** represents the background color and 'fg' represents the fore ground color of the label, '**font '** represents a tuple that contains font name, size and style.

4. **Check button:** Check buttons are also known as check boxes are useful for the user to select one or more options from available group of options. Check buttons are displayed in the form of square shaped boxes. When check box is selected, a tick mark is displayed on the check box. Check box is created as an object of Checkbutton class as,

   c1=Checkbutton(f, text="Python", bg='yellow', fg='red', font=('Times',20,'italic'))

   Here, '**c1'** is the object of Checkbutton class, '**f'** represents the frame for which the check button is created as a child. The '**text'** option represents the text to be display on the check box, '**bg'** represents the background color and 'fg' represents the fore ground color of the check box, '**font '** represents a tuple that contains font name, size and style.
5. **Radio button:** A radio button is similar to a check box, but it is used to select only one option from a group of available options. A radio button is displayed in the form of round shaped button. The user cannot select more than one option. When a radio button is selected, there appears a dot in the radio button. We can create a radio button as an object of the Radiobutton class as,

   r1=Radiobutton(f, text="male", bg='green', fg='red', font=('Times',20,'italic'), value=0)

   Here, '**r1'** is the object of Radiobutton class, '**f'** represents the frame for which the radio button is created as a child. The '**text'** option represents the text to be display on the radio button, '**bg'** represents the background color and 'fg' represents the fore ground color of the radio button, '**font '** represents a tuple that contains font name, size and style, '**value'** represents a value that is set to this object when the radio button is clicked..

WAP to illustrate button, label, text, checkbox and radio button

```
from tkinter import *
def buttonClick(self):
    print ("You have cliked Ok button")
  root=Tk()
root.title ("Button Example")
f=Frame(root,bg="blue",height=700,width=1200,cursor='cross')
f.propagate(0)
f.pack()
b=Button(f, text="Ok", width=15,height=3, bg="yellow", fg="blue")
b.pack()

b.bind("<Button-1>", buttonClick)

l=Label(f, text=" Label Demo",width=25,height=3,
font=('Times',14,'bold'), fg='blue',bg='yellow')
l.pack()
t=Text(f,width=20,
height=3,font=('Times',20,'italic'),fg='blue',bg="yellow",wrap=WORD)
t.insert(END," Text Demo")
```

```
t.pack(side=LEFT)

c1=Checkbutton(f, text="Python",bg='yellow',fg='red',
font=('Times',20,'italic'))
c1.pack(side=LEFT)
c2=Checkbutton(f, text="Networking",bg='yellow',fg='red',
font=('Times',20,'italic'))
c2.pack(side=LEFT)
c3=Checkbutton(f, text="Java",bg='yellow',fg='red',
font=('Times',20,'italic'))
c3.pack(side=LEFT)
r1=Radiobutton(f, text="male", bg='green',fg='red',
font=('Times',20,'italic'),value=0)
r1.pack(side=LEFT)
r2=Radiobutton(f, text="female", bg='green',fg='red',
font=('Times',20,'italic'),value=1)
r2.pack(side=LEFT)
root.mainloop()
```

WAP to draw BAR chart/graph

```
import matplotlib.pyplot as plt
left = [1, 2, 3, 4, 5]
height = [10, 24, 0, 4, 50]
tick_label = ['one', 'two', 'three', 'four', 'five']
plt.bar(left, height, tick_label = tick_label,

        width = 0.8, color = ['red', 'green'])
plt.xlabel('x - axis')
plt.ylabel('y - axis')
plt.title('My bar chart!')
plt.show()
```

WAP to draw PIE chart/graph

```
import matplotlib.pyplot as plt

activities = ['routine', 'sleep', 'work', 'play']

slices = [3, 7, 8, 6]
colors = ['r', 'y', 'g', 'b']
plt.pie(slices, labels = activities, colors=colors,
        startangle=90, shadow = True, explode = (0, 0.1, 0, 0),
        radius = 1.2, autopct = '%1.2f%%')
plt.legend()
plt.show()
```

--- Good Luck ---