PYTHON PROGRAMMING (PS01CMCA51)

Unit – IV

Additional Features of Python

- Object-oriented programming in Python
- File handling in Python
- Modules and packages
- Introduction to GUI applications and database connectivity

Object-oriented programming in Python

There are five important features related to Object Oriented Programming System :

- Classes and objects
- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

Object-oriented programming in Python

Class

– A blueprint/template that describes the behavior/state that the object of its type support.

– E.g. A class Car that has

- State (Attributes) :: Color, Brand, Weight, Model
- Behaviors (Methods) :: Break, Accelerate, Slow Down etc.

– This is just a blueprint, it does not represent any Car, however using this we can create Car objects (or instances) that represents the car.

Object

– It is an instance of class. When the objects are created, they inherit all the variables and methods from the class.

E.g. Audi, Toyota, Volvo, Maruti, Honda

Object-oriented programming in Python

Classes and Objects

An object is anything that really exists in the world and can be distinguished from others.

A **class** is a user **defined** blueprint or prototype from which objects are created.

Class : Person

Attributes: Name, Age, Gender

Actions: talking, walking, eating

Object : Raju

Variables: Raju,25,Male

Methos: talk(), walk(), eat()

Object-oriented programming in Python

Create Class

- A class is created with the **keyword class and** then writing the **Classname**.
- After the Classname, 'object' is written inside the Classname. This **'object' represents the base class name from where all classes in Python are derived.**
- We should mention 'object' in the parentheses. **Writing 'object' is an optional**.
- **E.g.**

```
class Student:
```

OR

```
class Student(object):
```

Object-oriented programming in Python

- We write a class with the **attributes and actions** of objects.
- **Attributes** are represented by **variables** and **actions** are performed by **methods**. So, **a class contains variable and methods.**

```
class Student:
    def sayHi(self):
        print ('Hello, how are you?')
    def sayBye(self):
        print ('Bye, Bye')
```

Object-oriented programming in Python

Create an object

- To create an instance (or object) the syntax is :

  instancename = Classname()

E.g. create an instance (or object) of the Student class, we can write as:

s1 = Student()

s2 = Student()

s3 = Student()

Object-oriented programming in Python

Create a student class and its object to access its methods.

```
class Student:
    def sayHi(self):
        print ('Hello, how are you?')
    def sayBye(self):
        print ('Bye, Bye')


s = Student()
s.sayHi();  s.sayBye();
```

Object-oriented programming in Python

Create a student class & multiple objects

```
class Student:
    def sayHi(self):
        print ('Hello, how are you?')
    def sayBye(self):
        print ('Bye, Bye')
s1 = Student() # First Object
s1.sayHi(); s1.sayBye();

s2 = Student() # Second object
s2.sayHi(); s2.sayBye();
```

Object-oriented programming in Python

self parameter

- **Class methods have a special parameter** at the beginning of the parameter list. It refers the object itself and name is **self.**
- **We do not give value for this parameter (self) when we call the method.**
- For example,
  - if we have a **class** called **MyClass** and
  - an instance (**object**) of this class called **MyObject**,
  - when we call a method of this object as

    MyObject.method(arg1, arg2)
  - It is automatically converted to

    MyClass.method(MyObject, arg1, arg2).
- This is what the special **self** is all about.

Object-oriented programming in Python

Constructor

- A constructor is a special method that is used **to initialize the instance variables of a class**.
- The **first parameter** of the constructor will be **'self'** variable that **contains the memory address of the instance.**
- E.g.

```
def __init__(self):
        self.pi = 3.14
def __init__(self):
        self.city = 'Anand'
        self.weight= 55
```

- A constructor is called at the time of creating an instance.
- So, the above **constructor will be called when we create an instance as: s1 = Student()**

Object-oriented programming in Python

Constructor

- **__init__()** method is useful **to initialize the variables.** Hence, the name 'init'.
- The method name has **two underscores** before and after. This indicates that this method is internally defined and we cannot call this method explicitly.
- **'self' is a variable that refers to current class instance**. When we create an instance for the Student, a separate memory block is allocated on the heap and that **memory location** is by default **stored in 'self'.**
- The instance (or object) contains the variables 'name', 'weight' which are called instance variables. To refer to instance variables, we can use the **dot operator** notation along with self as: '**self.name**' , '**self.age**' and '**self.marks**'

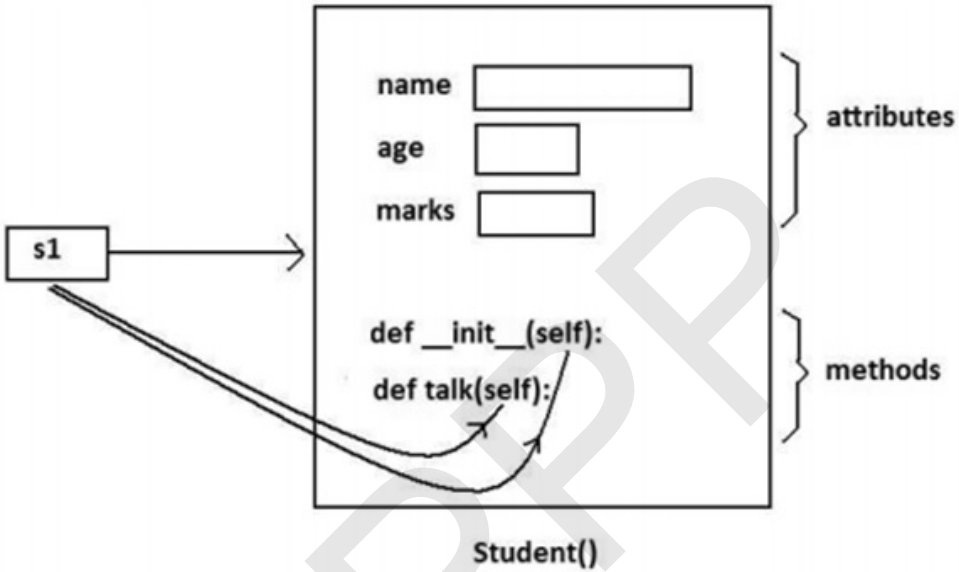
Object-oriented programming in Python

When we create an instance following steps will take place internally:

- **First** of all, a block of memory is allocated on heap. How much memory is to be allocated is decided from the attributes and methods available in the **Student class.**
- **Second , the special method by the name '__init__(self)' is called internally.** This method stores the initial data into the variables.
- **Finally**, the allocated memory location address of the instance is returned into object variable. To see this memory location in decimal number format, we can use id() function as id(object-name).

Object-oriented programming in Python

E.g. Student class with three **variables** name, age and marks. Also one **constructor** __init___ and one **method** talk() is as below.



- Now, 's1' refers to the instance of the Student class.
- Hence any variables or methods in the instance can be referenced by 's1' using **dot operator**
- s1.name #this refers to data in name variable, i.e. Vishnu
  s1.age #this refers to data in age variable, i.e. 20

  s1.marks #this refers to data in marks variable, i.e. 900

  s1.talk() #this calls the talk() method

Object-oriented programming in Python

A student class having name & age as a variable and **one constructor and one method** get_data to display data.

```
class Student:
    def __init__(self,name,age):
        self.name = name
        self.age = age
    def getData(self):
        print('Hi', self.name)
        print('Your age:', self.age)
        print('------------------')
s1 = Student('Ajay',22);          s1.getData();
s2 = Student('Shruti',21);        s2.getData();
```

Object-oriented programming in Python

A student class having name & age as a variable and **two methods set_data** to take inputs and **get_data** to display data.

```
class Student:
    def setData(self):
        self.name = input('Enter name: ')
        self.age = input('Enter age: ')
    def getData(self):
        print('Hi', self.name)
        print('Your age:', self.age)
        print('--------------------')
s = Student()
s.setData()
s.getData()
```

Object-oriented programming in Python

Program to create a student class having two methods set_data to take inputs from the user and get_data to display data for N students.

```
class Student:
    def setData(self):
        self.name = input('Enter name: ')
        self.age = input('Enter age: ')
    def getData(self):
        print('Hi', self.name)
        print('Your age:', self.age)
n = int(input('How many students?'))
i=0
while( i < n ):
   s = Student();
   s.setData();   s.getData();
   i = i + 1
```

Object-oriented programming in Python

Program to create a student class to accept and display data of N students.

```
class Student:
    def setName(self, name):
        self.name = name
    def getName(self):
        return self.name
    def setAge(self, age):
        self.age = age
    def getAge(self):
        return self.age
```

Object-oriented programming in Python

```
n = int(input('How many students?'))
i=0
while(i<n):
    s = Student()
    name = input('Enter name: ')
    s.setName(name)
    age = input('Enter age: ')
    s.setAge(age)
    print('Hi', s.getName())
    print('Your Age:', s.getAge())
    i = i + 1
```

Object-oriented programming in Python

- **Inheritance** allows us to define a class that inherits all the methods and properties from another class.
- **Parent class** is the class being inherited from, also called base class.
- **Child class** is the class that inherits from another class, also called derived class.

Object-oriented programming in Python

Create a Parent Class

syntax is the same as creating any other class:

Create a class named Person, with firstname and lastname properties, and a printname method:

```
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname
  def printname(self):
    print(self.firstname, self.lastname)
# create an object, and then execute the printname method:
x = Person("Kartik", "Dave")
x.printname()
```

Object-oriented programming in Python

- **Create a Child Class**
- To create a class that inherits the functionality from another class, send the parent class as a parameter when creating the child class: **Example**
- Create a class named Student, which will inherit the properties and methods from the Person class:

```
class Student(Person):
  pass
```

- Use the **pass keyword** when you do not want to add any other properties or methods to the class.
- Use the Student class to create an object, and then execute the printname method:

```
x = Student("Mitesh", "Trivedi")
x.printname()
```

Object-oriented programming in Python

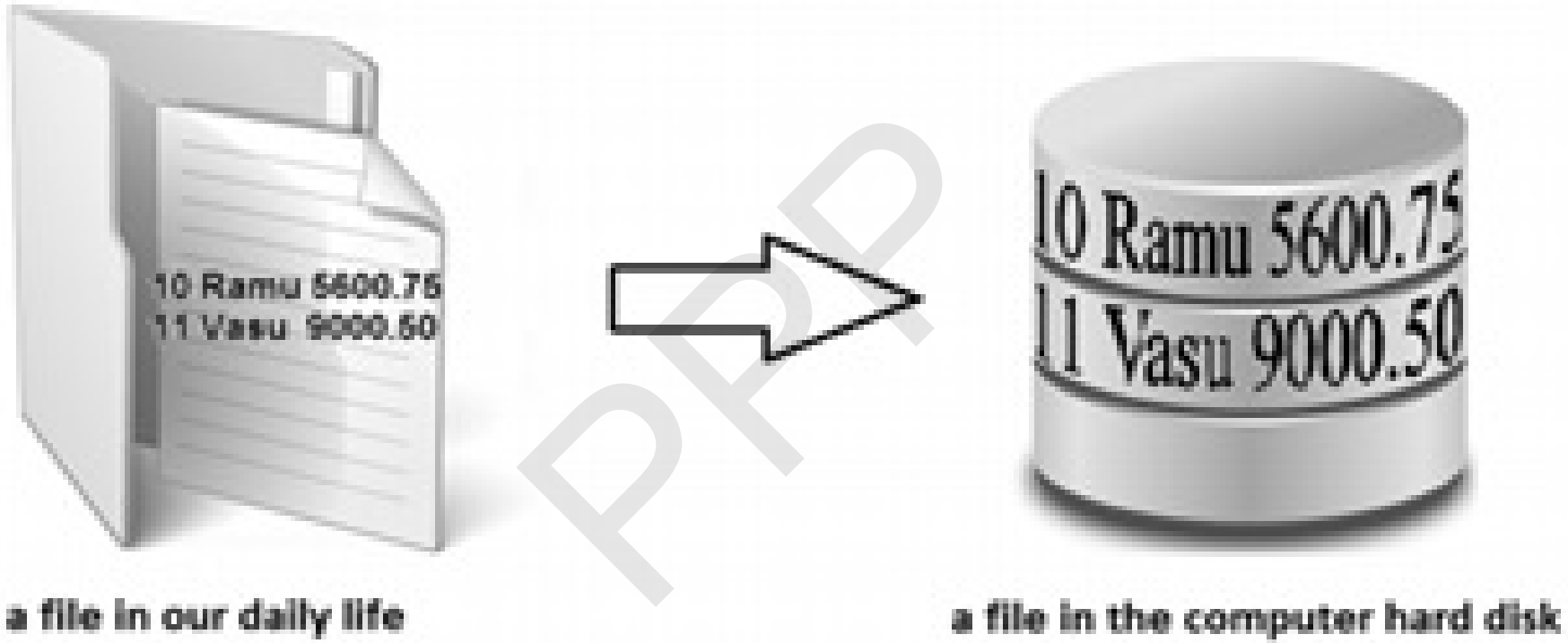
```
class Person: # base class or super class
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname
  def printname(self):
    print(self.firstname, self.lastname)


class Student(Person):  # inherited class / Child class
            pass
# create an object of child class
x = Student("Mitesh", "Trivedi")
x.printname()
```

File handling in Python

- Data is very important. Every organization depends on its data for continuing its business operations. If the data is lost, the organization has to be closed.
- This is the reason computers are primarily created for handling data, especially for storing and retrieving data.
- To store data in a computer, we need files.
- For example,
  - we can store employee data like employee number, name and salary in a file in the computer & later use it whenever we want.
  - we can store student data like student roll number, name and marks in the computer.
- A file is the collection of data that is available to a program. Once we store data in a computer file, we can retrieve it and use it depending on our requirements.

File handling in Python



a file in our daily life

a file in the computer hard disk

File handling in Python

Advantages of storing data in a file:

- When the data is stored in a file, **it is stored permanently.** This means that even though the computer is switched off, the data is not removed from the memory since the file is stored on hard disk or CD. This file data can be utilized later, whenever required.
- **It is possible to update the file data.** For example, we can add new data to the existing file, delete unnecessary data from the file and modify the available data of the file.
- **Once the data is stored in a file, the same data can be shared by various programs.** For example, once employee data is stored in a file, it can be used in a program to calculate employee's net salaries or in another program to calculate income tax payable by the employees.

File handling in Python

Two Types of Files

- Text files
- Binary files

- **Text files** store the data in the form of characters.
  - For example, if we store employee name "Ganesh", it will be stored as 6 characters and the employee salary 8900.75 is stored as 7 characters. Normally, text files are used to store characters or strings.
- **Binary files** store entire data in the form of bytes, i.e. a group of 8 bits each.
  - Image files are generally available in .jpg, .gif or .png formats. Images contain pixels which are minute dots with which the picture is composed of. Each pixel can be represented by a bit, i.e. either 1 or 0. Since these bits can be handled by binary files, we can say that they are highly suitable to store images.

File handling in Python

Opening File

- We should use **open() function** to open a file. This function accepts 'filename' and 'open mode' in which to open the file.

- **file handler = open("file name", "open mode")**
  - 'file name' represents a name of file.
  - 'open mode' represents the purpose of opening the file.

File handling in Python

File open mode	Description
w	**To write data into file.** If any data is already present in the file, it would be overwrite in the file.
r	**To read data from the file.** The file pointer is positioned at the beginning of the file.
a	**To append data to the file.** Appending means adding at the end of existing data. The file pointer is placed at the end of the file. If the file does not exist, it will create a new file.
w+	**To write and read data of a file.** The **previous data** in the file will be **deleted**.
r+	**To read and write data into a file.** The **previous data** in the file will **not be deleted**.
a+	**To append and read data of a file.** The file pointer will be at the end of the file if the file exists. If the file does not exist, it creates a new file for reading and writing.

File handling in Python

Closing Files

- A file which is opened should be closed using the **close() method.**

- If the file is not closed, the memory utilized by the file is not freed, leading to problems like insufficient memory. Hence it is mandatory to close the file.

- **file-handler-name.close()**

File handling in Python

Writing into a file

- we are creating a file where we want to store some characters. we store the string into the file using write() method as:
- **f.write(str)**

Reading from file

- To read data from a text file, we can use read() method as:

  str = f.read()

  - This will read all characters from the file 'f' and returns them into the string 'str'.
- To read only a specified number of bytes from the file:

  str = f.read(n)

  - where 'n' represents the number of bytes to be read from the beginning of the file.

File handling in Python

- **A Python program to create a text file to store individual characters.**

```
#open the file for writing data
f = open('myfile.txt', 'w')

#Input characters from user
str = input('Enter text:')
f.write(str)

#close the file
f.close()
```

File handling in Python

A Python program to read characters from a text file.

```
#open the file for reading data
f = open('myfile.txt', 'r')

#read all characters from file
str = f.read()

#display them on the screen
print(str)

#closing the file
f.close()
```

File handling in Python

A Python program to create a text file to store names and retrieve it.

```
f = open('myfile.txt', 'w')
print('Enter Names ( # at end):')
while str != '#':
   str = input()
   if(str != '#'):
     f.write(str+"\n")
f.close()


f = open('myfile.txt', 'r')
print('The file contents are:')
str = f.read()
print(str)
f.close()
```

File handling in Python

A program to append data to an existing file and then displaying the entire file. Same as previous program only mode is change to 'a'

```
f = open('myfile.txt', 'a')
print('Enter Names ( # at end):')
while str != '#':
   str = input()
   if(str != '#'):
     f.write(str+"\n")
f.close()

f = open('myfile.txt', 'r')
print('The file contents are:')
str = f.read()
print(str)
f.close()
```

File handling in Python

File exists or not

The operating system **(os) module** has a **sub module by the name 'path'** that contains a **method isfile()**.

This method can be used to know whether a file that we are opening really exists or not.

For example, **os.path.isfile(fname)** gives True if the file exists otherwise False.

File handling in Python

File exists or not

```
import os, sys
fname = input('Enter filename:')
if os.path.isfile(fname):
   f = open(fname, 'r')
else:
   print(fname+' does not exist')
   sys.exit()

print('The file contents are:')
str = f.read()
print(str)
f.close()
```

File handling in Python

A Python program to count number of lines, words and characters in a text file.

```
f = open('myfile.txt', 'r')
cl= cw= cc = 0
for line in f:              # To read one by one line from text file.
   cl +=1
   words = line.split()     #words is a list datatype
   cw += len(words)         # len method gives no. of elements in list
   cc += len(line)
print('No. of lines:', cl)
print('No. of words:', cw)
print('No. of characters:', cc)
f.close()
```

File handling in Python

Take N numbers from user & store in file. Also find the sum of all nos.

```
f = open('myfile.txt', 'w')
no=int(input('Total numbers you wants to enter::'))
for i in range(1,no+1):
    print('Enter the value of number [ ' , i ,' ] ::-->' )
    no = int(input())
    f.write(str(no))
    f.write('\n')
f.close()
f = open('myfile.txt', 'r')
ans = 0
for no in f:
    no = int(no)
    ans = ans + no
print('sum of all numbers ::',ans)
f.close()
```

File handling in Python

Binary Files

- **Binary files** handle data in the form of bytes. Hence, they can be used to read or write text, images or audio and video files.
- To open a binary file **for reading purpose**, we can use **'rb' mode.** Here, 'b' is attached to 'r' to represent that it is a binary file. Similarly to **write bytes** into a binary file, we can use **'wb' mode.**
- To **read bytes** from a binary file, we can use the **read() method** and to **write bytes** into a binary file, we can use the **write() methods.**

File handling in Python

- A program to open an image file like .jpg, .gif or .png and copying it as another file.

```
#Open the files in binary mode
f1 = open('first.jpg', 'rb')
f2 = open('new.jpg', 'wb')

#read bytes from f1 and write into f2
bytes = f1.read()
f2.write(bytes)

#close the files
f1.close()
f2.close()
```

Modules and packages

- Grouping related code into a module makes the code easier to understand and use. It also makes the code logically organized. A Python module is a file containing Python definitions and statements. A module can define functions, classes, and variables. A module can also include runnable code.
- A built-in module may be a Python script (with .py extension) containing useful utilities.
- To display list of all available modules, use following command in Python console:
- **>>> help("modules")**

Modules and packages

Most frequently used Built-in modules as below.

- **os module :** It has functions to perform many tasks of operating system.
- **random module :** It has functions for handling randomization.
- **math module :** It has functions for mathematical operations.
- **time module:** It has time related functions.
- **sys module :** It has functions to manipulate different parts of the Python runtime environment.
- **collections module:** provides alternatives to built-in container data types such as list, tuple and dict.
- **statistics module :** It provides statistical functions.

Modules and packages

Import Module– Import statement

- We can import the functions, classes defined in a module to another module using the import statement in some other Python source file.
- **Syntax:**

  import module-name

- When the interpreter encounters an import statement, it imports the module if the module is present in the search path. A search path is a list of directories that the interpreter searches for importing a module.
- It does not import the functions or classes directly instead imports the module only. **To access the functions inside the module the dot(.) operator is used.**

Modules and packages

python built-in modules:

```
# importing built-in module math
import math


# using square root(sqrt) function contained in math module
print(math.sqrt(25))


# using pi function contained in math module
print(math.pi)


# 1 * 2 * 3 * 4 = 24
print(math.factorial(4))
```

Modules and packages

 # importing built in module random

import random

# printing random integer between 0 and 5
print(**random.randint(0, 5)**)

# print random floating point number between 0 and 1
print(**random.random()**)

# random number between 0 and 100
print(**random.random() * 100**)

Modules and packages

The dir() function

- The dir() built-in function returns a sorted list of strings containing the names defined by a module.
- The list contains the names of **all the modules, variables, and functions** that are defined in a module.

E.g.

>>> import math

>>> print(dir(math))

['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', **'ceil'**, 'comb', 'copysign', 'cos', 'cosh', 'degrees', 'dist', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', **'factorial', 'floor',** 'fmod', 'frexp', 'fsum', 'gamma', **'gcd',** 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'isqrt', **'lcm'**, 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'nextafter', 'perm', **'pi', 'pow', 'prod', 'radians', 'remainder', 'sin',** 'sinh', **'sqrt', 'tan',** 'tanh', 'tau', 'trunc', 'ulp']

Modules and packages

>>> import random

>>> print(dir(random))

['BPF', 'LOG4', 'NV_MAGICCONST', 'RECIP_BPF', 'Random', 'SG_MAGICCONST', 'SystemRandom', 'TWOPI', '_Sequence', '_Set', '__all__', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', '_accumulate', '_acos', '_bisect', '_ceil', '_cos', '_e', '_exp', '_floor', '_inst', '_log', '_os', '_pi', '_random', '_repeat', '_sha512', '_sin', '_sqrt', '_test', '_test_generator', '_urandom', '_warn', 'betavariate', 'choice', 'choices', 'expovariate', 'gammavariate', 'gauss', 'getrandbits', 'getstate', 'lognormvariate', 'normalvariate', 'paretovariate', 'randbytes', 'randint', 'random', 'randrange', 'sample', 'seed', 'setstate', 'shuffle', 'triangular', 'uniform', 'vonmisesvariate', 'weibullvariate']

Modules and packages

statistics module

- This module provides following statistical functions :
- **mean() :** calculate arithmetic mean of numbers in a list
- **median()** : returns middle value of numeric data in a list. For odd items in list, it returns value at (n+1)/2 position. For even values, average of values at n/2 and (n/2)+1 positions is returned.
- **mode():** returns most repeated data point in the list.

```
import statistics
print(statistics.mean([10,20,30,40,50]))
print(statistics.median([10,20,30,40,50]))   # odd items
print(statistics.median([10,20,30,40,50,60])) # even items
print(statistics.mode([10,20,30,20,10,20]))
```

Modules and packages

```
# A simple module, calc.py
def add(x, y):
    return (x+y)
def subtract(x, y):
    return (x-y)
```

Example: Importing modules in Python

```
# importing  module calc.py
import calc
print(calc.add(10, 2))
print(calc.subtract(10, 2))
```

Modules and packages

Packages

- A **module** may contain several classes, functions, variables, etc. whereas a Python **package** contains several module. In simpler terms a package is folder that contains various modules as files.

Creating Package

- A **package** named **mypckg** that will contain **two modules mod1 and mod2.** To create this module follow the below steps –
  - **Create a folder named mypckg.**
  - Inside this folder **create an empty Python file i.e. __init__.py**
  - Then **create two modules mod1 and mod2 in this folder.**

Modules and packages

mod1.py

========

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

mod2.py

========

```
def subtract(x, y):
    return (x-y)
```

Modules and packages

The hierarchy of the our package looks like this –

mypckg

|

|

---__init__.py

|

|

---mod1.py

|

|

---mod2.py

Modules and packages

Understanding __init__.py

- __init__.py helps the Python interpreter to recognize the folder as package.
- It also specifies the resources to be imported from the modules. **If the __init__.py is empty this means that all the functions of the modules will be imported.**
- We can also specify the functions from each module to be made available.

  E.g. we can also create the **__init__.py file for the above module as –**

  from .mod1 import add

  from .mod2 import subtract
- This __init__.py will only allow the add and subtract functions from the mod1 and mod2 modules to be imported.

Modules and packages

Import Modules from a Package

we use from...import statement and the dot(.) operator.

Syntax: **import package_name.module_name**

E.g.

[1] **import mypckg.mod1**

ans = **mypckg.mod1.**add(10,20)

print(ans)

[2]

from mypckg import mod1

from mypckg import mod2

print('Addition of two numbers::',**mod1.add**(20,10))

res = **mod2.subtract**(20,10)

print('Subtraction of two numbers ::', res)

Introduction to GUI applications and database connectivity

- To interact with an application through graphics, pictures or images.
- User need not remember any commands. He/She can perform the task just by clicking on relevant images.
- E.g. To send data to printer, the user will simply click on the Printer image (or picture). Then he has to tell how many copies he wants and the printing will continue.
- This environment where the user can interact with an application through graphics or images is called GUI (Graphical User Interface). One example for GUI is **Windows operating system**.

Introduction to GUI applications and database connectivity

GUI Advantages:

- **It is user-friendly.** The user need not worry about any commands. Even a layman will be able to work with the application developed using GUI.
- **It adds attraction and beauty to any application** by adding pictures, colors, menus, animation, etc. For example, all websites on Internet are developed using GUI to lure their visitors and improve their business.
- **It is possible to simulate the real life objects using GUI.** For example, a calculator program may actually display a real calculator on the screen. The user feels that he is interacting with a real calculator without any training. **It eliminates need of user training.**
- GUI helps to create graphical components like push buttons, radio buttons, check buttons, menus, etc. and use them effectively.

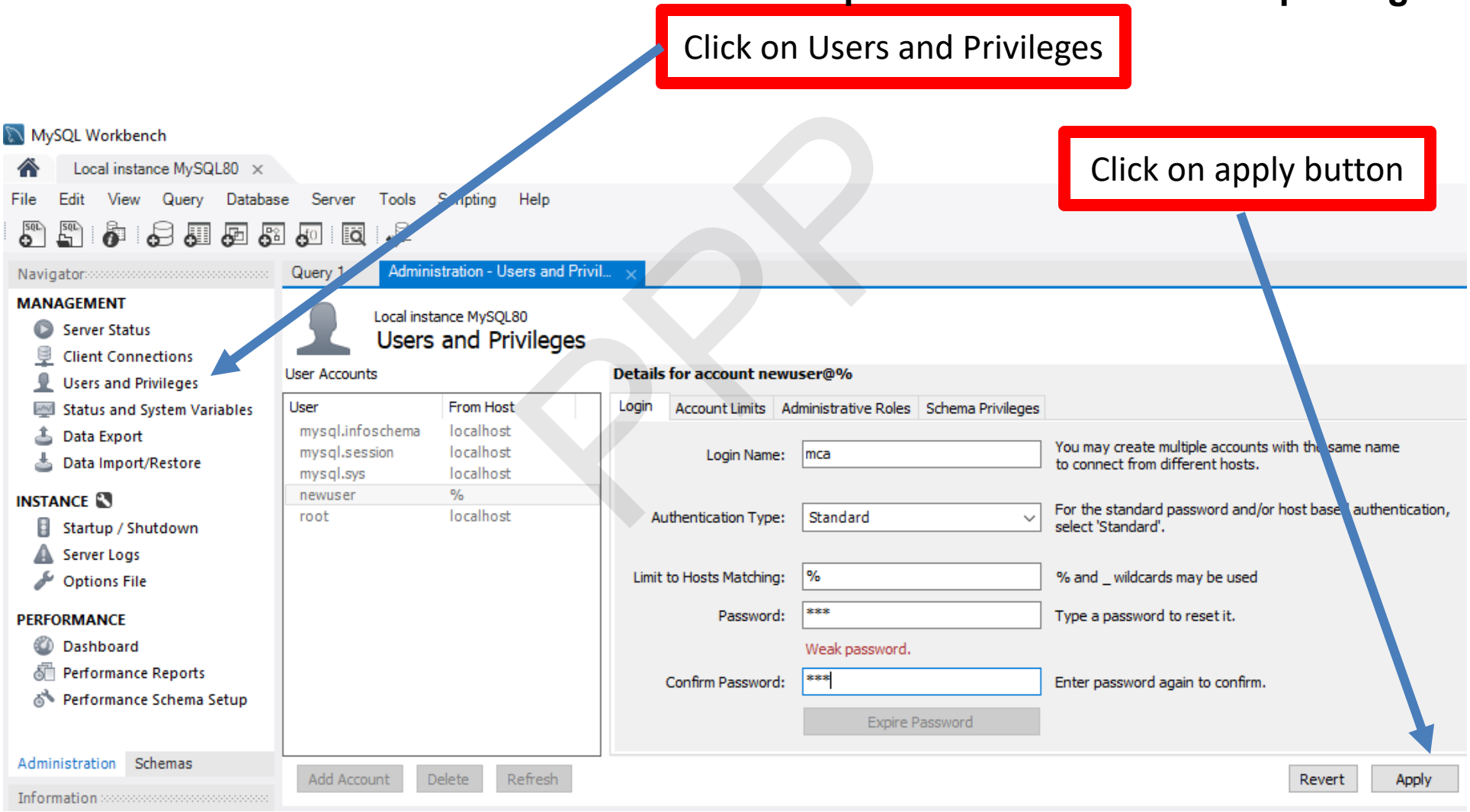
Introduction to GUI applications and database connectivity

- Download & Install **MySQL Workbench 8.0.27 from below website .** https://dev.mysql.com/downloads/workbench/

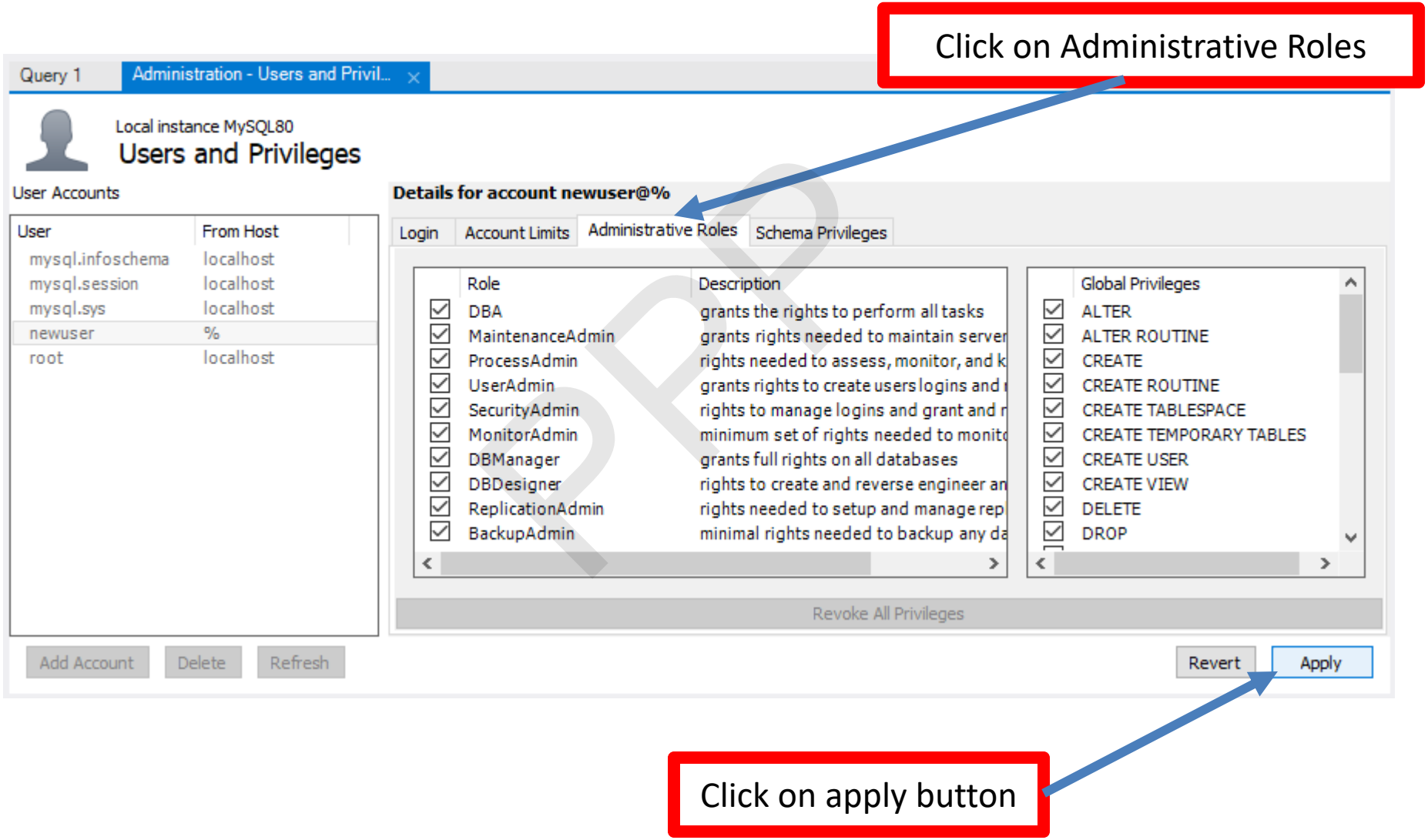
- **Launch mysql workbench with default username and password is "root".**

Introduction to GUI applications and database connectivity

- **CREATE A NEW USER with name mca and password mca with DBA privileges**



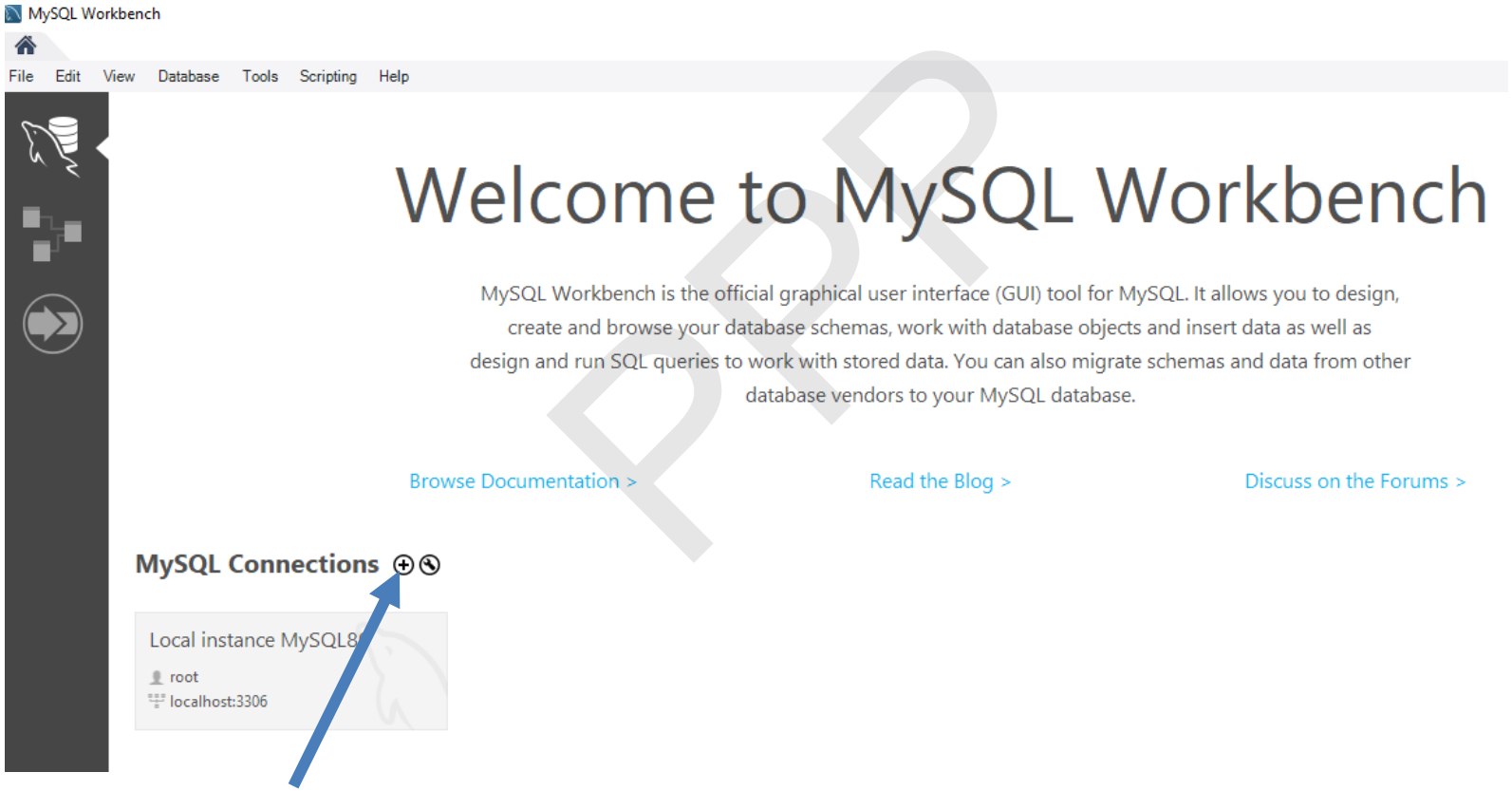
Introduction to GUI applications and database connectivity



Introduction to GUI applications and database connectivity

Creating A New MySQL Connection

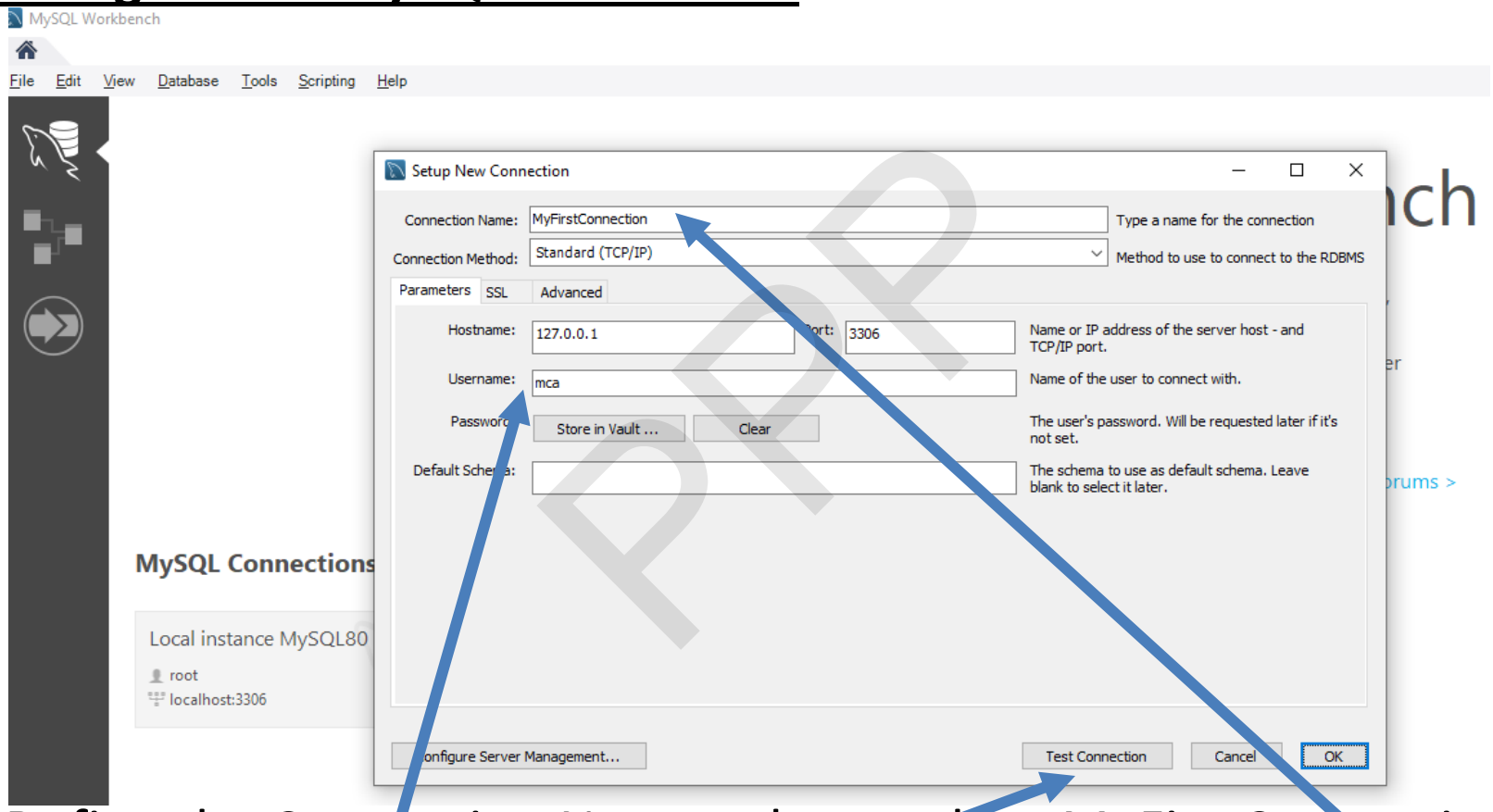
- Launch MySQL Workbench to open the home screen.



- click the [+] icon near the MySQL Connections label to open the Setup New Connection wizard.

Introduction to GUI applications and database connectivity

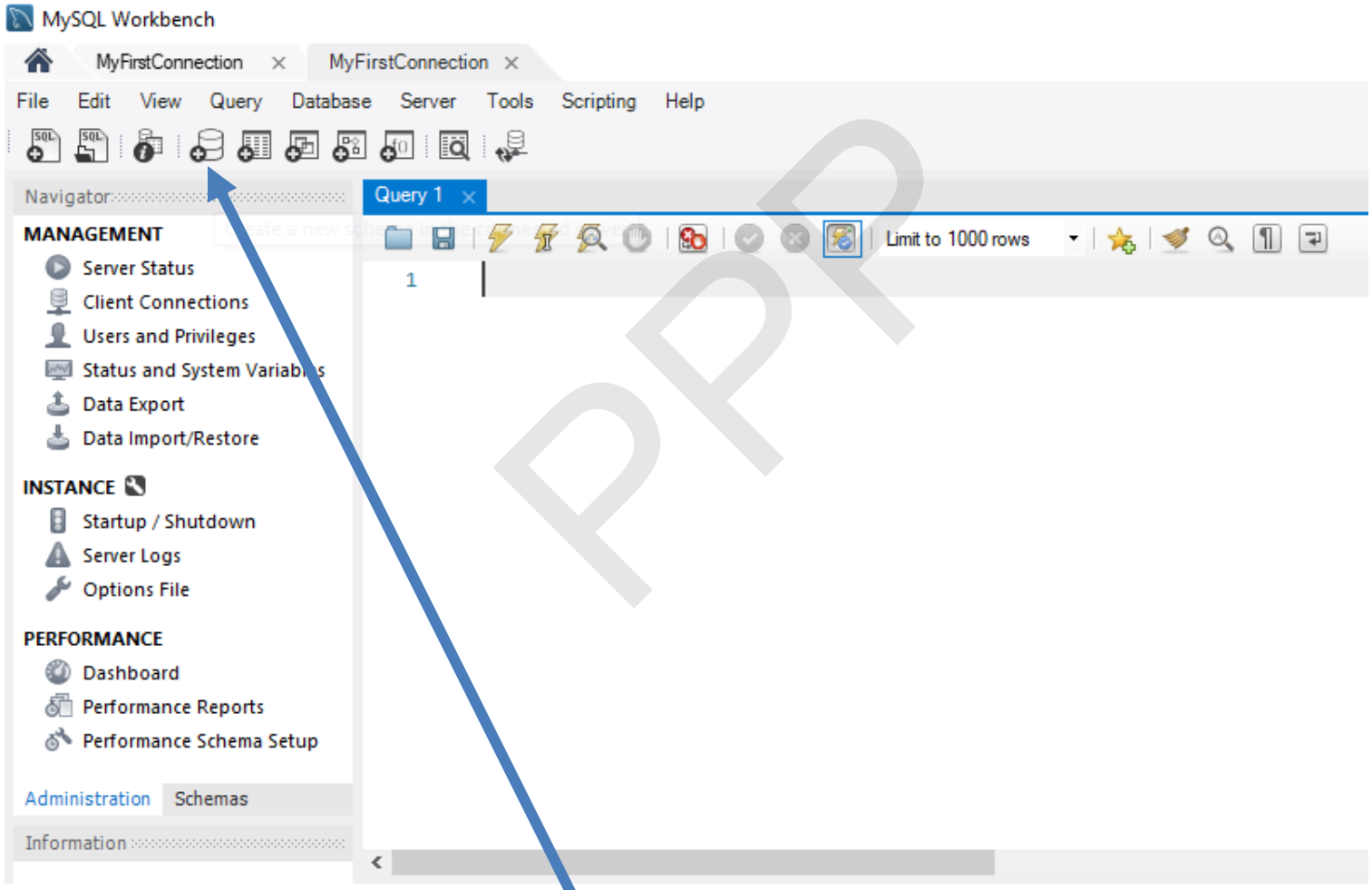
Creating A New MySQL Connection



- Define the Connection Name value, such as MyFirstConnection
- Username = mca
- Test connection.

Introduction to GUI applications and database connectivity

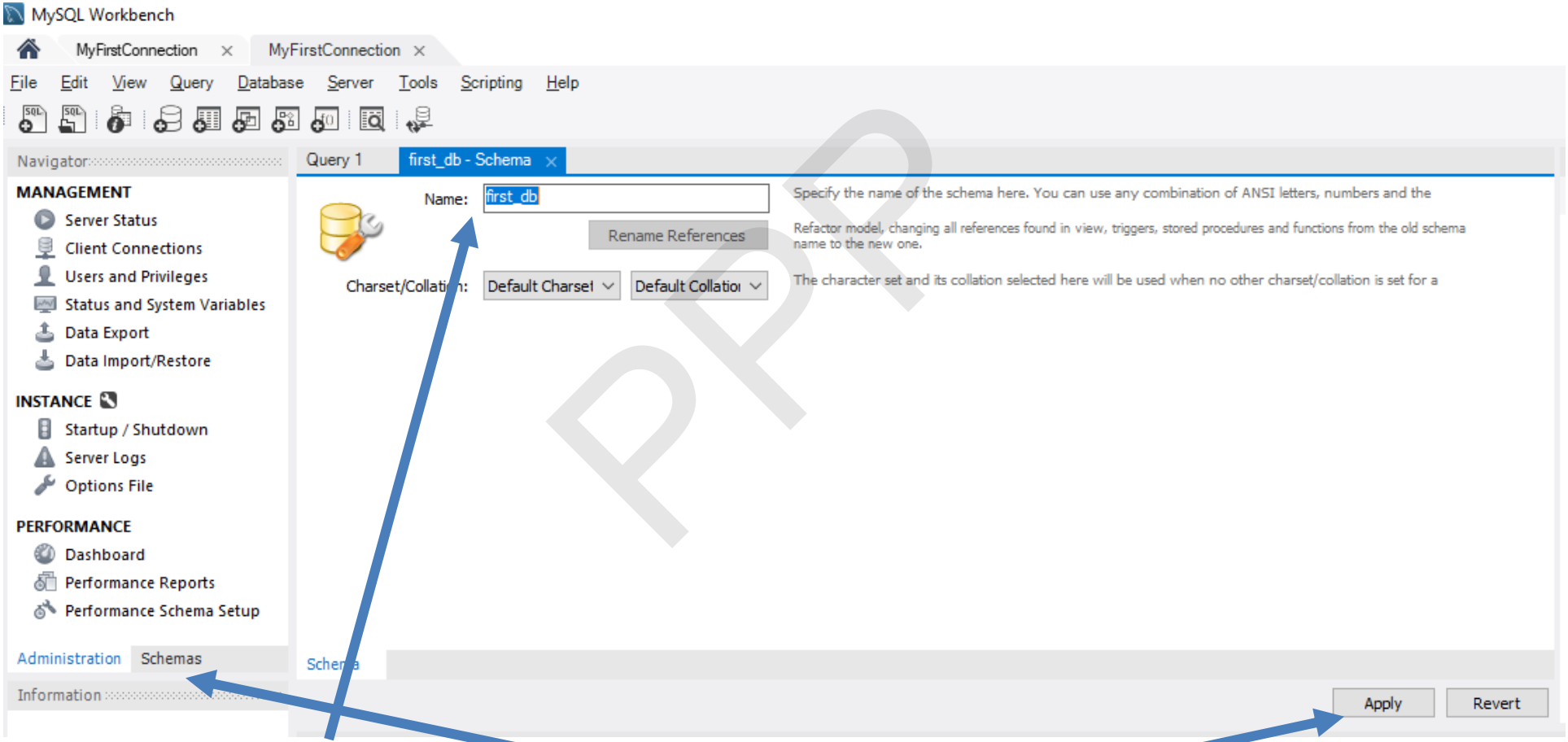
Creating A New Database/Schema



Click on the create schema icon in above figure.

Introduction to GUI applications and database connectivity

Creating A New Database/Schema



- Name = first_db and click on apply button.
- It will be seen by clicking on Schema tab option in above figure.

Introduction to GUI applications and database connectivity

- **Open mysql workbench with username and password is "mca".**
- There are the following steps to connect a python application to database.
  1. Import mysql.connector module
  2. Create the connection object.
  3. Create the cursor object
  4. Execute the query

Introduction to GUI applications and database connectivity

1. Import mysql.connector module

- Python needs a **MySQL driver to access the MySQL database.** The driver "**MySQL Connector**".
- use **PIP to install "MySQL Connector".**
- **To Download and install** "MySQL Connector":
- Open the windows command prompt with cmd command.
- Move to the specific directory of python.
- C:\Users*Your Name*\AppData\Local\Programs\Python\Python36-32\Scripts>**python -m pip install mysql-connector-python**
- **Test MySQL Connector**
  - **import mysql.connector**
  - If the above code was executed with no errors, "MySQL Connector" is installed and ready to be used.

Introduction to GUI applications and database connectivity

2. Creating the connection

- To create a connection between the MySQL database and the python application, the **connect() method of mysql.connector module** is used.
- Pass the database details like **HostName, username, and the database password in the method call**. The method returns the connection object.
- **syntax**

Connection-object = mysql.connector.connect (host=<hostname> , user = <username> , passwd = <password>)

Introduction to GUI applications and database connectivity

E.g.

```
import mysql.connector

#Create the connection object
myconn = mysql.connector.connect (
host="localhost",user="mca",passwd="mca",database="first_db")

#printing the connection object
print(myconn)
```

Introduction to GUI applications and database connectivity

3. Creating a cursor object

- The cursor object can be defined as an abstraction specified in the Python DB-API 2.0.
- It facilitates us to have multiple separate working environments through the same connection to the database.
- We can create the cursor object by calling the 'cursor' function of the connection object.
- The cursor object is an important aspect of executing queries to the databases.

syntax **<my_cur> = conn.cursor()**

Introduction to GUI applications and database connectivity

E.g.

```
import mysql.connector

#Create the connection object
myconn = mysql.connector.connect (
host="localhost",user="mca",passwd="mca",database="first_db")

#printing the connection object
print(myconn)

#creating the cursor object
cur=myconn.cursor()
print(cur)
```

Introduction to GUI applications and database connectivity

4. Execute the query

- This method executes the given database operation (query or command).
- Executing cursor with execute method and pass SQL query.

Introduction to GUI applications and database connectivity

E.g. Program to create a student table in first_db database.

```
import mysql.connector
#Create the connection object
myconn = mysql.connector.connect (
host="localhost",user="mca",passwd="mca",database="first_db")
#creating the cursor object
cur=myconn.cursor()
#Creating a table
cur.execute("CREATE TABLE student (name VARCHAR(255), address VARCHAR(255))")
print('student table is successfully created')
```

Introduction to GUI applications and database connectivity

E.g. Program to create a customer table in first_db database and apply insert , update, select and delete statement.

```
import mysql.connector
#Create the connection object
myconn = mysql.connector.connect(host = "127.0.0.1", user = "mca",password = "mca")
#printing the connection object
print(myconn)
mycursor = myconn.cursor()
#Create database with name firstdb if not exist
#mycursor.execute("CREATE DATABASE first_db")
mycursor.execute("CREATE DATABASE IF NOT EXISTS first_db")
mycursor.execute("SHOW DATABASES")
for x in mycursor:
  print(x)
```

Introduction to GUI applications and database connectivity

#create connection with database name as "first_db"

myconn = mysql.connector.connect(host = "127.0.0.1", user = "mca",password = "mca",database="first_db")

mycursor = myconn.cursor()

mycursor.**execute**("CREATE TABLE customers (id INT AUTO_INCREMENT PRIMARY KEY, name VARCHAR(255), address VARCHAR(255))")

sql = "INSERT INTO customers (name, address) VALUES (%s, %s)"

val = ("Amit", "Anand")

mycursor.**execute(sql, val)**

myconn**.commit()**

print(mycursor.**rowcount**, "record inserted.")

Introduction to GUI applications and database connectivity

```
val = ("Krina", "Surat")
mycursor.execute(sql, val)
myconn.commit()
print(mycursor.rowcount, "record inserted.")
val = ("Ramesh", "Baroda")
mycursor.execute(sql, val)
myconn.commit()
print(mycursor.rowcount, "record inserted.")
mycursor.execute("SELECT * FROM customers")
myresult = mycursor.fetchall()
for x in myresult:
  print(x)
```

Introduction to GUI applications and database connectivity

```
sql = "UPDATE customers SET address = 'Ahmedabad' WHERE address = 'Anand'"
mycursor.execute(sql)
myconn.commit()
print(mycursor.rowcount, "record(s) affected")

sql = "DELETE FROM customers WHERE address = 'Rajkot'"
mycursor.execute(sql)
myconn.commit()
print(mycursor.rowcount, "record(s) deleted")
```

Introduction to GUI applications

Tkinter: It is Python's standard GUI (graphical user interface) package. It is the most commonly used toolkit for GUI programming in Python.

Tk provides the following widgets:

button	canvas	combo-box	frame
check-button	message	radio button	Menu
list – box			

Introduction to GUI applications

Creating a GUI program using this Tkinter is simple. For this, programmers need to follow the steps mentioned below:

- Import the module Tkinter
- Build a GUI application (as a window)
- Add widgets
- Enter the primary, i.e., the main event's loop for taking action when the user triggered the event.

To display the graphical output, we need space on the screen. This space that is initially allocated to every GUI program is called **'top level window' or 'root window'.** We can reach this root window by creating an object to Tk class.

Introduction to GUI applications

A Python program to create root window or top level window.

```
from tkinter import *
#create top level window
root = Tk()
#set window title
root.title("First GUI Application")
#set window size
root.geometry("600x300")
#set window icon
root.wm_iconbitmap('first.ico')
#display window and wait for any events
root.mainloop()
```

Introduction to GUI applications

Python Containers

- A container is a component that is used as a place where drawings or widgets can be displayed. There are **two important containers:**
- **Canvas**: This is a container that is generally used to draw shapes like lines, curves, arcs and circles.
- **Frame**: This is a container that is generally used to display widgets like buttons, check buttons or menus.
- After creating the root window, we have to create space, i.e. the container in the root window so that we can use this space for displaying any drawings or widgets.

Introduction to GUI applications

Python Containers: Canvas

- A canvas is a rectangular area which can be used for drawing pictures like lines, circles, polygons, arcs, etc. To create a canvas, create an object to Canvas class as:
- **c = Canvas(root, bg="blue", height=500, width=600, cursor='pencil')**
- Here, **'c'** is the Canvas class object. **'root'** is the name of the parent window. **'bg'** represents background color, **'height' and 'width'** represent the height and width of the canvas **in pixels**. The option **'cursor'** represents the shape of the cursor in the canvas. Important cursor shapes are: arrow, box_spiral, center_ptr, circle, clock, pencil etc.

Introduction to GUI applications

Python Containers: Canvas

- Once the canvas is created, it should be added to the root window. Then only it will be visible. This is done using the **pack() method**, as : **c.pack()**
- After the canvas is created, we can draw any shapes on the canvas. For example, **to create a line**, we can use **create_line() method**, as:
- **id = c.create_line(50, 50, 200, 50, 200, 150, width=4, fill="white")**
- This creates a line with the connecting **points (50, 50), (200, 50) & (200, 150). 'width'** specifies the width of the line. The default width is 1 pixel. **'fill'** specifies the color of the line. The create_line() method returns an identification number.

Introduction to GUI applications

Python Containers: Canvas

- Similarly, to **create a rectangle or square shaped box**, we can use the create_rectangle() method as:
- **id = c.create_rectangle(500, 200, 700, 600, width=2, fill="gray", outline="black", activefill="yellow")**
- Here, the rectangle will be formed with the **top left coordinate** at (500, 200) pixels and the **lower bottom coordinate** at (700, 600). The **color** of the rectangle will change from **gray to yellow when the mouse is placed on the rectangle.**

Introduction to GUI applications

Python Containers: Canvas

- **To display some text in the canvas**. For this purpose, we should use the create_text() method as:
- **id = c.create_text(500, 100, text="My canvas", font= fnt, fill="yellow", activefill="green")**

  Here, the **'font' option** is showing 'fnt' object that can be created as:

  fnt =('Times', 40, 'bold')

  fnt =('Times', 40, 'bold italic underline')

Introduction to GUI applications

Python Containers: Canvas

```
from tkinter import *
#create root window
root = Tk()
#create Canvas as a child to root window
c = Canvas(root, bg="blue", height=700, width=1200, cursor='pencil')
#create a line in the canvas
id = c.create_line(50, 50, 200, 50, 200, 150, width=4, fill="white")
#create a rectangle in the canvas
id = c.create_rectangle(500, 200, 700, 600, width=2, fill="gray", outline="black", activefill="yellow")
#create some text in the canvas
fnt =('Times', 40, 'bold italic underline')
id = c.create_text(500, 100, text="My canvas", font= fnt, fill="yellow", activefill="green")
#add canvas to the root window
c.pack()
#wait for any events
root.mainloop()
```

Introduction to GUI applications

Python Containers: Canvas

program to display images in the canvas.

```
from tkinter import *
#create root window
root = Tk()
#create Canvas as a child to root window
c = Canvas(root, bg="white", height=700, width=1200)
#copy images into files
file1 = PhotoImage(file="first.gif")
#display the image in the canvas in NE direction
id = c.create_image(500, 200, anchor=NE, image=file1)
#add canvas to the root
c.pack()
```

Introduction to GUI applications

Python Containers: Frame

- A frame is similar to canvas that represents a rectangular area where some **text or widgets** can be displayed. Our root window is in fact a frame. 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 the object of Frame class. The frame is created as a child of 'root' window. The options **'height' and 'width'** represent the height and width of the 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** as follows: **f.pack()**

Introduction to GUI applications

Python Containers: Frame

program to display a frame in the root window.

```
from tkinter import *
#create root window
root = Tk()
#give a title for root window
root.title("My Frame")
#create a frame as child to root window
f= Frame(root, height=400, width=500, bg="yellow", cursor="cross")
 #attach the frame to root window
f.pack()
#let the root window wait for any events
root.mainloop()
```

Introduction to GUI applications

Python Widgets

- A widget is a GUI component that is displayed on the screen and can perform a task as desired by the user.
- We create widgets as objects. For example, a push button is a widget that is nothing but an object of Button class.
- Similarly, label is a widget that is an object of Label class. Once a widget is created, **it should be added to canvas or frame.**
- **The following are important widgets in Python:**

Button	Label	Message	Text
Scrollbar	Checkbutton	Radiobutton	Menu
Listbox	**etc.**		

Introduction to GUI applications

Python Widgets

- In general, working with widgets takes the following **four steps:**

1. **Create the widgets** that are needed in the program. A widget is a GUI component that is represented as an object of a class. For example, a push button is a widget that is represented as Button class object. An object to Button class as:

   b = Button(f, text='My Button')

   Here, **'f'** is **Frame object** to which the button is added. **'My Button'** is the **text** that is displayed on the button.

Introduction to GUI applications

Python Widgets

2. When the user interacts with a widget, an event is generated. For example, **clicking on a push button is an event.** Such **events should be handled by writing functions or routines.** These functions are called in response to the events. Hence they are called 'callback handlers' or 'event handlers'. Other examples for events are pressing the Enter button, right clicking the mouse button, etc. **E.g. let's write a function that may be called in response to button click.**

```
def buttonClick(self):
        print('You have clicked me')
```

Introduction to GUI applications

Python Widgets

3. When the user clicks on the push button, that 'clicking' event should be linked with the 'callback handler' function. Then only the button widget will appear as if it is performing some task. **E.g. let's bind the button click with the function as:**

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

Introduction to GUI applications

Python Widgets

4. The preceding 3 steps make the widgets ready for the user. Now, the user has to interact with the widgets. This is done by entering text from the keyboard or pressing mouse button. **These are called events.** These events are continuously monitored by our program with the help of a loop, called 'event loop'. **E.g. we can use the mainloop() method that waits and processes the events as:**

```
root.mainloop()
```

Introduction to GUI applications

Python Widgets

program to create one buttons and change the text in lable as the button clicked by the user.

```
from tkinter import *
class MyButtons:
  def __init__(self, root):
    self.f = Frame(root, height=500, width=700)
    self.f.propagate(0)
    self.f.pack()
    self.lbl = Label(self.f, text="HI", width=500, height=4, font=('Courier', 20, 'bold underline'), fg='blue')
    self.lbl.pack()
    self.b1 = Button(self.f, text='Welcome Message', width=15, height=2, command=self.buttonClick)
    #attach buttons to the frame
    self.b1.pack()

    #the event handler method
  def buttonClick(self):
      self.lbl.config(text = "Welcome to Python Programming")

root = Tk()
mb = MyButtons(root)
root.mainloop()
```

Introduction to GUI applications

Python Widgets

program to create two push buttons and change the background of the frame according to the button clicked by the user.

```
from tkinter import *
class MyButton:
  def __init__(self, root):
    self.f = Frame(root, height=400, width=500)
    self.f.propagate(0)
    self.f.pack()
    self.b1 = Button(self.f, text='Red', width=15, height=2, command=lambda: self.buttonClick(1))
    self.b2 = Button(self.f, text='Green', width=15, height=2, command=lambda: self.buttonClick(2))
    self.b1.pack()
    self.b2.pack()
  def buttonClick(self, num):
   if num==1:
    self.f["bg"]= 'red'
   if num==2:
    self.f["bg"]= 'green'

root = Tk()
mb = MyButton(root)
root.mainloop()
```

Introduction to GUI applications

Python Widgets

- we have to call the event handler function by passing some argument that represents which button is clicked. For example, b1 = Button(f, text='Red', width=15, height=2, command=lambda: buttonClick(1))
- Here, we are creating a push button 'b1'. Please observe the 'command' option. Here, we are using the 'lambda' expression to pass the argument 1 to the buttonClick() method, in the following format:

  command=lambda: buttonClick(arg1, arg2,...)

- When the button is clicked, we are changing the background color of the frame.

Introduction to GUI applications

Program to display welcome and bye message in label according to button click.

```
From tkinter import *
class MyButtons:
  def __init__(self, root):
    self.f = Frame(root, height=500, width=700)
    self.f.propagate(0)
    self.f.pack()
    self.lbl = Label(self.f, text="HI", width=500, height=4, font=('Courier', 20, 'bold underline'), fg='blue')
    self.lbl.pack()
    self.b1 = Button(self.f, text='Welcome Message', width=15, height=2, command=lambda: self.buttonClick(1))
    self.b2 = Button(self.f, text='Good Bye Message', width=15, height=2, command=lambda: self.buttonClick(2))
    #attach buttons to the frame
    self.b1.pack()
    self.b2.pack()
    #the event handler method
  def buttonClick(self,num):
    if num == 1:
      self.lbl.config(text = "Welcome to Python Programming")
    if num == 2:
      self.lbl.config(text = "Good Bye from Python Programming")
 root = Tk()
mb = MyButtons(root)
root.mainloop()
```