MODULE-03

Object Oriented Programming: Classes and Objects; Creating Classes and Objects; Constructor Method; Classes with Multiple Objects; Objects as Arguments; Objects as Return Values; Inheritance- Single and Multiple Inheritance, Multilevel and Multipath Inheritance; Encapsulation-Definition, Private Instance Variables; Polymorphism- Definition, Operator Overloading.

GU Interface: The tkinter Module; Window and Widgets; Layout Management- pack, grid and place.

Python SQLite: The SQLite3 module; SQLite Methods- connect, cursor,execute, close; Connect to Database; Create Table; Operations on Tables-Insert, Select, Update. Delete and Drop Records.

Data Analysis: NumPy- Introduction to NumPy, Array Creation using NumPy, Operations on Arrays; Pandas- Introduction to Pandas, Series and DataFrames, Creating DataFrames from Excel Sheet and .csv file,Dictionary and Tuples. Operations on DataFrames.

Data Visualisation: Introduction to Data Visualisation; Matplotlib Library; Different Types of Charts using Pyplot- Line chart, Bar chart and Histogram and Pie chart.

OBJECT ORIENTED PROGRAMMING

CLASS

- In object-oriented programming (OOP), a class is a blueprint or template for creating objects (instances).
- It defines the common attributes (data) and behaviors (methods) that objects of that class will have.
- A class serves as a blueprint from which objects are created, each possessing its ownunique set of data.
- In simple terms, a class is like a blueprint for creating objects, and an object is aninstance of a class.

Here's an example of a Python class representing a basic car:

```
class Car:
  def___init_(self, make, model, year):
  self.make = make
     self.model = model
```

```
        self.year  = year
    def drive(self):

        print("The  car is driving.")
    def stop(self):
        print("The  car has  stopped.")
```

- In this example, the Car class has three attributes: make, model, and year.
- These attributes represent the data associated with a car object, such as its make (e.g., Toyota), model (e.g., Camry), and year of manufacture.
- The class also has two methods: drive() and stop().
- These methods define the behavior or actions that a car object can perform.
- For example, the drive() method prints a message indicating that the car is driving, and the stop() method prints a message indicating that the car has stopped.

<u>OBJECT</u>

- In object-oriented programming (OOP), an object is an instance of a class.
- It is a entity that represents a specific instance or occurrence of the class, with its own unique set of data (attributes) and behavior (methods).
- In simpler terms, an object can be thought of as a real-world entity or a "thing" that has certain characteristics and can perform certain actions.
- Let's take the example of the Car class. An object of the Car class represents a specific car with its own unique make, model, and year. Here's an example of creating an object (instance) of the Car class.

my_car = Car("Toyota", "Camry", 2021)

- In this example, my_car is an object of the Car class. It represents a specific car instance with the make "Toyota", model "Camry", and year 2021.
- Once an object is created, we can access its attributes and methods using dot notation.

For example:

```
print(my_car.make)  # Output: Toyota

print(my_car.model)  # Output: Camry

print(my_car.year)  # Output: 2021

my_car.drive()  # Output: The car is driving.

my_car.stop()   # Output: The car has stopped.
```

- In this code we access the attributes of my_car (make, model, year) using dot notation (my_car.make, my_car.model, my_car.year). We can also call the methods of my_car (drive(), stop()) using dot notation (my_car.drive(), my_car.stop()).
- Each object of a class can have its own set of attribute values, which makes it distinct and separate from other objects of the same class.

CREATING CLASS

To create a class in Python, you can use the class keyword followed by the name of the class. The class can contain attributes (data) and methods (functions) that define its behavior. Here's a basic example of creating a class in Python:

```
class MyClass:

    def__init__(self, name):
        self.name = name

    def greet(self):

        print("Hello, " + self.name + "!")

# Creating an object of the class

obj = MyClass("John")

# Accessing attributes and calling methods of the object

print(obj.name)      # Output: John

obj.greet()        # Output: Hello, John!
```

- In the above example, we define a class named MyClass. It has an __init___method (constructor) that initializes the name attribute of the class. The greet method is a simple method that prints a greeting message using the name attribute.
- To create an instance (object) of the class, we simply call the class name followed by parentheses, passing any required arguments to the __init___method. In this case, we create an object obj of MyClass and pass the name "John" to initialize the name attribute.
- We can then access the attributes and call the methods of the object using the dot notation. In the example, we print the value of the name attribute and call the greet method to display the greeting message.
- By creating classes in Python, you can define your own custom types with specific attributes and behaviors, encapsulating related data and functionality into a single unit.

CREATING OBJECT

- To create an object in Python, you need to instantiate a class. An object, also referred to as an instance, is a specific occurrence of a class, representing a unique entity with its own set of attributes and behaviors. Here's an example of creating an object in Python:

```
class MyClass:
    def___init_(self, name):
        self.name = name
    def greet(self):
        print("Hello, " + self.name + "!")
# Creating an object of the class
obj = MyClass("John")
```

- In the example above, we have a class called MyClass with an_init_method that initializes the name attribute. To create an object, we use the class name followed by parentheses and pass any required arguments to the_init_method. In this case, we create an object named obj and pass the name "John" as an argument.
- Once the object is created, you can access its attributes and call its methods using the dot notation. For instance, you can access the name attribute of the object using obj.name. Similarly, you can call the greet method of the object using obj.greet().

Here's an example of accessing the attributes and calling methods of the created object:

```
print(obj.name)     # Output: John

obj.greet()         # Output: Hello, John!
```

- In the code above, we print the value of the name attribute using obj.name, and then we call the greet method using obj.greet(), which will display the greeting message "Hello, John!".
- By creating objects in Python, you can work with instances of classes and utilize their attributes and behaviors to perform specific tasks and operations.

CLASS WITH MULTIPLE OBJECT

- In object-oriented programming (OOP), you can create multiple objects (instances) ofa class.
- Each object represents a distinct instance with its own set of data and behavior.
- Here's an example of a Python class called Rectangle with multiple objects:

```
class Rectangle:

   def___init_(self, width, height):

      self.width = width

      self.height = height

   def area(self):

      return self.width * self.height

# Creating objects of the Rectangle class

rectangle1 = Rectangle(4, 5)

rectangle2 = Rectangle(3, 6)

# Accessing attributes and calling methods of each object

print(rectangle1.width) # Output: 4
print(rectangle1.height)  # Output: 5
print(rectangle1.area())  # Output: 20
print(rectangle2.width)  # Output: 3
print(rectangle2.height)  # Output: 6
print(rectangle2.area())  # Output: 18
```

- In this example, we define the Rectangle class with an __init___method that initializes the width and height attributes of each object.
- The class also has a method called area that calculates and returns the area of the rectangle.
- We then create two objects, rectangle1 and rectangle2, by calling the Rectangle class constructor with different arguments.
- Each object has its own set of width and height attributes.
- We can access the attributes (width, height) and call the method (area) of each object separately.
- The attributes and methods of each object are independent of each other.
- By creating multiple objects of a class, you can represent multiple instances of the same entity or concept, each with its own unique set of data and behavior.
- This allows you to work with and manipulate distinct objects individually, enabling modular and reusable code structures.

OBJECTS AS ARGUMENTS

- In object-oriented programming, you can pass objects as arguments to functions ormethods.
- By doing so, you can perform operations on the attributes and behaviors of the objects within the function.
- Here's an example that demonstrates passing objects as arguments:

```
class Rectangle:
    def___init_(self, width, height):
        self.width = width
        self.height = height
    def area(self):
        return self.width * self.height
def print_rectangle_area(rect):
    area = rect.area()
    print(f"The area of the rectangle is: {area}")
# Creating an object of the Rectangle class
rectangle = Rectangle(4, 5)
# Calling the function with the object as an argument
print_rectangle_area(rectangle)
```

- In this example, we have a Rectangle class with an_init_method to initialize the width and height attributes and an area method to calculate the area of the rectangle. The
- print_rectangle_area function takes a rect parameter, which expects an object of the Rectangle class. Inside the function, we call the area method on the rect object to calculate the area of the rectangle and then print the result.
- We create an object rectangle of the Rectangle class and pass it as an argument to the print_rectangle_area function. The function then operates on the attributes and behavior of the rectangle object, calculating and printing its area.
- By passing objects as arguments, you can encapsulate related data and behavior within objects and manipulate them using functions or methods. This promotes modularity, reusability, and code organization, allowing you to work with objects and perform operations on them in a structured and cohesive manner.

OBJECTS AS RETURN VALUE

- In object-oriented programming, functions or methods can also return objects as their return values.
- This allows you to perform operations on objects and return the resulting object as the output of the function.
- Here's an example that demonstrates returning objects from a function:

Example

```
class Rectangle:
    def___init_(self, width, height):
        self.width = width
        self.height = height
    def area(self):
        return self.width * self.height
def create_square(side_length):
    square = Rectangle(side_length, side_length)
    return square
# Calling the function and receiving the returned object
my_square = create_square(5)
# Accessing attributes and calling methods of the returned object
print(my_square.width) # Output: 5
print(my_square.height) # Output: 5
print(my_square.area()) # Output: 25
```

- In this example, we have a Rectangle class with an __init___method to initialize the width and height attributes and an area method to calculate the area of the rectangle.
- The create_square function takes a side_length parameter and creates a square object by creating an instance of the Rectangle class with equal width and height. The square object is then returned as the output of the function.
- We call the create_square function with a side length of 5 and store the returned object in the variable my_square. We can then access the attributes (width, height) and call the method (area) of the my_square object.
- By returning objects from functions, you can perform operations on objects and encapsulate the resulting object within the function. This allows you to create and manipulate objects in a more modular and flexible way, enabling code reuse and promoting a clean and organized code structure.

INHERITANCE

- Inheritance can be defined as a process through which one class acquires the features(attributes and methods) of an existing class without modifying it.
- The class which inherits the features is referred to as child class or derived class and the class from which the features inherited is referred to as parent class or base class.
- In other words, the newly formed class is the child class while the existing class is known as the parent class.
- For instance, in the real world, a father and mother denote the parent class while their kids denote the child class.
- A kid has acquired several features from their parents and at the same time, the kid has got some unique features that the parents may not have. In programming terms,

we can say that a child class acquires all the attributes and methods of a parent class, but at the same time child class holds its own unique characteristics.

Syntax

Class Parent_class:

   Body of parent_class

Class Child_class(Parent_class):

   Body of Child_class

Different types of inheritance in python

In python, Inheritance is divided into five types. They are :

- Single Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance
- Multiple Inheritance
- Hybrid Inheritance

Single Inheritance in Python

- Single inheritance is one of the simplest and easiest types of inheritance.
- In single inheritance, the child class inherits all the attributes and methods in the parent class.
- This enables code reusability and modularity.
- In this case, class A is the parent class and class B is the child class. Class B has its own unique attributes and method even then it inherits the attributes and methods of class A.

Syntax

```
class ParentClass:

   # Parent class attributes and methods

class ChildClass(ParentClass):

   # Child class attributes and methods
```

Example

class employee1()://This is a parent class

```
    def__init__(self, name, age, salary):
        self.name = name
        self.age = age
        self.salary = salary
class childemployee(employee1)://This is a child class
    def__init_(self, name, age, salary,id):
        self.name = name
        self.age = age
        self.salary = salary
        self.id = id
emp1 = employee1('harshit',22,1000)
print(emp1.age)
```

Output: 22

Multilevel Inheritance in Python

Multilevel inheritance is a type of inheritance where a class is created from a derived class which itself inherits a base class. A multilevel inheritance shows the possibility of inheriting from a derived class or child class. So a minimal multilevel inheritance is 3 tier structure with a child class that inherits features from a derived class which in turn inherits the properties from a superclass

Syntax

```
class Grandparent:
  # Grandparent class attributes and methods
class Parent(Grandparent):
  # Parent class attributes and methods
class Child(Parent):
  # Child class attributes and methods
```

In this example, the DerivedClass inherits from the IntermediateClass, which in turn inherits from the BaseClass, creating a multilevel inheritance chain. The DerivedClass can access and extend the attributes and methods defined in both parent classes.

Example

```
class employee()://Super class

  def__init__(self,name,age,salary):
        self.name  = name
        self.age  = age
        self.salary  = salary
class childemployee1(employee)://First  child  class
    def__init__(self,name,age,salary):
        self.name  = name
        self.age  = age
        self.salary  = salary
class childemployee2(childemployee1)://Second  child  class
    def__init__(self, name, age, salary):
        self.name  = name
        self.age  = age
        self.salary  = salary
emp1  = employee('harshit',22,1000)
emp2  = childemployee1('arjun',23,2000)
print(emp1.age)
print(emp2.age)
```

Output: 22,23

Multiple Python Inheritance

- Multiple inheritance is a concept in object-oriented programming (OOP) where aderived class inherits properties and behavior from multiple base classes.
- In this type of inheritance, a class can inherit attributes and methods from more than one parent class, combining their functionality into a single derived class.

Syntax

```
class ParentClass1:
    # Parent Class 1 attributes  and methods
class ParentClass2:
    # Parent Class 2 attributes  and methods
class ChildClass(ParentClass1, ParentClass2):
    # Child Class attributes  and methods
```

- In this the Child Class inherits from both ParentClass1 and ParentClass2 using multiple inheritance.
- The child class can access and extend the attributes and methods defined in both parent classes.
- However, it's important to carefully handle potential conflicts or ambiguity that may arise from inheriting multiple classes with overlapping names or functionalities.

Example

```
class employee1(): //Parent class
     def___init_(self, name, age, salary):
        self.name = name
       self.age = age
      self.salary = salary
class employee2(): //Parent class
      def___init_(self,name,age,salary,id):
        self.name = name
       self.age = age
       self.salary = salary
       self.id = id
class childemployee(employee1,employee2):
     def___init_(self, name, age, salary,id):
     self.name = name
     self.age = age
     self.salary = salary
     self.id = id
emp1 = employee1('harshit',22,1000)
emp2 = employee2('arjun',23,2000,1234)
```

Hierarchical Inheritance

- Hierarchical inheritance is a concept in object-oriented programming (OOP) where multiple derived classes inherit properties and behavior from a single base class.
- In this type of inheritance, a base class serves as the common parent for multiple derived classes, each of which can have its own additional attributes and methods.
- Hierarchical inheritance allows for creating a hierarchical structure of classes, where different derived classes can have their own specialized features while sharing common characteristics defined in the base class.

- The derived classes can access and extend the functionality defined in the Base Class. They can add their own unique attributes and methods while still inheriting and utilizing the common features defined in the base class.

  Syntax:

  ```
  class BaseClass:
     # Base class attributes and methods
  class DerivedClass1(BaseClass):
     # Derived class 1 attributes and methods
  class DerivedClass2(BaseClass):
     # Derived class 2 attributes and methods
  ```

- In this both DerivedClass1 and DerivedClass2 inherit from the BaseClass, demonstrating hierarchical inheritance. The derived classes can access and extend the attributes and methods defined in the base class.

Example

```
class employee():
    def__init_(self, name, age, salary): //Hierarchical Inheritance
       self.name = name
       self.age = age
       self.salary = salary
class childemployee1(employee):
      def__init_(self,name,age,salary):
         self.name = name
         self.age = age
         self.salary = salary
class childemployee2(employee):
     def__init__(self, name, age, salary):
        self.name = name
      self.age = age
      self.salary = salary
   emp1 = employee('harshit',22,1000)
   emp2 = employee('arjun',23,2000)
```

Hybrid Python Inheritance

Hybrid inheritance is a combination of multiple types of inheritance, such as multiple inheritance and multilevel inheritance. It allows a class to inherit properties and behavior from multiple base classes as well as from classes in a multilevel inheritance hierarchy.

Syntax

```
class BaseClass1:

   # Base Class 1 attributes and methods
class BaseClass2:
   # Base Class 2 attributes and methods
class IntermediateClass(BaseClass1):
   # Intermediate Class attributes and methods

class DerivedClass(IntermediateClass, BaseClass2):
   # Derived Class attributes and methods
```

- In this the DerivedClass inherits from both IntermediateClass and BaseClass2, combining multiple inheritance and multilevel inheritance. The DerivedClass can access and extend the attributes and methods defined in all parent classes involved in the hybrid inheritance.

Multipath inheritance

- Multipath inheritance refers to a situation in object-oriented programming where a class inherits from multiple base classes, and there exists a common ancestor class in the inheritance hierarchy from which the derived class inherits indirectly through multiple paths.
- In a typical inheritance hierarchy, a derived class can inherit from a single base class. However, in multipath inheritance, a derived class can inherit from multiple base classes, which can lead to more complex relationships and potential conflicts.

Example

```
class A:
    def method_a(self):
        print("This is method A.")
class B(A):
    def method_b(self):
        print("This is method B.")
class C(A):
    def method_c(self):
        print("This is method C.")
class D(B, C):
    def method_d(self):
        print("This is method D.")
```

- In this example, class D inherits from both class B and class C, which means it indirectly inherits from class A through multiple paths. Here's how the inheritance hierarchy looks:

```
    A
   / \
  B  C
   \   /
    D
```

- As a result, class D has access to all the methods and attributes of classes A, B, and C. However, if there are conflicts between methods or attributes inherited from multiple paths (e.g., if both class B and class C have a method with the same name), it can create ambiguity, and the programmer needs to carefully handle such cases.

- Multipath inheritance can offer flexibility and code reuse in certain scenarios, but it can also make the code more complex and harder to maintain. Therefore, it is generally recommended to use multipath inheritance judiciously and consider alternative design patterns, such as composition or interfaces, to achieve the desired functionality when possible.

ADVANTAGES OF PYTHON INHERITANCE

- **Modular Codebase:** Increases modularity, i.e., breaking down the codebase into modules, making it easier to understand. Here, each class we define becomes a separate module that can be inherited separately by one or many classes.
- **Code Reusability:** the child class copies all the attributes and methods of the parent class into its class and use. It saves time and coding effort by not rewriting them, thus following modularity paradigms.
- **Less Development and Maintenance Costs:** changes must be made in the base class, and all derived classes will automatically follow.
- **Reusability** – Inheritance allows obtaining new classes from existing classes without modification. This helps reusability of information in the child class and adds extra functionality.

DISADVANTAGES OF PYTHON INHERITANCE

- **Decreases the Execution Speed:** loading multiple classes because they are interdependent
- **Tightly Coupled Classes:** Even though parent classes can be executed independently, child classes can only be conducted by defining their parent classes.

CONSTRUCTOR

- A constructor is a special type of method (function) which is used to initialize the instance members of the class.
- Constructors can be of two types.
  - Parameterized Constructor
  - Non-parameterized Constructor
- Constructor is executed when we create the object of this class. Constructors also verify that there are enough resources for the object to perform any start-up task.

CREATING THE CONSTRUCTOR IN PYTHON

- In Python, the method the __init_() simulates the constructor of the class. This method is called when the class is instantiated.It accepts the self-keyword as a first argument which allows accessing the attributes or method of the class.
- We can pass any number of arguments at the time of creating the class object, depending upon the___init_() definition. It is used to initialize the class attributes.

- A class provides a special method, **init_**. This method, knownas 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

RULES FOR PYTHON CONSTRUCTOR

- The constructor method must be named __init_. This is a special name that is recognized by Python as the constructor method.
- The first argument of the constructor method must be self. This is a reference to the object itself, and it is used to access the object's attributes and methods.
- The constructor method must be defined inside the class definition. It cannot bedefined outside the class.
- The constructor method is called automatically when an object is created. You don't need to call it explicitly.
- You can define both default and parameterized constructors in a class. If you define both, the parameterized constructor will be used when you pass arguments to the object constructor, and the default constructor will be used when you don't pass any arguments.

Example

```
class Employee:
   def__init_(self, name, id):
      self.id = id
      self.name = name
   def display(self):
      print("ID: %d \nName: %s" % (self.id, self.name))
emp1 = Employee("Janav", 101)
emp2 = Employee("Divya", 102)
 emp1.display() # accessing display() method to print employee 1 information
emp2.display() # accessing display() method to print employee 2 information
```

Output:

```
ID: 101
Name: Janav
ID: 102
Name: Divya
```

TYPES OF CONSTRUCTOR

- **Default Constructor:** A default constructor is a constructor that takes no arguments. It is used to create an object with default values for its attributes.
- **Parameterized Constructor:** A parameterized constructor is a constructor that takes one or more arguments. It is used to create an object with custom values for its attributes.
- **Non-Parameterized Constructor:** A non-parameterized constructor is a constructor that does not take any arguments. It is a special method in Python that is called when you create an instance of a class. The non-parameterized constructor is used to initialize the default values for the instance variables of the object.

PYTHON PARAMETERIZED CONSTRUCTOR

Parameterized constructors are useful when you want to create an object with custom values for its attributes. They allow you to specify the values of the object's attributes when the object is created, rather than using default values.

Here is an example of a class with a parameterized constructor:

```
class Person:
   def__init__(self, name, age):
      self.name = name
      self.age = age
person = Person("Alice", 25)
print(person.name)
print(person.age)
```

Output:

```
Alice
25
```

- In this example, the **__init__** method is the parameterized constructor for the **Person** class.
- It takes two arguments, **name** and **age**, and it sets the values of the name and age attributes of the object to the values of these arguments.

PYTHON DEFAULT CONSTRUCTOR

Default constructors are useful when you want to create an object with a predefined set of attributes, but you don't want to specify the values of those attributes when the object is created. Here is an example of a default constructor:

Example

```
class Person:

    def__init_(self):

        self.name = "John"

        self.age = 30

person = Person()

print(person.name)

print(person.age)
```

Output:

```
John
30
```

In this example, the __init___method is the default constructor for the Person class. It is called automatically when the object is created, and it sets the default values for the name and age attributes.

NON-PARAMETERIZED CONSTRUCTORS

- There is not necessarily a need for a non-parameterized constructor in Python. It is upto the programmer to decide whether to include a non-parameterized constructor in a class.
- However, a non-parameterized constructor can be useful in the following cases:
- When you want to initialize the default values for the instance variables of an object.
- When you want to perform some operations when an object is created, such as opening a file or establishing a connection to a database.
- When you want to create a “skeleton” object that can be used as a template for creating other objects.

Example

```
class MyClass:
    def__init_(self):
        self.arg1 = 10
        self.arg2 = 20
obj = MyClass()
print(obj.arg1)
print(obj.arg2)
```

Output:

```
10

20
```

In this case, the non-parameterized constructor is used to initialize the default values for the instance variables arg1 and arg2. If you create an instance of the MyClass class without passing any arguments, the default values will be used.

ENCAPSULATION

Encapsulation is an object-oriented programming concept that involves hiding the internal state and implementation details of an object and providing controlled access to it through methods.In Python, you can achieve encapsulation by using private instance variables.

PRIVATE INSTANCE VARIABLES

Private instance variable are variables that are intended to be accessed only within the class itself and not from outside the class.

They are typically denoted by prefixing an underscore (_) to their names. Although Python doesn't enforce strict data hiding or access restrictions, the use of the underscore convention indicates that the variable should be treated as private and should not be accessed directly from outside the class.

Example

```
class Person:

  def___init_(self, name, age):

    self._name = name

    self._age = age

  def get_name(self):
    return self._name

  def get_age(self):

    return self._age

  def set_age(self, age):

    if age > 0:

      self._age = age
```

```
# Creating an object of the Person class
person = Person("John", 25)
# Accessing private instance variables using public methods
print(person.get_name()) # Output: John
print(person.get_age()) # Output: 25
# Attempting to access private instance variables directly
print(person._name) # Output: John
print(person._age) # Output: 25
# Modifying private instance variables using public methods
person.set_age(30)
print(person.get_age()) # Output: 30
```

- In this example, the Person class has private instance variables _name and _age.
- These variables are accessed through public methods get_name() and get_age().
- The set_age() method is provided to modify the _age variable, but with validation to ensure a positive age value.

POLYMORPHISM

Polymorphism is having different forms. Polymorphism refers to a function having the same name but being used in different ways and different scenarios.It basically creates a structure that can use many forms of objects. This polymorphism may be accomplished in two distinct ways: overloading and overriding.

Operator overloading

Operator overloading is a kind of overloading in which an operator may be used in ways other than those stated in its predefined definition.

```
>>>print(2*7)
14
>>>print("a"*3)
aaa
```

Thus, in the first example, the multiplication operator multiplied two numbers; but, in the second, since

multiplication of a string and an integer is not feasible, the character is displayed three times twice.

Example

```
class example:

    def__init_(self, X):

        self.X = X

      # adding two objects

       def__add_(self, U):

        return self.X + U.X

object_1 = example( int( input( print ("Please enter the value: "))))

object_2 = example( int( input( print ("Please enter the value: "))))

print (": ", object_1 + object_2)

object_3 = example(str( input( print ("Please enter the value: "))))

object_4 = example(str( input( print ("Please enter the value: "))))

print (": ", object_3 + object_4)
```

Output

```
Please enter the value: 23

Please enter the value: 21

: 44

Please enter the value: Python

Please enter the value: Programming

: PythonProgramming
```

GUI PROGRAMMING IN PYTHON

A graphical user interface (GUI) is a desktop interface that allows you to communicate with computers. They carry out various activities on desktop computers, laptops, and other mobile devices.

There are many ways to develop GUI based programs in Python. These different ways are given below:

1. **Tkinter:**

   In Python, Tkinter is a standard GUI (graphical user interface) package. Tkinter is Python's default GUI module and also the most common way that is used for GUI programming in Python. Note that Tkinter is a set of wrappers that implement the Tk widgets as Python classes.

2. **wxPython:**

   This is basically an open-source, cross-platform GUI toolkit that is written in C++. Also an alternative to Tkinter.

3. **JPython:**

   JPython is a Python platform for Java that is providing Python scripts seamless access to Java class Libraries for the local machine.

WHAT IS TKINTER?

- Tkinter in Python helps in creating GUI Applications. Among various GUI Frameworks, Tkinter is the only framework that is built-in into
- An important feature in favor of Tkinter is that it is cross-platform, so the same code can easily work on Windows, macOS, and Linux.
- Tkinter is a lightweight module.
- It is simple to use.

WHAT ARE TCL, TK, AND TKINTER?

- Tkinter is Python's default GUI library, which is nothing but a wrapper module on top of the Tk toolkit.
- Tkinter is based upon the Tk toolkit, and which was originally designed for the Tool Command Language (Tcl).
- As Tk is very popular thus it has been ported to a variety of other scripting languages, including Perl (Perl/Tk), Ruby (Ruby/Tk), and Python (Tkinter).
- GUI development portability and flexibility of Tk makes it the right tool which can be used to design and implement a wide variety of commercial-quality GUI applications.

- Python with Tkinter provides us a faster and efficient way in order to build useful applications that would have taken much time if you had toprogram directly in C/C++ with the help of native OS system libraries.

INSTALL TKINTER

- Tkinter may be already installed on your system along with Python. But it is not true always. So let's first check if it is available.
- If you do not have Python installed on your system - Install Python 3.8 first, and then check for Tkinter.
- You can determine whether Tkinter is available for your Python interpreter by attempting to import the Tkinter module - If Tkinter is available, then there will be no errors, as demonstrated in the following code:

```
import tkinter
```

- If you see any error like module not found, etc, then your Python interpreter was not compiled with Tkinter enabled, the module import fails and you might need to recompile your Python interpreter to gain access to Tkinter.

ADDING TK TO YOUR APPLICATIONS

Basic steps of setting up a GUI application using Tkinter in Python are as follows:

1. First of all, import the Tkinter module.
2. The second step is to create a top-level windowing object that contains your entire GUI application.
3. Then in the third step, you need to set up all your GUI components and their functionality.
4. Then you need to connect these GUI components to the underlying application code.
5. Then just enter the main event loop using mainloop()

<u>TKINTER WINDOWS</u>

- The Tkinter window is the foundational element of the Tkinter GUI.
- Tkinter window is a container in which all other GUI elements(widgets) live.
- Here is the syntax for creating a basic Tkinter Window

```
win = Tk()
```

- The top-level window object in GUI Programming contains all of the little window objects that will be part of your complete GUI.
- The little window objects can be text labels, buttons, list boxes, etc., and these individual little GUI components are known as Widgets.

- If you see any error like module not found, etc, then your Python interpreter was not compiled with Tkinter enabled**,** the module import fails and you might need to recompile your Python interpreter to gain access to Tkinter**.**

ADDING TK TO YOUR APPLICATIONS

Basic steps of setting up a GUI application using Tkinter in Python are as follows:

1. First of all, import the Tkinter module.
2. The second step is to create a top-level windowing object that contains your entire GUI application.
3. Then in the third step, you need to set up all your GUI components and their functionality.
4. Then you need to connect these GUI components to the underlying application code.
5. Then just enter the main event loop using mainloop()

TKINTER WINDOWS

- The Tkinter window is the foundational element of the Tkinter GUI.
- Tkinter window is a container in which all other GUI elements(widgets) live.
- Here is the syntax for creating a basic Tkinter Window

```
win = Tk()
```

- The top-level window object in GUI Programming contains all of the little window objects that will be part of your complete GUI.
- The little window objects can be text labels, buttons, list boxes, etc., and these individual little GUI components are known as Widgets

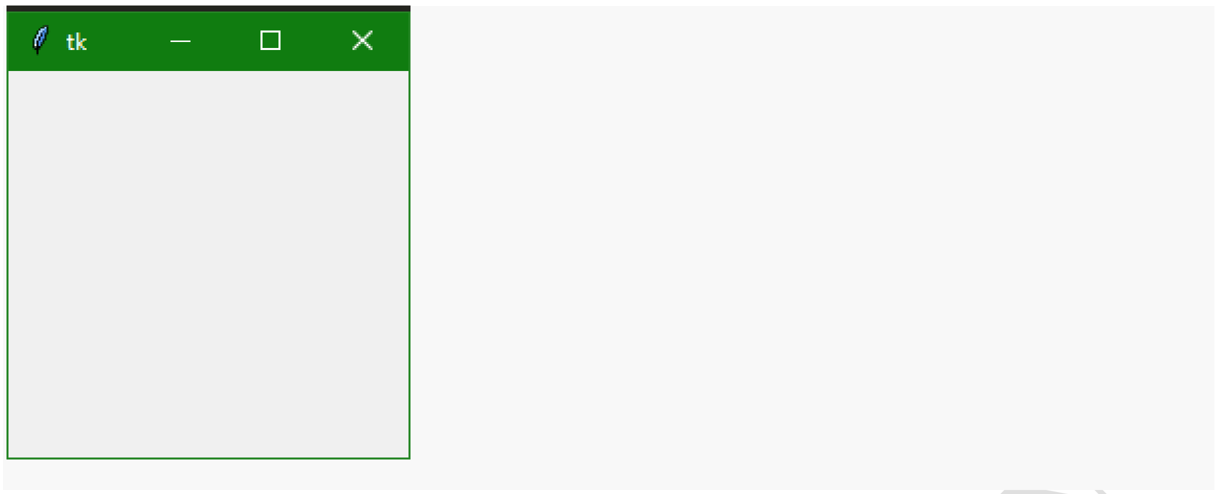
- So, having a top-level window object will act as a container where you will put all your widgets. In Python, you'd typically do so like this using the following code: win = tkinter.Tk()
- The object that is returned by making a call to tkinter.Tk() is usually referred to as the Root Window.
- Top-level windows are mainly stand-alone as part of your application, also you can have more than one top-level window for your GUI
- First of all, you need to design all your widgets completely, and then addthe real functionality.
- The widgets can either be stand-alone or can be containers. If one widget contains other widgets, it is considered the parent of those widgets.
- Similarly, if a widget is contained within another widget, it's known as achild of the parent, the parent is the next immediate enclosing container widget.
- The widgets also have some associated behaviours, such as when a button is pressed, or text is filled into a text field, so we have a Tkinter Windows The term "Window"
- has different meanings in the different contexts, But generally "Window" refers to a rectangular area somewhere on the user's display screen through which you can interact.

FIRST TKINTER EXAMPLE

As mentioned earlier that in GUI programming all main widgets are only built on the top-level window object.

The top-level window object is created by the Tk class in Tkinter. Let us create a top-level window:

```
import tkinter as tk
win = tk.Tk()
win.mainloop()
```



TKINTER METHODS USED ABOVE:

The two main methods are used while creating the Python application with GUI. You must need to remember them and these are given below:

1. Tk(screenName=None, baseName=None, className='Tk', useTk=1)

This method is mainly used to create the main window. You can also change the name of the window if you want, just by changing the className to the desired one.

The code used to create the main window of the application and we have also used it in our above example:

```
win = tkinter.Tk()        ## where win indicates name of the main window object
```

2. The mainloop() Function

This method is used to start the application. The mainloop() function is an infinite loop which is used to run the application, it will wait for an event to occur and process the event as long as the window is not closed.

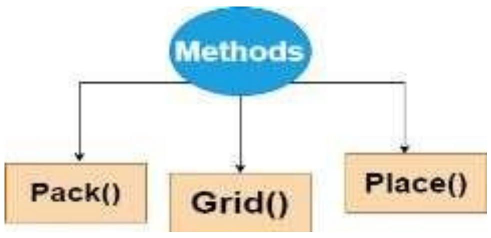
PYTHON TKINTER GEOMETRY MANAGER

- In this tutorial, we will learn how to control the layout of the Application with the help of the Tkinter Geometry Managers.

Controlling Tkinter Application Layout

- In order to organize or arrange or place all the widgets in the parent window, Tkinter provides us the geometric configuration of the widgets.
- The GUI Application Layout is mainly controlled by Geometric Managers of Tkinter.
- It is important to note that each window and Frame in your application is allowed to use only one geometry manager. Also, different frames can use different geometry managers, even if they're already assigned to a frame or window using another geometry manager.

- There are mainly three methods in Geometry Managers:



1. Tkinter **pack()** Geometry Manager

- The pack() method mainly uses a packing algorithm in order to place widgets in a Frame or window in a specified order.
- This method is mainly used to organize the widgets in a block**.**

Packing Algorithm:

The steps of Packing algorithm are as follows:

- Firstly this algorithm will compute a rectangular area known as
  a Parcel which is tall (or wide) enough to hold the widget and then it will fill the remaining width (or height) in the window with blank space.
- It will center the widget until any different location is specified.
- This method is powerful but it is difficult to visualize.

Here is the syntax for using pack() function:

```
widget.pack(options)
```

The possible options as a parameter to this method are given below:

1. **Fill:**The default value of this option is set to NONE**.** Also, we can set it to X or Y in order to determine whether the widget contains any extra space**.**
2. Side

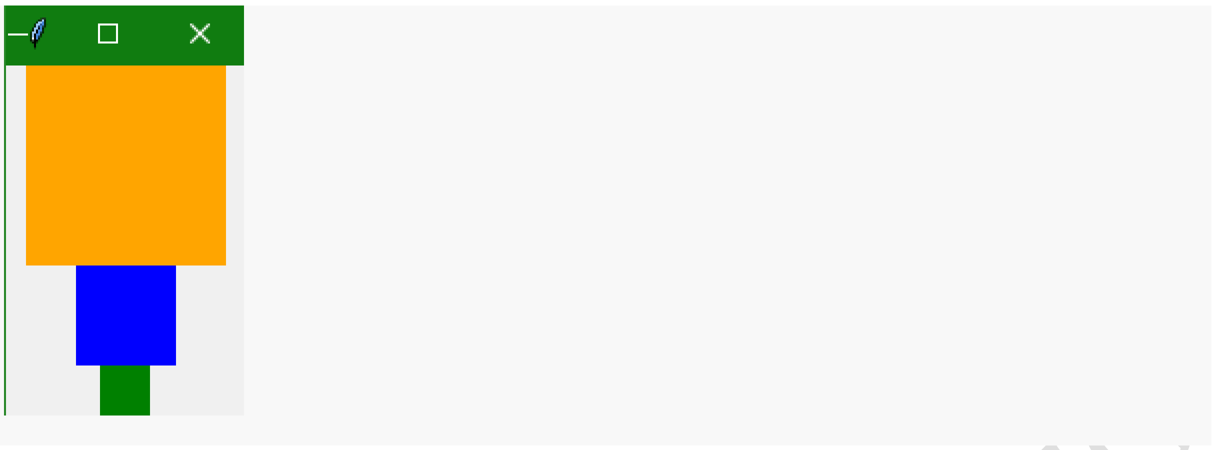
   This option specifies which side to pack the widget against**.** If you want to pack widgets vertically, use TOP which is the default value. If you want to pack widgets horizontally, use LEFT**.**

3. Expand

   This option is used to specify whether the widgets should be expanded to fill any extra space in the geometry master or not. Its default value is false. If it is false then the widget is not expanded otherwise widget expands to fill extra space.

Tkinter pack() Geometry Manager Example:

```
import tkinter as tk
win = tk.Tk()
# add an orange frame
frame1 = tk.Frame(master=win, width=100, height=100, bg="orange")
frame1.pack()
# add blue frame
frame2 = tk.Frame(master=win, width=50, height=50, bg="blue")
frame2.pack()
# add green frame
frame3 = tk.Frame(master=win, width=25, height=25, bg="green")
frame3.pack()
window.mainloop()
```



According to the output of the above code, the pack() method just places each Frame below the previous one by default, in the same order in which they're assigned to the window.

Tkinter pack() with Parameters

- Let's take a few more code examples using the parameters of this function like fill, side and, expand.
- You can set the fill argument in order to specify in which direction you want the frames should fill.
- If you want to fill in the horizontal direction then the option is tk.X, whereas, tk.Y is used to fill vertically, and to fill in both directions tk.BOTH is used.

2. Tkinter grid() Geometry Manager

The most used geometry manager is grid() because it provides all the power of pack() function but in an easier and maintainable way.

The grid() geometry manager is mainly used to split either a window or frame into rows and columns.

- You can easily specify the location of a widget just by calling grid() function and passing the row and column indices to the row and column keyword arguments, respectively.

- Index of both the row and column starts from 0, so a row index of 2 and acolumn index of 2 tells the grid() function to place a widget in the third column of the third row(0 is first, 1 is second and 2 means third).

Here is the syntax of the grid() function:

widget.grid(options)

The possible options as a parameter to this method are given below:

- **Column**

  This option specifies the column number in which the widget is to beplaced. The index of leftmost column is 0.

- **Row**

  This option specifies the row number in which the widget is to be placed. The topmost row is represented by 0.

- **Columnspan**

  This option specifies the width of the widget. It mainly represents the number of columns up to which, the column is expanded.

- **Rowspan**

  This option specifies the height of the widget. It mainly represents thenumber of rows up to which, the row is expanded.

- **padx, pady**

  This option mainly represents the number of pixels of padding to beadded to the widget just outside the widget's border.

- **ipadx, ipady**

  This option is mainly used to represents the number of pixels of padding to be added to the widget inside the widget's border

3. Trinket place() Geometry Manager

The place() Geometry Manager organizes the widgets to place them in aspecific position as directed by the programmer.

- This method basically organizes the widget in accordance with its x and ycoordinates. Both x and y coordinates are in pixels.
- Thus the origin (where x and y are both 0) is the top-left corner ofthe Frame or the window.
- Thus, the y argument specifies the number of pixels of space from the topof the window, to place the widget, and the x argument specifies the number of pixels from the left of the window.

Here is the syntax of the place() method:

widget.place(options)

The possible options as a parameter to this method are given below:

- x, y

  This option indicates the horizontal and vertical offset in the pixels.

- height, width

  This option indicates the height and weight of the widget in the pixels.

- Anchor

  This option mainly represents the exact position of the widget within the container. The default value (direction) is NW that is (the upper left corner).

- bordermode

  This option indicates the default value of the border type which is INSIDE and it also refers to ignore the parent's inside the border. The other option is OUTSIDE.

- relx, rely

  This option is used to represent the float between 0.0 and 1.0 and it isthe offset in the horizontal and vertical direction.

- relheight, relwidth

  This option is used to represent the float value between 0.0 and 1.0indicating the fraction of the parent's height and width.

PYTHON WIDGETS

Button	**If you want to add a button in your application then Button widget will be used.**
Canvas	**To draw a complex layout and pictures (like graphics, text, etc.)Canvas Widget will be used.**
CheckButton	**If you want to display a number of options as checkboxes then Checkbutton widget is used. It allows you to select multiple options at a time.**
Entry	**To display a single-line text field that accepts values from the user Entry widget will be used.**
Frame	**In order to group and organize another widgets Frame widget will be used.Basically it acts as a container that holds other widgets.**
Label	**To Provide a single line caption to another widget Label widget will be used.It can contain images too.**
Listbox	**To provide a user with a list of options the Listbox widget will be used.**

Menu	To provides commands to the user Menu widget will be used. Basically these commands are inside the Menubutton. This widget mainly creates allkinds of Menus required in the application.
Menubutton	The Menubutton widget is used to display the menu items to the user.
Message	The message widget mainly displays a message box to the user. Basically it is a multi-line text which is non-editable.
Radiobutton	If you want the number of options to be displayed as radio buttons then the Radiobutton widget will be used. You can select one at a time.
Scrollbar	To scroll the window up and down the scrollbar widget in python will be used.
Text	The text widget mainly provides a multi-line text field to the user where usersand enter or edit the text and it is different from Entry.
Toplevel	The Toplevel widget is mainly used to provide us with a separate window container
SpinBox	The SpinBox acts as an entry to the "Entry widget" in which value can beinput just by selecting a fixed value of numbers.
PanedWindo w	The PanedWindow is also a container widget that is mainly used to handle different panes. Panes arranged inside it can either Horizontal or vertical
LabelFrame	The LabelFrame widget is also a container widget used to mainly handle the complex widgets.
MessageBox	The MessageBox widget is mainly used to display messages in the Desktop applications.

Label Widget

- The label widget is mainly used to provide a message about the other widgets used in the Python Application to the user.
- You can change or update the text inside the label widget anytime you want.
- This widget uses only one font at the time of displaying some text.
- You can perform other tasks like underline some part of the text and you can also span text to multiple lines.
- There are various options available to configure the text or the part of the text shown in the Label.
- The **syntax** of the label widget is given below, W =

  Label(master,options)

In the above syntax, the master parameter denotes **the parent window.** You can use many options to configure the text and these options are written as **comma-separated key-value pairs**.

Label Widget Options

- **Bd**-This option is used for the border width of the widget. Its default value is 2 pixels.
- **Bg**-This option is used for the background color of the widget.
- **Cursor**-This option is used to specify what type of cursor to show when the mouse is moved over the label. The default of this option is to use the standard cursor.
- **Fg**-This option is used to specify the foreground color of the text that is written inside the widget.
- **Font-** This option specifies the font type of text inside the label.
- **Height**-This option indicates the height of the widget

Button Widget

- The Button widget in Tkinter is mainly used to add a button in any GUI Application.
- In Python, while using the Tkinter button widget, we can easily modify the style of the button like adding a background colors to it, adjusting height and width of button, or the placement of the button, etc. very easily.

Button widget options

- **Activebackground**-This option indicates the background of the button atthe time when the mouse hovers the button.
- **Bd**-This option is used to represent the width of the border in pixels.
- **Bg**-This option is used to represent the background color of the button.
- **Command** The command option is used to set the function call which isscheduled at the time when the function is called.
- **Activeforeground**-This option mainly represents the font color of the button when the mouse hovers the button.
- **Fg**-This option represents the foreground color of the button.
- **Padx**-This option indicates the additional padding of the button in the horizontal direction.
- **Pady**-This option indicates the additional padding of the button in the vertical direction.

Check-button widget.

- It allows you to select multiple options or a single option at a time just by clicking the button corresponding to each option.
- For example, in a form, you see option to fill in your Gender, it hasoptions, Male, Female, Others, etc., and you can tick on any of the

options, that is a checkbox. We use the <input> tag in HTML, to createcheckbox

- It can either contain text or image. There are a number of options available to configure the Checkbutton widget as per your requirement.

w=checkbutton(master,option=value)

In the above syntax, the master parameter denotes the parent window. You canuse many options to configure your checkbutton widget and these options are written as comma-separated key-value pair.

Checkbutton Widget Options

- **Font**- This option indicates the font of the checkbutton.
- **Height**-This option indicates the height of the button. This height indicates the number of text lines in the case of text lines and it indicates the number of pixels in the case of images. the default value is 1.
- **Image**-This option indicates the image representing the checkbutton.
- **Cursor**-This option helps in changing the mouse pointer to the cursor name when it is over the checkbutton.
- **Bisableforeground**-This option is the color which is used to indicate the text of a disabled checkbutton.
- **Higlightcolor**-This option indicates the highlight color when there is afocus on the checkbutton
- **Justify**-This option is used to indicate the way by which the multiple text lines are represented. For left justification, it is set to LEFT and it is set toRIGHT for the right justification, and CENTER for the center justification.

Checkbutton Widget Methods:

- **Invoke()**-This method in checkbutton widget is used to invoke the method associated with the checkbutton.

- **Select()**-This method in the checkbutton widget is called to turn on the checkbutton.
- **Deselect()**-This method in the checkbutton widget is called to turn off the checkbutton.
- **Toggle()**-This method in the checkbutton widget is used to toggle between the different Checkbuttons.
- **Flash()**-This method in the checkbutton widget is used to flashed between active and normal colors.

Radiobutton Widget

Tkinter radiobutton widget is used to implement multiple-choice options that are mainly created in user input forms.

- Allows the user to select only one option from the given ones. Thus it is also known as implementing one-of-many selection in a Python Application.
- Also, different methods can also be associated with radiobutton.
- You can also display multiple line text and images on the radiobutton. Each
- radiobutton displays a single value for a particular variable.
- You can also keep a track of the user's selection of the radiobutton because it is associated with a single variable

syntax of the Radiobutton

W = Radiobutton(master, options)

In the above syntax, the master parameter denotes the parent window. You can use many options to change the look of the radiobutton and these options are written as comma-separated key-value pairs.

Radiobutton Widget Options:

- **Bd**-This option is used to represent the width of the border in pixels.

- **Bg**-This option is used to represent the background color of the button.
- **Command** The command option is used to set the function call which is scheduled at the time when the function is called.
- **Activeforeground**-This option mainly represents the font color of the button when the mouse hovers the button.
- **Fg**-This option represents the foreground color of the button.
- **Padx**-This option indicates the additional padding of the button in the horizontal direction.
- **Pady**-This option indicates the additional padding of the button in the vertical direction.

Radiobutton Widget Methods:

- **Deselect()**-This method is used to deselect or turns off the radio button
- **Select()**-This method is used to select the radio button
- **Invoke()**-This method is generally used to call a function when the state of radio button gets changed.
- **Flash()**-This method is generally used to flash the radio button between its normal and active colors many times.

Menu Widget

- The following types of menus can be created using the Tkinter Menu widget: pop-up, pull-down, and top level.
- Top-level menus are those menus that are displayed just under the title bar of the root or any other top-level windows. For example, all the websites have a top navigation menu just below the URL bar in the browser.
- Menus are commonly used to provide convenient access to options like opening any file, quitting any task, and manipulating data in anapplication.

W = Menu(master, options)

In the above syntax, the master parameter denotes the parent window. You canuse many options to change the look of the menu and these options are written as comma-separated key-value pairs**.**

<u>Menu Widget Options:</u>

- **Cursor**-This option will convert the mouse pointer to the specified cursortype and it can be set to an arrow, dot, etc.
- **Font**-This option is used to represent the font type of the text of the widget.
- **Fg**-This option is used to represent the foreground color of the text of the widget.
- **Height**-This option indicates the vertical dimension of the widget
- **Width**-This option indicates the horizontal dimension of the widget and itis represented as the number of characters.
- **Padx**-This option represents the horizontal padding of the widget.
- **Pady**-This option represents the vertical padding of the widget

<u>Menu Widget Methods:</u>

- **add_command()**- This method is used to add menu items to the menu.
- **add_radiobutton()**-This method is used to add the radiobutton to themenu.
- **add_checkbutton()**-This method is mainly used to add checkbuttons tothe menu.
- **add_cascade()**-This method is used to create a hierarchical menu to the parent menu by associating the given menu to the parent menu.
- **add_seperator()**-This method is used to add the separator line to themenu items.

Frame Widget

The Tkinter Frame widget is used to group and organize the widgets in a better and friendly way.

The Frame widget is basically a container (an invisible container) whose task is to hold other widgets and arrange them with respect to each other.

syntax

W = Frame(master, options)

In the above syntax, the master parameter denotes the parent window. You canuse many options to change the look of the frame and these options are written as comma-separated key-value pairs**.**

Frame Widget options

- **bd**-This option is used to represent the width of the border. Its default value is 2 pixels.
- **Bg**-This option is used to indicate the normal background color of awidget.
- **Cursor**-With the help of this option, the mouse pointer can be changed tothe cursor type which is set to different values like an arrow, dot, etc.
- **Height**-This option is used to indicate the height of the frame.
- **Width**-This option is used to indicate the width of the frame.
- **Highlightbackground**-This option denotes the color of the background color when it is under focus.

Canvas Widget

- Tkinter Canvas widget is mainly used as a general-purpose widget which is used to draw anything on the application window in Tkinter.

- This widget is mainly used to draw graphics and plots, drawings, charts, and showing images
- You can draw several complex layouts with the help of canvas, for example, polygon, rectangle, oval, text, arc bitmap, graphics, etc.
- Canvas widget is also used to create graphical editors.

Syntax

w = Canvas(master, option=value)

In the above syntax, the master parameter denotes the parent window. You canuse many options to change the layout of the canvas and these options are written as comma-separated key-values.

Canvas Widget Options:

- **Bg**-This option is used to set the background color.
- **Cursor**-Whether to use an arrow, dot, or circle on the canvas for thecursor, this option can be used.
- **Confine**-This option is set to make the canvas non-scrollable outside thescroll region.
- **Height**-This option is used for controlling the height of the canvas.
- **Width**-This option is used to set the width of the widget.
- **Highlightcolor**-This option indicates the highlight color when there is afocus on the button
- **Scrollregion**-This is option is mainly used to represent the coordinates that are specified as the tuple containing the area of the canvas
- **Xscrollincrement**-If the value of this option is set to a positive value then, the canvas is placed only to the multiple of this value.
- **Yscrollincrement**-It is mainly used for vertical movement and it worksin the same way xscrollincrement option works.

<u>Python Tkinter Text Widget</u>

- The text widget is used to provide a multiline textbox (input box) becausein Tkinter single-line textbox is provided using Entry widget.

- You can use various styles and attributes with the Text widget.
- You can also use marks and tabs in the Text widget to locate the specific sections of the text.
- Media files like images and links can also be inserted in the Text Widget.
- There are some variety of applications where you need multiline text like sending messages or taking long inputs from users, or to show editable long format text content in application, etc. use cases are fulfilled by this widget.
- Thus in order to show textual information, we will use the Text widget.

syntax

W = Text(master, options)

- In the above syntax, the master parameter denotes the parent window.

  You can use many options to configure the text editor and these options are written as comma-separated key-value pairs.

Text Widget Options:

- **Exportselection**-This option is used to export the selected text in the selection of the window manager. If you do not want to export the text then you can set the value of this option to 0.
- **Highlightbackground**-This option indicates the highlightcolor at the time when the widget isn't under the focus.
- **Higlightthickness**-This option is used to indicate the thickness of the highlight. The default value of this option is 1.
- **Highlightcolor**-This option indicates the color of the focus highlight when the widget is under the focus.

- **Insertbackground**-This option is used to represent the color of the insertion cursor.
- **Spacing1**-This option indicates the vertical space to insert above each line of the text.
- **Spacing2**-This option is used to specify how much extra vertical space to add between displayed lines of text when a logical line wraps. The default value of this option is 0
- **Spacing3**-This option indicates the vertical space to insert below each line of the text.
- **Insertofftime**-This option represents the time amount in Milliseconds and during this time the insertion cursor is off in the blink cycle
- **Insertontime**-This option represents the time amount in Milliseconds and during this time the insertion cursor is on in the blink cycle

Text Widget Methods:

- **Index(index)**-This method is used to get the specified index.
- **See(index)**-This method returns true or false on the basis that if the string is visible or not at the specified index.
- **insert(index,string)**-This method is used to insert a string at the specified index.
- **get(startindex,endindex)**-This method returns the characters in the specified range
- **delete(startindex,endindex)**-This method deletes the characters in the specified range

Listbox Widget

- The items contain the same type of font and the same font color.

- It is important to note here that only text items can be placed inside a Listbox widget.
- From this list of items, the user can select one or more items according to the requirements.

Syntax

W = Listbox(master, options)

- In the above syntax, the master parameter denotes the parent window. You can use many options to change the look of the ListBox and these options are written as comma-separated key-value pairs.

Listbox Widget Options:

- **Fg-**This option indicates the color of the text.
- **Height-**This option is used to represents the count of the lines shown in the Listbox. The default value of this option is 10.
- **Highlightcolor-**This option is used to indicate the color of the Listbox items when the widget is under focus.
- **Highlightthickness-**This option is used to indicate the thickness of the highlight.
- **Selectbackground-**This option is used to indicate the background color that is used to display the selected text.
- **Selectmode-**This option is used to determine the number of items that can be placed from the list. It can set to browse, single, multiple, extended.
- **Xscrollcommand-**This option is used to let the user scroll the Listbox horizontally.
- **Yscrollcommand-**This option is used to let the user scroll the Listbox vertically.

ListBox Widget Methods:

- **activate(index)**-This method is mainly used to select the lines at thespecified index.
- **curselection()**-This method is used to return a tuple containing the line numbers of the selected element or elements, counting from 0. If nothing is selected, return an empty tuple.
- **delete(first, last = None)**-This method is used to delete the lines which exist in the given range.
- **get(first, last = None)**-This method is used to get the list of items thatexist in the given range.
- **index(i)**-This method is used to place the line with the specified index atthe top of the widget.
- **insert(index, *elements)**-This method is used to insert the new lines with the specified number of elements before the specified index.

PYTHON SQLITE

- SQLite is embedded relational database management system. It is self-contained, serverless, zero configuration and transactional SQL database engine.
- SQLite is free to use for any purpose commercial or private. In other words, "SQLite is an open source, zero-configuration, self-contained, stand alone, transaction relational database engine designed to be embedded into an application".
- SQLite is different from other SQL databases because unlike most other SQL databases, SQLite does not have a separate server process. It reads and writes directly to ordinary disk files. A complete SQL database with multiple tables, indices, triggers, and views, is contained in a single disk file.

SQLITE FEATURES

Following is a list of features which makes SQLite popular among other lightweight databases:

- **SQLite is totally free:** SQLite is open-source. So, no license is required to work with it.
- **SQLite is serverless:** SQLite doesn't require a different server process or system to operate.
- **SQLite is very flexible:** It facilitates you to work on multiple databases on the same session at the same time.

- **Configuration Not Required:** SQLite doesn't require configuration. No setup or administration required.
- **SQLite is a cross-platform DBMS:** You don't need a large range of different platforms like Windows, Mac OS, Linux, and Unix. It can also be used on a lot of embedded operating systems like Symbian, and Windows CE.
- **Storing data is easy:** SQLite provides an efficient way to store data.
- **Provide large number of API's**: SQLite provides API for a large range of programming languages. For example: .Net languages (Visual Basic, C#), PHP, Java, Objective C, Python and a lot of other programming language.

SQLITE 3 MODULE

- The SQLite3 module in Python provides a simple and convenient way to interact with SQLite databases.
- It allows you to perform various database operations, such as creating tables, inserting data, querying data, updating data, and deleting data.
- Here's an overview of the SQLite3 module and its functionality:

Importing the module:

To use the SQLite3 module, you need to import it in your Python script:

```
import sqlite3
```

Connecting to a database:

To establish a connection to an SQLite database, you can use the connect() function provided by the SQLite3 module. It takes the database file path as a parameter and returns a Connection object that represents the connection to the database.

```
connection = sqlite3.connect('database.db')
```

Creating a cursor:

After establishing a connection, you need to create a Cursor object to execute SQL statements and interact with the database. The Cursor object allows you to execute queries, fetch results, and perform database operations.

```
cursor = connection.cursor()
```

Executing SQL statements:

You can execute SQL statements using the execute() method of the Cursor object. It takes an SQL query as a parameter and executes it against the database.

```
cursor.execute("CREATE TABLE employees (id INTEGER PRIMARY KEY, name
TEXT, age INTEGER)")
```

Committing changes:

After executing SQL statements that modify the database, such as inserting, updating, or deleting data, you need to commit the changes to make them permanent using the commit() method of the Connection object.

```
connection.commit()
```

Querying data:

To retrieve data from the database, you can execute a SELECT query using the execute() method and then fetch the results using methods like fetchone() (to fetch a single row) or fetchall() (to fetch all rows).

```
cursor.execute("SELECT * FROM employees")
rows = cursor.fetchall()
for row in rows:
    print(row)
```

Closing the connection:

After you're done with the database operations, it's important to close the connection using the close() method of the Connection object.

connection.close()

These are the basic steps involved in using the SQLite3 module to interact with an SQLite database in Python.

SQLITE METHODS

- **Import sqlite3 module**
- import sqlite3 statement in the program.
- Using the classes and methods defined in the sqlite3 module we can communicate with the SQLite database.
- **Use the connect() method**
- Use the connect() method of the connector class with the database name.
- To establish a connection to SQLite, you need to pass the database name you want to connect.
- If you specify the database file name that already presents on the disk, it will connect to it.
- But if your specified SQLite database file doesn't exist, SQLite creates a new database for you.
- This method returns the SQLite Connection Object if the connection is successful.
- **Use the cursor() method**
- Use the cursor() method of a connection class to create a cursor object to execute SQLite command/queries from Python.
- **Use the execute() method**
- The execute() methods run the SQL query and return the result.
- **Extract result using fetchall()**
- Use cursor.fetchall() or fetchone() or fetchmany() to read query result.
- **Close() method**
- After you're done with the database operations, it's important to close the connection using the close() method

CONNECT TO SQLITE DATABASE

- The first thing to do is create a database and connect to it:

```
import sqlite3

dbName = 'database.db'

try:

  conn = sqlite3.connect(dbName)

  cursor = conn.cursor()

  print("Database created!")

except Exception as e:

  print("Something bad happened: ", e)

  if conn:

    conn.close()
```

- On line 1, we import the sqlite3 library.
- Then, inside a try/except code block, we call sqlite3.connect() to initialize aconnection to the database.
- If everything goes right, conn will be an instance of the Connection object.
- If the try fails, we print the exception received and the connection to the database is closed.
- As stated in the official documentation, each open SQLite database is represented bya Connection object.
- Each time we have to execute an SQL command, the Connection object has a method called cursor().
- In database technologies, a cursor is a control structure that enables traversal over the records in a database.
- Now, if we execute this code we should get the following output:

```
Database created!
```

- If we look at the folder where our Python script is, we should see a new file called database.db. This file has been created automatically by sqlite3

<u>Close cursor and connection objects</u>

- use cursor.clsoe() and connection.clsoe() method to close the cursor and SQLite connections after your work completes

```
import sqlite3

try:

    sqliteConnection = sqlite3.connect('SQLite_Python.db')
    cursor = sqliteConnection.cursor()
    print("Database created and Successfully Connected to SQLite")
    sqlite_select_Query = "select sqlite_version();"
    cursor.execute(sqlite_select_Query)
    record = cursor.fetchall()
    print("SQLite Database Version is: ", record)
    cursor.close()
  except sqlite3.Error as error:
     print("Error while connecting to sqlite", error)
 finally:
    if sqliteConnection:
        sqliteConnection.close()
        print("The SQLite connection is closed")
```

OPERATIONS ON TABLE

➢ **Create Table**

To create a table in SQLite, you use the CREATE TABLE statement.

Syntax

```
CREATE TABLE table_name (
    column1 datatype,
    column2 datatype,
    ...
);
```

Example:

```
CREATE TABLE employees (
    id INTEGER PRIMARY KEY,
    name TEXT,
    age INTEGER,
    city TEXT
);
```

- **Insert**

To insert data into a table, you use the INSERT INTO statement.

Syntax

```
INSERT INTO table_name (column1, column2, ...)
VALUES (value1, value2, ...);
```

Example:

```
INSERT INTO employees (name, age, city)
VALUES ('John', 25, 'New York');
```

- **Update**

To update existing data in a table, you use the UPDATE statement.

Syntax

```
UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;
```

Here's an example that updates the age of an employee with a specific ID

```
UPDATE employees
SET age = 30
WHERE id = 1;
```

- **Delete**

To delete data from a table, you use the DELETE FROM statement.

Syntax

```
DELETE FROM table_name
WHERE condition;
```

Here's an example that deletes an employee with a specific ID

```
DELETE FROM employees
```

```
WHERE id = 1;
```

- **Querying data**

To retrieve data from a table, you use the SELECT statement.

Syntax

```
SELECT column1, column2, ...

FROM table_name

WHERE condition;
```

Here's an example that selects all employees

```
SELECT * FROM employees;
```

- **Filtering data**

You can use the WHERE clause to filter data based on specific conditions.

Syntax

```
SELECT column1, column2, ...

FROM table_name

WHERE condition;
```

Here's an example that selects employees from a specific city

```
SELECT * FROM employees

WHERE city = 'New York';
```

- **Sorting data**

You can use the ORDER BY clause to sort the data in a specific order.

Syntax

```
SELECT column1, column2, ...

FROM table_name

ORDER BY column_name [ASC|DESC];
```

Here's an example that selects employees sorted by age in ascending order

```
SELECT * FROM employees

ORDER BY age ASC;
```

NUMPY

- NumPy is a powerful Python library for numerical computing that provides support for efficient operations on large, multi-dimensional arrays and matrices.
- The name "NumPy" is short for "Numerical Python." It is widely used in scientific computing, data analysis, and machine learning due to its performance, versatility, and extensive set of mathematical functions.

Some key features of NumPy

Arrays:

- The fundamental data structure in NumPy is the nd array, which stands for N-dimensional array.
- It is a homogeneous collection of elements with a fixed size in memory.
- Arrays can have one or more dimensions and can store data of different types, such as integers, floating-point numbers, or even complex numbers.

Vectorized operations:

- NumPy enables efficient and concise numerical operations on arrays through vectorization.
- Vectorization eliminates the need for explicit loops in many cases, leading to faster computations.
- It allows you to perform element-wise operations, mathematical functions, and linear algebra operations on entire arrays or subsets of arrays.

Array creation:

- NumPy provides various functions for creating arrays, such as numpy.array(), numpy.zeros(), numpy.ones(), numpy.arange(), and numpy.random. These functions allow you to create arrays with specific shapes, initialize them with specific values, or generate random numbers.

Indexing and slicing:

- NumPy provides powerful indexing and slicing capabilities to access and manipulate elements within arrays.
- You can use integers, slices, to extract specific elements or subsets of arrays based on conditions.

Broadcasting:

- Broadcasting is a powerful feature in NumPy that allows operations between arrays of different shapes and sizes.
- It simplifies calculations by automatically expanding arrays to compatible shapes, avoiding the need for explicit looping or unnecessary memory duplication.

Mathematical functions:

- NumPy offers a wide range of mathematical functions, including basic arithmetic operations, trigonometric functions, exponential functions, logarithmic functions, and more.
- These functions operate element-wise on arrays, making it convenient for numerical computations.

ARRAAY CREATION USING NUMPY

NumPy provides several functions for creating arrays with different properties and initial values.

Here are some commonly used methods for array creation:

- **numpy.array:** Creates an array from a Python list or tuple.

```
import numpy as np

arr1 = np.array([1, 2, 3, 4, 5])

print(arr1)
```

Output: [1 2 3 4 5]

```
arr2 = np.array([[1, 2, 3], [4, 5, 6]])

print(arr2)
```

Output:

```
 [[1 2 3]

[4 5 6]]
```

- **numpy.zeros:** Creates an array filled with zeros.

```
arr3 = np.zeros((3, 4))
```

```
print(arr3)
```

Output:

```
[[0. 0. 0. 0.]
[0. 0. 0. 0.]
 [0. 0. 0. 0.]]
```

- **numpy.ones:** Creates an array filled with ones.

```
arr4 = np.ones((2, 3)) # shape as argument
print(arr4)
```

Output:

```
[[1. 1. 1.]
[1. 1. 1.]]
```

- **numpy.arange:** Creates an array with regularly spaced values.

```
arr5 = np.arange(0, 10, 2) # start, stop, step as arguments
print(arr5)
```

Output:

```
 [0 2 4 6 8]
```

- **numpy.linspace:** Creates an array with a specified number of evenly spaced values between a start and end point.

```
arr6 = np.linspace(0, 1, 5) # start, end, number of values as arguments
print(arr6)
```

Output:

```
[0.  0.25 0.5  0.75 1.  ]
```

- **numpy.random:** Generates arrays with random values.

```
arr7 = np.random.rand(3, 2) # shape as argument for random values between 0 and 1
print(arr7)
```

Output:

```
[[0.56935657 0.69338216]
[0.12691842 0.50071383]
```

[0.14643409 0.9199429]]

arr8 = np.random.randint(1, 10, (2, 3)) # low, high, shape as arguments for random integers between 1 and 10

print(arr8)

Output:

[[7 2 1]

[9 8 5]]

OPERATIONS ON ARRAY USING NUMPY

NumPy provides a wide range of operations for manipulating arrays efficiently. Here are some common operations you can perform on arrays using NumPy:

- **Arithmetic operations:** You can perform element-wise arithmetic operations such as addition, subtraction, multiplication, division, and exponentiation on arrays.

import numpy as np

arr1 = np.array([1, 2, 3, 4])

arr2 = np.array([5, 6, 7, 8])

i)arr_sum = arr1 + arr2

print(arr_sum)

Output: [6 8 10 12]

ii)arr_diff = arr2 - arr1

print(arr_diff)

Output: [4 4 4 4]

iii)arr_prod = arr1 * arr2

print(arr_prod)

Output: [5 12 21 32]

iv)arr_div = arr2 / arr1

print(arr_div)

Output: [5.   3.   2.33333333 2.   ]

v)arr_pow = arr1 ** 2

print(arr_pow)

Output: [1 4 9 16]

- **Aggregation functions:** NumPy provides functions to compute statistical measures on arrays, such as sum, mean, median, standard deviation, minimum, maximum, etc.

arr = np.array([1, 2, 3, 4, 5])

i)arr_sum = np.sum(arr)

print(arr_sum)

Output: 15

ii)arr_mean = np.mean(arr)

print(arr_mean)

Output: 3.0

iii)arr_median = np.median(arr)

print(arr_median)

Output: 3.0

iv)arr_std = np.std(arr)

print(arr_std)

Output: 1.4142135623730951

v)arr_min = np.min(arr)

print(arr_min)

Output: 1

vi)arr_max = np.max(arr)

print(arr_max)

Output: 5

- **Array manipulation:** NumPy provides various functions for reshaping, transposing, and concatenating arrays.

arr = np.array([[1, 2, 3], [4, 5, 6]])

i)arr_reshaped = np.reshape(arr, (3, 2))

print(arr_reshaped)

Output:

```
 [[1 2]

 [3 4]

[5 6]]
```

ii)arr_transposed = np.transpose(arr)

print(arr_transposed)

Output:

```
[[1 4]

[2 5]

[3 6]]
```

iii)arr_concatenated = np.concatenate((arr, arr), axis=0)

print(arr_concatenated)

Output:

```
 [[1 2 3]

[4 5 6]
[1 2 3]
[4 5 6]]
```

- **Mathematical functions:** NumPy provides a wide range of mathematical functions that operate element-wise on arrays.

```
arr = np.array([1, 2, 3, 4])
```

i)arr_exp = np.exp(arr)

print(arr_exp)

Output: [2.71828183 7.3890561 20.08553692 54.59815003]

ii)arr_sqrt = np.sqrt(arr)

print(arr_sqrt)

Output: [1.   1.41421356 1.73205081 2.   ]

iii)arr_sin = np.sin(arr)

print(arr_sin)

Output: [0.84147098 0.90929743 0.14112001 -0.7568025]

INTRODUCTION TO PANDAS

- Pandas is a powerful and popular open-source Python library widely used for data manipulation and analysis.
- It provides easy-to-use data structures and data analysis tools, making it a go-to library for working with structured and tabular data.
- Pandas is built on top of NumPy, extending its capabilities with additional functionality specifically designed for data manipulation tasks.

Some key features and concepts in Pandas

Data structures: Pandas introduces two primary data structures, namely Series and DataFrame.

i. **Series:** A Series is a one-dimensional labelled array that can hold any data type. It is similar to a column in a spread sheet or a single column of data in a NumPy array, with associated index labels for each element.

ii. **DataFrame:** A DataFrame is a two-dimensional labelled data structure, resembling a table or a spread sheet with rows and columns. It is composed of multiple Series objects that share a common index, allowing efficient data alignment. DataFrames provide a convenient way to store, manipulate, and analyze tabular data.

iii. **Data manipulation:** Pandas provides a rich set of functions for data manipulation tasks, such as filtering, sorting, merging, grouping, reshaping, and aggregating data.

iv. **Selection and filtering:** You can select subsets of data from a DataFrame using various indexing techniques, such as label-based indexing, integer-based indexing, or Boolean indexing based on specific conditions.

v. **Data cleaning and preprocessing:** Pandas offers methods to handle missing data, perform data imputation, remove duplicates, perform string operations, and convert data types. These features are crucial for data cleaning and preprocessing tasks.

vi. **Merging and joining:** Pandas allows you to merge or join multiple DataFrames based on common columns or indices, enabling the combination of different datasets into a single dataset.

vii. **Grouping and aggregating:** Pandas provides powerful tools for grouping data based on one or more columns, and then performing aggregations or calculations on these groups. This functionality is useful for performing data summarization and generating insights from grouped data.

viii. **Data input and output:** Pandas supports reading and writing data in various file formats, including CSV, Excel, SQL databases, and more. It simplifies the process of loading data into a DataFrame and saving the modified data back to different file formats.

ix. **Integration with other libraries:** Pandas integrates well with other libraries in the Python data science ecosystem, such as NumPy, Matplotlib, and scikit-learn.

SERIES IN PANDAS

- In Pandas, a Series is a one-dimensional labelled array-like data structure that can hold any data type.
- It is similar to a column in a spread sheet or a NumPy array, but with additional capabilities and features.
- Each element in a Series is associated with a unique label, called the index, which allows for efficient data alignment and easy access to values.

The basic **syntax to create a Series object in Pandas** is as follows:

```
import pandas as pd

s = pd.Series(data, index)
```

- Here, data can be a list, NumPy array, dictionary, or scalar value that represents the data you want to store in the Series.
- index is an optional parameter that specifies the labels for each element in the Series. If not provided, a default integer index starting from 0 is assigned.

Example:- creating a Series:

```
import pandas as pd

data = [10, 20, 30, 40, 50]

s = pd.Series(data)

print(s)
```

Output:

```
0   10

1   20

2   30

3   40

4   50
```

In this example, a Series is created with the provided data list, and the default integer index is assigned. The resulting Series is displayed, showing the elements along with their corresponding index.

SERIES OBJECTS PROVIDE SEVERAL USEFUL FUNCTIONALITIES AND OPERATIONS:

- **Indexing and slicing:** You can access elements of a Series using the associated index labels. Slicing can also be performed to select subsets of the Series.
- **Arithmetic operations:** Series support element-wise arithmetic operations, such as addition, subtraction, multiplication, and division, as well as mathematical functions from NumPy.
- **Alignment:** Series objects align data based on their index labels, allowing for effortless operations between Series with different lengths or indexes. Missing values are introduced as NaN (Not a Number) during alignment.
- **Handling missing data:** Pandas provides methods to handle missing data in Series, such as isnull(), notnull(), and fillna().
- **Data alignment and automatic labeling:** When performing operations involving multiple Series or combining Series into a DataFrame, Pandas aligns the data based on index labels. It automatically labels the resulting data to ensure proper alignment.
- **Name attribute:** Series can have a name attribute, which helps identify the Series when it becomes part of a DataFrame or during data analysis.

DATAFRAME

- In Pandas, a DataFrame is a two-dimensional labelled data structure that represents tabular data, similar to a table or a spreadsheet.
- It consists of rows and columns, where each column can hold data of different types (e.g., integers, floats, strings) and is referred to as a Series.
- The DataFrame provides a flexible and efficient way to handle and manipulate structured data.
- The DataFrame provides powerful indexing and alignment capabilities, making it easy to perform data manipulations, analysis, and transformations.

- To create a DataFrame in Pandas, you can use various methods. One common way is by passing a dictionary of lists, arrays, or Series as input, where the keys of the dictionary represent column names, and the values represent the data in each column.
  Example:

```
import pandas as pd

data = {

    'Name': ['John', 'Jane', 'Mike', 'Lisa'],

    'Age': [25, 30, 28, 35],

    'City': ['New York', 'London', 'Paris', 'Tokyo']

}

df = pd.DataFrame(data)

print(df)
```

Output:

```
   Name  Age      City

0  John  25  New York

1  Jane   30   London

2  Mike   28    Paris

3  Lisa   35    Tokyo
```

- In this example, a DataFrame is created from the dictionary data, where each key represents a column name and the corresponding value represents the data in that column. The resulting DataFrame is displayed, showing the columns and their respective data.

DataFrames provide several useful functionalities and operations:

Indexing and selection: You can access specific rows, columns, or subsets of data using various indexing techniques, such as label-based indexing, integer-based indexing, or Boolean indexing.

Column and row operations: DataFrames support operations on columns, such as adding new columns, modifying existing columns, or dropping columns. Rows can be added, deleted, or modified using various methods.

Data alignment: Similar to Series, DataFrames align data based on their index labels when performing operations involving multiple DataFrames or combining DataFrames with different shapes.

Handling missing data: Pandas provides methods to handle missing data in DataFrames, such as isnull(), notnull(), dropna(), and fillna().

Data aggregation and groupby: DataFrames support various operations for aggregating data, including grouping data based on specific columns, performing calculations within groups, and applying functions across groups.

Data input and output: Pandas offers functions to read and write data in different formats, such as CSV, Excel, SQL databases, and more. It simplifies the process of loading data into a DataFrame and saving modified data back to different file formats.

CREATING DATAFRAMES FROM EXCEL SHEETS

- Pandas provides convenient functions to read data from Excel files and create DataFrames. Here's an example of how to create a DataFrame from an Excel sheet using Pandas:

```
import pandas as pd

df = pd.read_excel('data.xlsx', sheet_name='Sheet1')

print(df)
```

- In this example, the read_excel() function is used to read the data from the Excel file named data.xlsx.
- You need to specify the sheet name using the sheet_name parameter (e.g., 'Sheet1') to read data from a specific sheet.
- If the Excel file contains multiple sheets, you can also read all the sheets by omitting the sheet_name parameter or passing None.
- The resulting data from the Excel sheet is stored in the DataFrame df.
- You can then perform various operations on the DataFrame, such as indexing, filtering, and data manipulation.
- If the Excel file contains multiple sheets and you want to read all the sheets into separate DataFrames, you can use the read_excel() function with sheet_name=None.

- It will return a dictionary of DataFrames, where each key represents the sheet name and the corresponding value is the DataFrame containing the sheet data.

```
import pandas as pd

dfs = pd.read_excel('data.xlsx', sheet_name=None)

# Access individual DataFrames

df_sheet1 = dfs['Sheet1']

df_sheet2 = dfs['Sheet2']

# Display the DataFrames

print(df_sheet1)

print(df_sheet2)
```

- In this case, the read_excel() function reads all the sheets from the Excel file, and the resulting dictionary of DataFrames is stored in the variable dfs.
- You can access individual DataFrames by providing the sheet name as the key to the dictionary.
- By utilizing the read_excel() function, you can easily read data from Excel files and create corresponding DataFrames, allowing you to perform various data analysis and manipulation tasks using Pandas.

CREATING DATAFRAMES FROM .CSV FILES

Creating a DataFrame from a CSV (Comma-Separated Values) file is straightforward using Pandas. The read_csv() function in Pandas allows you to read data from a CSV file and create a DataFrame.

Example:

```
import pandas as pd

df = pd.read_csv('data.csv')

print(df)
```

- In this example, the read_csv() function reads the data from the CSV file named data.csv.

- The resulting data is stored in the DataFrame df. By default, the read_csv() function assumes that the CSV file has a header row, which contains column names.
- If the CSV file doesn't have a header row, you can specify header=None as an argument to the function.
- You can also customize various parameters of the read_csv() function to handle specific CSV file formats or requirements.

  Some common parameters include:

sep: Specifies the delimiter used in the CSV file. By default, it is a comma (,). You can change it to a different character or string if your CSV file uses a different delimiter.

header: Specifies the row number(s) to use as column names. By default, it is 0 (the first row). If the CSV file doesn't have a header row, you can set header=None and provide your own column names later.

index_col: Specifies the column(s) to use as the index of the DataFrame. It can take either a column name or column index.

usecols: Specifies the columns to read from the CSV file. You can provide a list of column names or column indices to select specific columns.

dtype: Specifies the data type for specific columns. It can be a dictionary where the keys are column names, and the values are the desired data types.

parse_dates: Specifies the columns to parse as dates. It can be a list of column names or column indices.

Here's an example that demonstrates using some of these parameters:

```
import pandas as pd

# Read CSV file with specific parameters

df = pd.read_csv('data.csv', sep=';', header=0, index_col='ID', usecols=['ID', 'Name', 'Age'],
dtype={'Age': int}, parse_dates=['Birthdate'])

# Display the DataFrame

print(df)
```

- In this example, the CSV file is read with a semicolon (;) as the delimiter, and the first row is used as the header row. The column named 'ID' is set as the index column. Only the columns 'ID', 'Name', and 'Age' are read from the CSV file. The 'Age' column is specified to have an integer data type, and the 'Birthdate' column is parsed as dates.

- By adjusting the parameters of the read_csv() function, you can handle various CSV file formats, specify column names and types, and customize the DataFrame creation according to your specific requirements.

CREATING DATAFRAMES FROM DICTIONARY

- Creating a DataFrame from a dictionary in Pandas is a common approach, especially when you have data already organized in a dictionary format.
- You can use the pd.DataFrame() function to convert the dictionary into a DataFrame.

Example:

```
import pandas as pd
data = {
   Name': ['Janavi', 'Emily', 'Ramya', 'Saranya'],
    'Age': [25, 30, 28, 32],
    'City': ['Mysuru', 'Bangalore', 'Mandya', 'Hassan']
}
df = pd.DataFrame(data, index=['A', 'B', 'C'])
print(df)
```

Output

```
   Name   Age        City
0  Janav    25      Mysuru
1  Emily    30      Bangalore
2  Ramya    28      Mandya
3  Saranya    32     Hassan
```

- In this example, the dictionary data contains three key-value pairs, where the keys represent the column names, and the corresponding values represent the data in each column.
- The pd.DataFrame() function is called with the dictionary as the argument, and the resulting DataFrame is stored in the variable df.
- The DataFrame is then displayed using the print() function.
- The DataFrame is created with three columns ('Name', 'Age', 'City'), and each column contains the data provided in the dictionary. The DataFrame automatically assigns a numeric index to each row starting from 0.

- If the dictionary contains lists or arrays of unequal lengths, Pandas will fill the missing values with NaN (Not a Number) to ensure a rectangular structure for the DataFrame.

CREATING DATAFRAMES FROM TUPLES

- Creating a DataFrame from tuples in Pandas can be done by passing a list of tuples to the pd.DataFrame() function.
- Each tuple represents a row of data, and the elements within the tuple correspond to the values in each column. Here's an example:

```
import pandas as pd
data = [
   ('John', 25, 'New York'),
   ('Jane', 30, 'London'),
   ('Mike', 28, 'Paris')]
df = pd.DataFrame(data, columns=['Name', 'Age', 'City'])
print(df)
```

- In this example, the list data contains three tuples, where each tuple represents a row of data.
- The elements within each tuple correspond to the values in each column.
- The pd.DataFrame() function is called with the list of tuples as the data argument, and the columns are specified using the columns parameter.

output:

```
  Name  Age     City
0  John  25  New York
1  Jane   30   London
2  Mike   28    Paris
```

OPERATIONS ON DATAFRAMES

- You can perform various operations on the DataFrame, such as filtering, grouping, and manipulation, using the powerful capabilities of Pandas.

- DataFrames in Pandas provide a wide range of operations and functionalities for manipulating and analyzing data. Here are some common operations you can perform on DataFrames

Accessing and viewing data:

- df.head(n): Returns the first n rows of the DataFrame.
- df.tail(n): Returns the last n rows of the DataFrame.
- df[column_name] or df.loc[:, column_name]: Accesses a specific column by its name.
- df.iloc[row_index] or df.iloc[row_index, column_index]: Accesses data by its index position.

Filtering data:

- df.query('condition'): Filters the DataFrame using a query string.
- df.loc[condition] or df.loc[condition, column_names]: Filters the DataFrame using a Boolean condition.

Adding, modifying, and deleting data:

- df['new_column'] = values: Adds a new column to the DataFrame with specified values.
- df['column'] = df['column'].map(function): Modifies values in a column using a function.
- df.drop('column_name', axis=1): Removes a column from the DataFrame.
- df.drop_duplicates(): Removes duplicate rows from the DataFrame.

Aggregation and summary statistics:

- df.describe(): Generates summary statistics of the DataFrame.
- df.groupby('column_name').aggregate(function): Groups data by a column and applies an aggregate function.
- df['column_name'].value_counts(): Calculates the count of unique values in a column.
- df['column_name'].mean(), df['column_name'].sum(),df['column_name'].max(), df['column_name'].min(): Calculates mean, sum, maximum, minimum, etc.

Sorting and reordering data:

- df.sort_values('column_name'): Sorts the DataFrame based on a specific column.

- df.sort_values(['column1', 'column2'], ascending=[True, False]): Sorts by multiple columns with different sort orders.
- df.sort_index(): Sorts the DataFrame by index.

Handling missing data:

- df.isnull(), df.notnull(): Checks for missing values in the DataFrame.
- df.dropna(): Drops rows with missing values.
- df.fillna(value): Fills missing values with a specific value.

Merging and joining DataFrames:

- pd.concat([df1, df2]): Concatenates DataFrames vertically.
- pd.merge(df1, df2, on='column_name'): Performs a database-style join based on a common column.
- df1.join(df2, on='column_name'): Joins two DataFrames based on a common index.

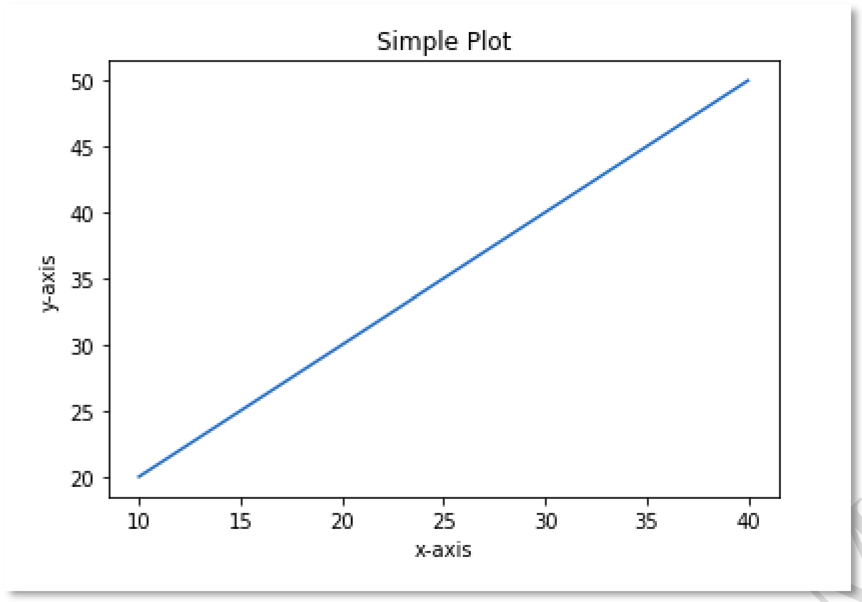
DATA VISUALIZATION

- Data visualization is an essential part of data analysis and exploration. It helps tounderstand patterns, trends, and relationships within the data by representing it visually.
- There are various libraries in Python that can be used for data visualization, such as Matplotlib, Seaborn, and Plotly. Here's a brief overview of these libraries and how you can use them for data visualization:

Matplotlib: Matplotlib is a widely used plotting library in Python. It provides a wide range of plots, including line plots, bar plots, scatter plots, histograms, and more. Here's a simple example of creating a line plot using Matplotlib

```
import matplotlib.pyplot as plt
# Sample data
x = [1, 2, 3, 4, 5]
y = [10, 20, 15, 25, 30]
# Creating a line plot
plt.plot(x, y)
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Line Plot')
plt.show()
```

Output:



Pyplot: Pyplot is a sub-module of the Matplotlib library, which is a popular data visualization library in Python. By importing the pyplot module from Matplotlib, you gain access to a wide range of functions and methods that allow you to create and manipulate figures, axes, and different types of charts.

Plotly: Plotly is a powerful library for interactive and web-based data visualization. It offers a wide range of plots, including line plots, bar plots, scatter plots, 3D plots, and more. Plotly allows you to create interactive plots that can be embedded in web applications or notebooks.

Here's an example of creating a bar plot using Plotly:

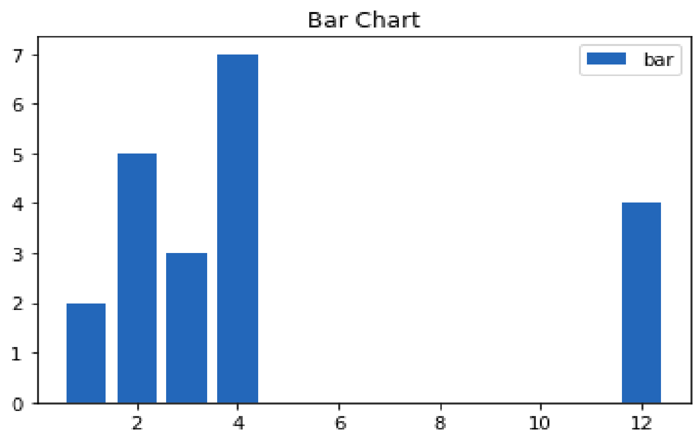
```
import plotly.express as px

# Sample data

data = {
    'Category': ['A', 'B', 'C', 'D'],
    'Value': [10, 20, 15, 25]
}

# Creating a bar plot
fig = px.bar(data, x='Category', y='Value', title='Bar Plot')
fig.show()
```

Output:



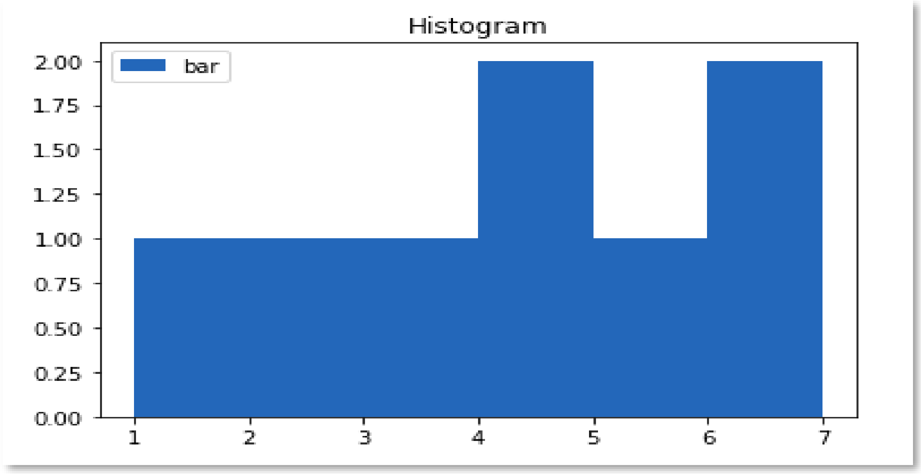
DIFFERENT TYPES OF CHARTS USING PYPLOT

HISTOGRAM

- A Histogram is a bar representation of data that varies over a range.
- It plots the height of the data belonging to a range along the y-axis and the range along the x-axis.
- Histograms are used to plot data over a range of values.
- They use a bar representation to show the data belonging to each range.

Example

```
import matplotlib.pyplot as plt
fig = plt.figure()
ax = fig.add_axes([0,0,1,1])
langs = ['C', 'C++', 'Java', 'Python', 'PHP']
students = [23,17,35,29,12]
ax.bar(langs,students)
plt.show()
```

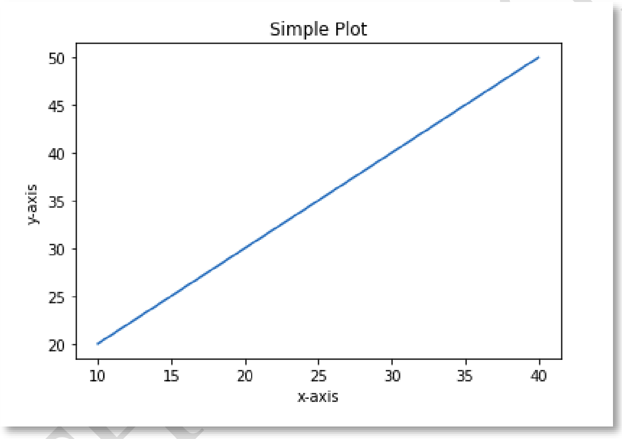


LINE CHART

- A Line chart is a graph that represents information as a series of data points connected by a straight line. In line charts, each data point or marker is plotted and connected with a line or curve.

Example

```
import matplotlib.pyplot as plt
x = [1, 2, 3, 4, 5]
y = [10, 15, 13, 18, 20]
plt.plot(x, y)
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Line Plot')
plt.show()
```

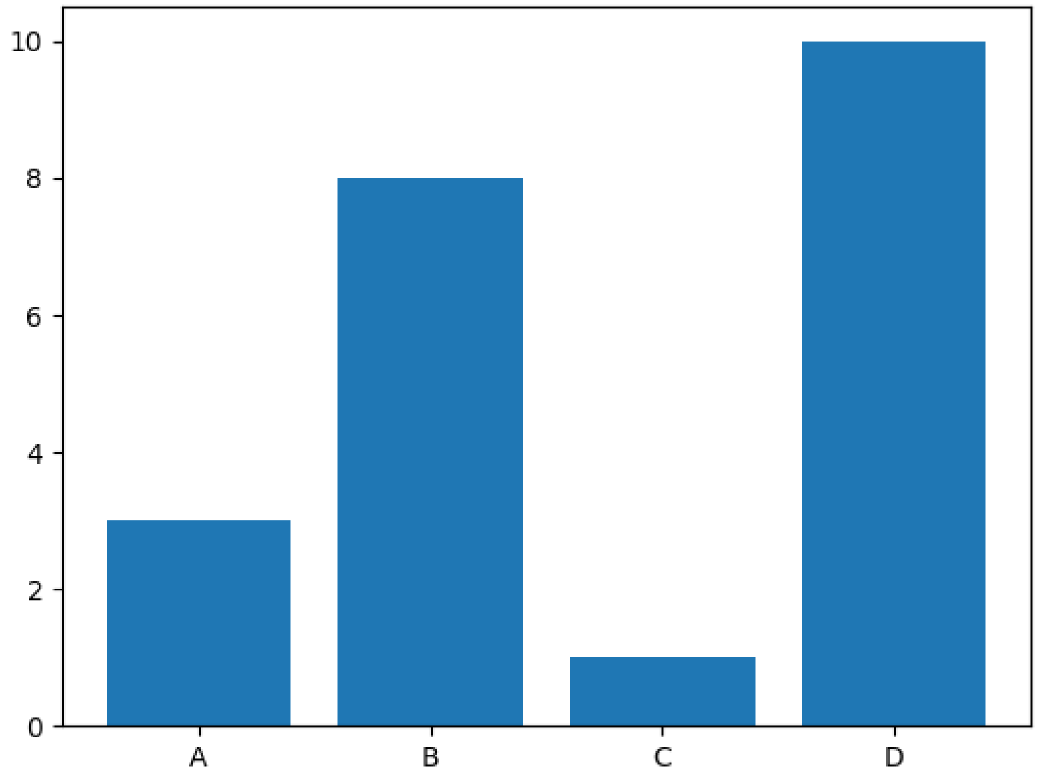


BAR GRAPHS

- When you have categorical data, you can represent it with a bar graph.
- A bar graph plots data with the help of bars, which represent value on the y-axis and category on the x-axis.

  Example

  ```
  import matplotlib.pyplot as plt
  x = ['A', 'B', 'C', 'D']
  y = [15, 8, 12, 10]
  plt.bar(x, y)
  plt.xlabel('Categories')
  plt.ylabel('Values')
  plt.title('Bar Plot')
  plt.show()
  ```

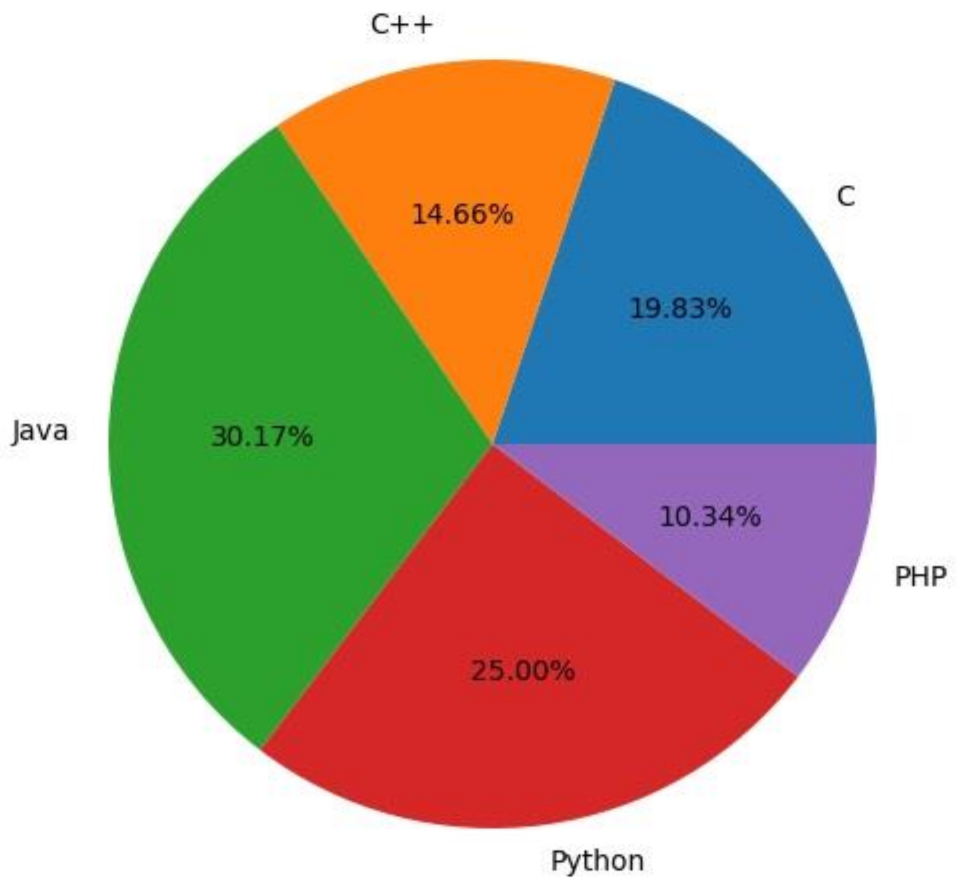


PIE CHART

- A pie chart represents data as sectors of a circle, where the size of each sector corresponds to the proportion or percentage it represents.
- Pie charts are commonly used to show the composition or relative contribution of different categories to a whole. They are effective in displaying categorical data and making comparisons between different categories.

Example

```
from matplotlib import pyplot as plt
import numpy as np
fig = plt.figure()
ax = fig.add_axes([0,0,1,1])
ax.axis('equal')
langs = ['C', 'C++', 'Java', 'Python', 'PHP']
students = [23,17,35,29,12]
ax.pie(students, labels = langs,autopct='%1.2f%%')
plt.show()
```


C++
C
Java
PHP
Python