1. **Define Python Pandas?**

Pandas is a Python library that provides data manipulation and analysis tools, particularly for tabular data (i.e., data organized in rows and columns). It is built on top of the NumPy library and provides high-performance, easy-to-use data structures such as Series (one-dimensional arrays), DataFrame (two-dimensional arrays), and Panel (three-dimensional arrays). Pandas is widely used for data analysis, cleaning, and preparation, and it offers functionalities for handling missing data, merging and grouping datasets, time series analysis, and much more. Pandas is an essential tool in the data science toolkit, and it is widely used in industries such as finance, economics, social sciences, and more.

1. **Mention different types of Data Structures in Panda?**

Pandas provides three main data structures for working with tabular data:

1. Series: A one-dimensional array-like object that can hold any data type, including numeric, string, boolean, etc. A series also has an associated array of labels, called an index.
2. DataFrame: A two-dimensional table-like data structure consisting of rows and columns. It is similar to a spreadsheet or SQL table, and it can store data of different types in each column.
3. Panel: A three-dimensional data structure that can store data in multiple tables, each of which is a DataFrame. A panel is less commonly used than a series or DataFrame.

In addition to these primary data structures, Pandas also provides several other data structures that are built on top of these, including TimeSeries, Categorical, and Sparse. These data structures offer more specialized functionality for working with specific types of data, such as time series or data with missing values.

3. **Explain different ways of creating Data Frames in Panda?**

There are several ways to create DataFrames in Pandas:

1. From a NumPy array: You can create a DataFrame by passing a 2D NumPy array to the DataFrame constructor. The columns and rows of the DataFrame will be labeled with default integer values.

import numpy as np

import pandas as pd

data = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

df = pd.DataFrame(data, columns=['a', 'b', 'c'])

1. From a Python dictionary: You can create a DataFrame by passing a dictionary to the DataFrame constructor. The keys of the dictionary will be used as the column labels, and the values will be used as the data.

data = {'a': [1, 4, 7], 'b': [2, 5, 8], 'c': [3, 6, 9]}

df = pd.DataFrame(data)

1. From a CSV file: You can create a DataFrame by reading in data from a CSV file using the **read\_csv()** method.

df = pd.read\_csv('file.csv')

1. From an Excel file: You can create a DataFrame by reading in data from an Excel file using the **read\_excel()** method.

df = pd.read\_excel('file.xlsx')

1. From a database: You can create a DataFrame by reading in data from a SQL database using the **read\_sql()** method.

import sqlite3

conn = sqlite3.connect('database.db')

df = pd.read\_sql('SELECT \* FROM table\_name', conn)

4**. Build a Numpy array filled with all zeros.**

You can create a NumPy array filled with zeros using the **numpy.zeros()** function. The function takes the shape of the array as input and returns an array of the specified shape filled with zeros. Here's an example:

import numpy as np

arr = np.zeros((3, 4))

print(arr)

Output:

[[0. 0. 0. 0.]

[0. 0. 0. 0.]

[0. 0. 0. 0.]]

In this example, we created a 2D array of zeros with shape **(3, 4)** and stored it in the variable **arr**. The resulting array is then printed using the **print()** function.

1. **Reverse a Numpy array.**

To reverse a NumPy array, you can use slicing with a step value of -1. Here's an example:

import numpy as np

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

reversed\_arr = arr[::-1]

print("Original array:", arr)

print("Reversed array:", reversed\_arr)

Output:

Original array: [1 2 3 4 5]

Reversed array: [5 4 3 2 1]

In this example, we first created a 1D NumPy array **arr** with values **[1, 2, 3, 4, 5]**. We then used slicing with a step value of -1 to reverse the order of the array and stored the result in the variable **reversed\_arr**. Finally, we printed both the original and reversed arrays using the **print()** function.

1. **Find the number of occurrences of a sequence in a NumPy array**

You can find the number of occurrences of a sequence in a NumPy array by first creating a new array of boolean values indicating where the sequence occurs, and then using the **numpy.count\_nonzero()** function to count the number of **True** values in the array. Here's an example:

import numpy as np

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

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

matches = np.convolve(arr, seq[::-1], mode='valid') == seq.sum()

count = np.count\_nonzero(matches)

print("Number of occurrences:", count)

Output:

Number of occurrences: 3

In this example, we first created a 1D NumPy array **arr** with values **[1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3]**. We then defined the sequence we want to search for as a NumPy array **seq** with values **[1, 2, 3]**.

To find the occurrences of the sequence, we used the **numpy.convolve()** function to perform a cross-correlation between **arr** and the reverse of **seq**. We used the **mode='valid'** parameter to ensure that the resulting array only includes values for which the entire sequence matches. We then compared the resulting array to the sum of **seq** to obtain a boolean array **matches** indicating where the sequence occurs.

Finally, we used the **numpy.count\_nonzero()** function to count the number of **True** values in **matches**, which gives us the number of occurrences of the sequence in **arr**.

7. **Analyse the simple working of an algorithm in Tensor Flow?**

TensorFlow is a popular open-source library for building and training machine learning models. At a high level, TensorFlow works by defining a computational graph that represents the computations that are performed on the input data to produce the desired output. The graph is then executed using a TensorFlow session, which allocates resources and performs the actual computations.

Here's a simple example of how TensorFlow works:

import tensorflow as tf

x = tf.placeholder(tf.float32)

y = tf.placeholder(tf.float32)

z = tf.add(x, y)

with tf.Session() as sess:

result = sess.run(z, feed\_dict={x: 2.0, y: 3.0})

print(result)

In this example, we first import the TensorFlow library. We then define two input placeholders **x** and **y** using the **tf.placeholder()** function. These placeholders will be used to feed input data into the model.

We then define the model by adding the input placeholders together using the **tf.add()** function and storing the result in the variable **z**. This creates a computational graph that represents the addition operation.

Finally, we create a TensorFlow session using the **tf.Session()** function and evaluate the model using the **sess.run()** function. We pass in the **z** variable to specify the output we want to compute, and we provide specific values for the input placeholders using the **feed\_dict** argument. In this case, we pass in the values **2.0** and **3.0** for **x** and **y**, respectively.

The session then executes the computational graph, performs the addition operation on the input values, and returns the result **5.0**. The result is then printed to the console using the **print()** function.

**8. Describe steps involved in making plots. Explain plotting two or more lines on the same plot with an example.**

Steps involved in making plots:

1. Import the required libraries: Before creating any plot, the required libraries need to be imported. The most commonly used libraries for plotting in Python are Matplotlib and Seaborn.
2. Prepare the data: The data to be plotted needs to be in the appropriate format. This involves creating NumPy arrays, Pandas data frames, or other data structures depending on the nature of the data.
3. Create the plot: Once the data is prepared, the plot can be created using Matplotlib or Seaborn. The type of plot, such as line plot, scatter plot, histogram, or box plot, can be specified based on the data being plotted.
4. Customize the plot: The plot can be customized by adding labels to the axes, setting the title, changing the color and style of the plot, adding a legend, and so on. Matplotlib and Seaborn provide various functions to customize the plot.
5. Save and show the plot: Once the plot is ready, it can be saved to a file or displayed on the screen. Matplotlib and Seaborn provide functions to save and show the plot.

Plotting two or more lines on the same plot involves the following steps:

1. Prepare the data: The data to be plotted needs to be in the appropriate format. For example, if we want to plot two lines on the same plot, we need to create two NumPy arrays, Pandas data frames, or other data structures containing the data for each line.
2. Create the plot: We can create the plot using Matplotlib or Seaborn, as described above. To plot multiple lines on the same plot, we can call the plot function multiple times, passing in the data for each line as arguments. We can also customize the plot by adding labels, titles, legends, and so on.

Here's an example of how to plot two lines on the same plot using Matplotlib:

import matplotlib.pyplot as plt

import numpy as np

x = np.linspace(0, 10, 100)

y1 = np.sin(x)

y2 = np.cos(x)

plt.plot(x, y1, label='sin(x)')

plt.plot(x, y2, label='cos(x)')

plt.xlabel('x')

plt.ylabel('y')

plt.title('Sine and Cosine Functions')

plt.legend()

plt.savefig('plot.png')

plt.show()

In this example, we first import the Matplotlib library and create two NumPy arrays **y1** and **y2** containing the sine and cosine values for 100 values of **x** between 0 and 10.

We then create the plot by calling the **plt.plot()** function twice, passing in **x** and **y1** as arguments for the first line, and **x** and **y2** as arguments for the second line. We also specify labels for each line using the **label** parameter.

We then customize the plot by adding x and y axis labels, a title, and a legend using the **plt.xlabel()**, **plt.ylabel()**, **plt.title()**, and **plt.legend()** functions, respectively.

Finally, we save the plot to a file named **plot.png** using the **plt.savefig()** function, and display it on the screen using the **plt.show()** function. The resulting plot shows the sine and cosine functions on the same plot, with different line colors and a legend to differentiate between them.

**9. Develop a Python program to plot two or more lines with legends, different widths and colours**

Here is an example Python program that plots two lines on the same plot with different colors, widths, and legends using Matplotlib:

import numpy as np

import matplotlib.pyplot as plt

x = np.linspace(0, 10, 100)

y1 = np.sin(x)

y2 = np.cos(x)

plt.plot(x, y1, color='red', linewidth=2, label='Sin(x)')

plt.plot(x, y2, color='blue', linewidth=3, label='Cos(x)')

plt.title('Sin(x) and Cos(x)')

plt.xlabel('x-axis')

plt.ylabel('y-axis')

plt.legend()

plt.show()

In this program, we first create some data by generating a NumPy array **x** with 100 evenly spaced values between 0 and 10, and calculating the sine and cosine values for each value of **x**.

We then create the plot by calling the **plt.plot()** function twice, passing in **x** and **y1** for the first line (sine) and **x** and **y2** for the second line (cosine). We specify the color and linewidth of each line using the **color** and **linewidth** parameters, respectively. We also specify a label for each line using the **label** parameter.

Next, we add a title and axis labels to the plot using the **plt.title()**, **plt.xlabel()**, and **plt.ylabel()** functions.

Finally, we add a legend to the plot using the **plt.legend()** function, which automatically creates a legend using the labels specified earlier in the **plt.plot()** calls.

When we run this program, it generates a plot with two lines (sine and cosine) with different colors, widths, and legends.

**10. Describe anatomy of a plot. Explain steps involved in making plots.**

The anatomy of a plot typically includes several elements that are used to visually convey data to the viewer. Here are the common elements of a plot:

1. Title: The title is a brief description of what the plot is showing.
2. X-axis and Y-axis: The x-axis and y-axis are the horizontal and vertical axes, respectively, that show the range of values being plotted.
3. Axis labels: The axis labels provide a description of the data represented by each axis.
4. Data: The data are the actual values being plotted.
5. Legends: Legends are used to explain the meaning of different colored lines or markers in a plot.
6. Gridlines: Gridlines are horizontal and vertical lines that help to guide the viewer's eye and make it easier to read the values on the axes.
7. Tick marks: Tick marks are small lines or dots that show the exact position of each value on the axes.

Here are the steps involved in making plots:

1. Import the required libraries: Before making a plot, you need to import the libraries that you will use to create the plot. The most common plotting library in Python is Matplotlib.
2. Prepare the data: The next step is to prepare the data that you want to plot. This involves creating arrays or dataframes that represent the x-axis and y-axis values.
3. Create the plot: Once the data is prepared, you can use Matplotlib functions to create the plot. You can choose from a variety of plot types, such as line plots, scatter plots, bar plots, and more.
4. Customize the plot: After creating the plot, you can customize its appearance by changing the colors, line widths, labels, and other settings.
5. Save or show the plot: Finally, you can save the plot as an image file or display it directly in the Python console using the **plt.show()** function.

Overall, the process of making a plot involves importing the necessary libraries, preparing the data, creating the plot, customizing the appearance of the plot, and saving or displaying the plot.

**11. Characterize the Data Frames in Pandas?**

In Pandas, a data frame is a two-dimensional labeled data structure with columns of potentially different data types. It is like a spreadsheet or SQL table, or a collection of Series objects aligned by index. Data frames can be created from various sources such as CSV files, SQL databases, and Excel sheets.

The main characteristics of data frames in Pandas are:

1. Two-dimensional structure: A data frame has rows and columns, where rows are identified by an index and columns are identified by column names.
2. Columnar data storage: The data in a data frame is stored in columns, where each column can have a different data type.
3. Label-based indexing: Rows and columns can be accessed by their labels or names using the **.loc** attribute.
4. Position-based indexing: Rows and columns can also be accessed by their positions using the **.iloc** attribute.
5. Missing data handling: Pandas data frames can handle missing data values, represented as NaN (Not a Number) or None.
6. Broadcasting operations: Arithmetic operations can be applied to entire columns or rows, rather than just individual elements.
7. Grouping and aggregation: Pandas data frames provide powerful tools for grouping and aggregating data based on one or more columns.
8. Merge and join operations: Data frames can be combined with other data frames based on common columns using the **merge()** and **join()** functions.
9. Time-series data handling: Pandas provides powerful tools for working with time-series data, including date/time parsing, resampling, and rolling window calculations.
10. Wide range of input/output options: Pandas can read and write data in various formats such as CSV, Excel, SQL databases, JSON, and more.

**12. Explain the following: a. Matplotlib b. Seaborn c. Plotly d. ggplot**

a. Matplotlib is a data visualization library for Python. It provides a wide range of tools for creating static, animated, and interactive visualizations in Python. Matplotlib is highly customizable, allowing users to control almost every aspect of the visualizations.

b. Seaborn is another data visualization library for Python built on top of Matplotlib. It provides a high-level interface for creating statistical graphics such as heat maps, time series, and regression plots. Seaborn is known for its simplicity and elegance, making it easy to create professional-looking visualizations with just a few lines of code.

c. Plotly is a data visualization library for creating interactive, web-based visualizations. It provides a range of tools for creating 2D and 3D visualizations, as well as dashboards and reports. Plotly is particularly popular for creating interactive visualizations in Jupyter notebooks and online.

d. ggplot is a data visualization library for Python based on the popular ggplot2 library in R. It provides a high-level interface for creating statistical graphics based on the grammar of graphics. ggplot allows users to create complex visualizations with minimal code, making it a powerful tool for data exploration and communication. However, ggplot is not as widely used as Matplotlib and Seaborn in the Python data science community.

**13. Explain the Applications of SciPy, Scrapy, Scikit-learn, PyGame, PyTorch, PyBrain and Keras.**

a. SciPy is a Python library for scientific and technical computing. It provides a range of tools for optimization, integration, interpolation, signal and image processing, and more. Some applications of SciPy include machine learning, signal processing, and scientific simulations.

b. Scrapy is a Python library for web crawling and web scraping. It allows users to extract data from websites and APIs, and store it in a structured format such as JSON or CSV. Some applications of Scrapy include price monitoring, content aggregation, and data mining.

c. Scikit-learn is a Python library for machine learning. It provides a range of tools for classification, regression, clustering, and dimensionality reduction, as well as tools for model selection and evaluation. Some applications of Scikit-learn include fraud detection, image classification, and natural language processing.

d. PyGame is a Python library for game development. It provides a range of tools for building 2D games, including graphics rendering, sound processing, and user input handling. Some applications of PyGame include educational games, arcade games, and simulations.

e. PyTorch is a Python library for machine learning and deep learning. It provides a range of tools for building and training neural networks, as well as tools for data loading and transformation. Some applications of PyTorch include image and speech recognition, natural language processing, and robotics.

f. PyBrain is a Python library for machine learning, specifically for building and training neural networks. It provides a range of tools for creating and training neural networks, as well as tools for reinforcement learning and unsupervised learning. Some applications of PyBrain include robotics, game AI, and financial forecasting.

g. Keras is a Python library for building and training deep learning models. It provides a high-level interface for creating and training neural networks, as well as a range of pre-trained models for image and text classification. Some applications of Keras include object recognition, sentiment analysis, and recommendation systems.

**14. List the advantages NumPy Arrays have over (nested) Python lists?**

NumPy arrays have several advantages over (nested) Python lists:

1. Speed: NumPy arrays are significantly faster than (nested) Python lists because they are implemented in C and optimized for numerical operations.
2. Memory Efficiency: NumPy arrays take up less memory than (nested) Python lists because they are homogeneous (i.e., all elements have the same data type).
3. Convenience: NumPy arrays provide a range of convenient functions for performing mathematical operations on arrays, such as element-wise multiplication, addition, and division.
4. Broadcasting: NumPy arrays allow for broadcasting, which means that operations can be performed on arrays with different shapes and sizes without the need for explicit looping.
5. Multi-dimensional arrays: NumPy arrays can have any number of dimensions, allowing for the efficient manipulation of multi-dimensional data.
6. Integration with other libraries: NumPy arrays can easily be integrated with other libraries, such as Pandas, Matplotlib, and SciPy, making it a powerful tool for scientific computing and data analysis.
7. Mathematical Functions: NumPy arrays provide a range of mathematical functions such as sin, cos, exp, etc. that are not available in Python lists.

Overall, NumPy arrays are a more efficient, convenient, and powerful tool for numerical computing than (nested) Python lists.

**16. Explain the use of init function in python.**

The **\_\_init\_\_** function is a special method in Python classes that is called when an object of the class is created. It is used to initialize the attributes of the object and perform any other setup that is required.

The **\_\_init\_\_** method is defined using the following syntax:

class MyClass:

def \_\_init\_\_(self, arg1, arg2, ...):

self.attribute1 = arg1

self.attribute2 = arg2

...

The **self** parameter is a reference to the object being created, and it is automatically passed in when the **\_\_init\_\_** method is called. The arguments **arg1**, **arg2**, etc., are used to initialize the attributes of the object.

For example, consider the following class definition:

class Person:

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

In this example, the **\_\_init\_\_** method takes two arguments, **name** and **age**, and initializes the attributes **name** and **age** of the **Person** object. This allows us to create **Person** objects with specific names and ages, as shown below:

person1 = Person("Alice", 25)

person2 = Person("Bob", 30)

**17. Explain Python's static methods.**

In Python, a static method is a method that belongs to a class rather than an instance of that class. This means that a static method can be called on the class itself, rather than on an object of that class.

Static methods are defined using the **@staticmethod** decorator, which is placed above the method definition. The method does not have access to the class or instance attributes, but can still be useful for encapsulating a utility function that does not depend on the class or instance state.

Here is an example of a static method:

class MyClass:

@staticmethod

def my\_static\_method(arg1, arg2):

# do something with arg1 and arg2

return result

In this example, the **my\_static\_method** method is defined as a static method using the **@staticmethod** decorator. It takes two arguments, **arg1** and **arg2**, and returns a result after performing some operation.

To call a static method, you can use the class name followed by the method name, like this:

result = MyClass.my\_static\_method(arg1, arg2)

Note that you do not need to create an instance of the class to call a static method.

**18. Explain Python's Nested Class.**

In Python, a nested class is a class that is defined inside another class. Nested classes are also known as inner classes. The nested class has access to the attributes and methods of the enclosing class, which makes them useful for organizing related functionality within a larger class.

Nested classes are defined by including a class definition inside another class definition. Here is an example:

In this example, **OuterClass** has a nested class called **InnerClass**. **InnerClass** has its own constructor method, which initializes its own **name** attribute.

The **print\_names** method of **OuterClass** demonstrates how to create an instance of **InnerClass** and access its **name** attribute. The **print\_names** method first prints the **name** attribute of the **OuterClass** instance, and then creates an instance of **InnerClass**. It then prints the **name** attribute of the **InnerClass** instance.

To create an instance of **OuterClass**, you can use the following code:

outer = OuterClass()

outer.print\_names()

This will output the following:

OuterClass name: OuterClass

InnerClass name: InnerClass

**19. List the advantages of using OOPs.**

Object-oriented programming (OOP) has several advantages over other programming paradigms. Some of the key advantages of using OOP are:

1. Modularity: OOP allows you to create self-contained, modular objects that can be reused in different parts of your code. This makes your code more organized and easier to maintain.
2. Encapsulation: OOP allows you to encapsulate the data and behavior of an object, which means that the internal workings of the object are hidden from the outside world. This makes it easier to reason about your code and reduces the risk of unwanted side effects.
3. Inheritance: OOP allows you to create new classes based on existing classes. This allows you to reuse existing code and create new classes that share common attributes and behaviors.
4. Polymorphism: OOP allows you to write code that can work with objects of different classes in a uniform way. This allows you to write more flexible and reusable code.
5. Code reuse: OOP allows you to create libraries of reusable code that can be used in different projects. This can save you time and effort and improve the quality of your code.
6. Simplicity: OOP allows you to write code that is easier to understand and maintain. It provides a more intuitive and natural way of thinking about your code.

Overall, OOP provides a powerful set of tools for creating flexible, reusable, and maintainable code. By using OOP, you can create more efficient, reliable, and scalable software applications.

**20.Explain access specifiers in python.**

In Python, access specifiers are not used in the same way as in other object-oriented programming languages like Java or C++. However, there are conventions that are used to indicate the level of accessibility of class attributes and methods.

The most commonly used convention is to use underscores to indicate the intended access level. Here are the different levels of access and how they are typically indicated:

1. Public: A public attribute or method is one that can be accessed from anywhere in your code, including outside of the class definition. Public attributes and methods are not typically marked with any underscores.
2. Protected: A protected attribute or method is one that is intended to be accessed only within the class definition and its subclasses. Protected attributes and methods are typically marked with a single underscore at the beginning of the name.
3. Private: A private attribute or method is one that is intended to be accessed only within the class definition. Private attributes and methods are typically marked with a double underscore at the beginning of the name.

It's important to note that these conventions are not enforced by the Python language itself. They are simply a way to communicate the intended level of access to other developers who may be working with your code.

Here's an example class that uses these access conventions:

class MyClass:

def \_\_init\_\_(self):

self.public\_attribute = 42

self.\_protected\_attribute = "hello"

self.\_\_private\_attribute = "world"

def public\_method(self):

print("This is a public method")

def \_protected\_method(self):

print("This is a protected method")

def \_\_private\_method(self):

print("This is a private method")

In this example, **public\_attribute** and **public\_method** are both public, **\_protected\_attribute** and **\_protected\_method** are both protected, and **\_\_private\_attribute** and **\_\_private\_method** are both private. However, it's important to note that the double underscore prefix causes Python to "name mangling", which modifies the name of the attribute or method to avoid name collisions with subclasses. This means that it is still technically possible to access private attributes and methods from outside the class definition, although it is not recommended.

Overall, while Python does not have traditional access specifiers like other languages, the conventions of using underscores to indicate access levels are widely used and can be useful for communicating your code's intended access levels to other developers.

**21.Can a parent class be called without first creating an instance of it? Explain.**

No, a parent class cannot be called without first creating an instance of it. In object-oriented programming, a class is a blueprint or template for creating objects, and an object is an instance of a class.

To access the attributes and methods of a parent class, you need to create an instance of the class (i.e., an object) and use dot notation to call the attributes or methods. This is because the attributes and methods of a class are defined within the context of an instance of the class, and without an instance, they have no meaning.

For example, suppose you have a parent class called **Animal** with an attribute **species** and a method **make\_sound()**. To access these attributes and methods, you need to first create an instance of the **Animal** class, like so:

class Animal:

def \_\_init\_\_(self, species):

self.species = species

def make\_sound(self):

print("Generic animal sound")

animal = Animal("cat")

print(animal.species)

nimal.make\_sound()

In this example, we create an instance of the **Animal** class called **animal** and use dot notation to access its **species** attribute and **make\_sound()** method. Without first creating an instance of the **Animal** class, we wouldn't be able to access these attributes and methods.

**22.How can you determine whether a class is a subclass of another class?**

In Python, you can determine whether a class is a subclass of another class using the **issubclass()** function or the **subclass()** method.

1. Using the **issubclass()** function:

The **issubclass()** function takes two arguments: a subclass and a superclass. It returns **True** if the subclass is a subclass of the superclass, and **False** otherwise. Here's an example:

class Animal:

pass

class Cat(Animal):

pass

class Dog(Animal):

pass

print(issubclass(Cat, Animal))

print(issubclass(Dog, Animal))

print(issubclass(Dog, Cat))

In this example, **Cat** and **Dog** are both subclasses of **Animal**, so **issubclass(Cat, Animal)** and **issubclass(Dog, Animal)** both return **True**. However, **Dog** is not a subclass of **Cat**, so **issubclass(Dog, Cat)** returns **False**.

1. Using the **subclass()** method:

The **subclass()** method is a method of the superclass that returns **True** if the specified class is a subclass of the superclass, and **False** otherwise. Here's an example:

class Animal:

pass

class Cat(Animal):

pass

class Dog(Animal):

pass

print(Animal.\_\_subclasscheck\_\_(Cat))

print(Animal.\_\_subclasscheck\_\_(Dog))

print(Cat.\_\_subclasscheck\_\_(Dog))

In this example, we use the **\_\_subclasscheck\_\_()** method to check whether **Cat** and **Dog** are both subclasses of **Animal**, and whether **Dog** is a subclass of **Cat**. As before, **Cat** and **Dog** are both subclasses of **Animal**, and **Dog** is not a subclass of **Cat**.

**23. How do you make a Python class that is empty?**

In Python, you can create an empty class by simply defining a class with no content inside the class definition. Here's an example:

class EmptyClass:

pass

In this example, we define a class called **EmptyClass** that has no content inside the class definition. The **pass** statement is used as a placeholder statement to indicate that the class is intentionally left empty.

This class can be instantiated, but it won't have any attributes or methods defined. Here's an example of instantiating an object of this class:

empty\_obj = EmptyClass()

In this example, we create an instance of **EmptyClass** called **empty\_obj**. Since **EmptyClass** has no attributes or methods defined, **empty\_obj** doesn't have any attributes or methods either.

**24. Explain a class object or instance in python.**

In Python, a class is a blueprint or template for creating objects, while an object or instance is a specific realization or instantiation of that class.

A class is defined using the **class** keyword followed by the name of the class, like so:

class MyClass:

# class definition

Within the class definition, you can define attributes (i.e., variables) and methods (i.e., functions) that belong to the class.

To create an instance or object of a class, you use the class name followed by parentheses, like so:

my\_obj = MyClass()

In this example, we create an instance of the **MyClass** class called **my\_obj**. This creates a new object with its own set of attributes and methods that are defined by the class.

You can access the attributes and methods of an object using dot notation, like so:

my\_obj.my\_attribute = "Hello"

print(my\_obj.my\_attribute) # output: "Hello"

def my\_method(self, arg):

print(arg)

my\_obj.my\_method("World") # output: "World"

In this example, we set the value of **my\_attribute** for **my\_obj** to "Hello", and then print the value of **my\_attribute**. We also define a method called **my\_method()** that takes an argument and prints it, and then call this method on **my\_obj** with the argument "World".

Note that each instance of a class has its own set of attributes and methods that are separate from those of other instances of the same class. This allows you to create multiple objects of the same class, each with their own unique properties and behaviors.

**25.Can you call the base class method without creating an instance? Explain.**

No, you cannot call the base class method without creating an instance of the class.

In Python, you typically call a method of a class by creating an instance of that class and then calling the method on that instance. For example, consider the following class:

class MyClass:

def my\_method(self):

print("Hello, World!")

my\_obj = MyClass()

my\_obj.my\_method()

In this example, we create an instance of the **MyClass** class called **my\_obj**, and then call the **my\_method()** method on this instance.

If you want to call a method of the base class from a subclass, you typically use the **super()** function. The **super()** function returns a temporary object of the superclass, which allows you to call its methods. For example, consider the following subclass:

class MySubclass(MyClass):

def my\_method(self):

super().my\_method() # call the base class method

print("How are you?")

my\_subobj = MySubclass()

my\_subobj.my\_method() # output: "Hello, World!" followed by "How are you?"

In this example, we define a subclass of **MyClass** called **MySubclass**, which overrides the **my\_method()** method of the base class to first call the base class method using **super().my\_method()**, and then print an additional message.

When we create an instance of **MySubclass** called **my\_subobj** and call its **my\_method()** method, it first calls the base class method (which prints "Hello, World!"), and then prints the additional message ("How are you?"). Note that we still need to create an instance of the subclass to call its method, as we do with the base class.

**26.What is the difference between a class and a structure?**

In Python, there is no such thing as a "structure" like there is in some other programming languages, such as C or C++. Instead, we typically use the term "class" to describe a user-defined type that encapsulates data and behavior.

That being said, in other programming languages that do have the concept of a "structure", there are some key differences between structures and classes:

1. In general, structures are simpler than classes. They typically only define data fields (also known as members) and don't have methods (i.e., functions) associated with them.
2. Structures are typically used to represent simple data types or data structures, such as a point in 3D space or a timestamp, whereas classes are used to represent more complex objects with behavior and data, such as a bank account or a car.
3. Structures are often used in performance-critical code, where the overhead of using a class (such as the cost of creating objects and calling methods) would be too high. In contrast, classes are often used in more complex applications where encapsulation and abstraction are important.

In summary, while the terms "structure" and "class" are sometimes used interchangeably, they typically refer to different concepts. Structures are simple data types used to represent data, while classes are more complex objects that encapsulate both data and behavior.

**27.Write a Python program to import a built in array module and display the namespace of the said module.**

Here's a Python program that imports the built-in **array** module and displays its namespace:

import array

print("Namespace of array module:")

print(dir(array))

In this program, we first import the **array** module using the **import** statement. We then use the **dir()** function to display the namespace of the **array** module. The **dir()** function returns a list of all the names (i.e., variables, functions, classes, etc.) defined in the specified module.

When you run this program, you should see output similar to the following:

Namespace of array module:

['\_\_add\_\_', '\_\_class\_\_', '\_\_contains\_\_', '\_\_copy\_\_', '\_\_deepcopy\_\_', '\_\_delattr\_\_', '\_\_delitem\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_getitem\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_iadd\_\_', '\_\_imul\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_iter\_\_', '\_\_le\_\_', '\_\_len\_\_', '\_\_lt\_\_', '\_\_mul\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_reversed\_\_', '\_\_rmul\_\_', '\_\_setattr\_\_', '\_\_setitem\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', 'append', 'buffer\_info', 'byteswap', 'count', 'extend', 'frombytes', 'fromfile', 'fromlist', 'fromunicode', 'index', 'insert', 'itemsize', 'pop', 'remove', 'reverse', 'tobytes', 'tofile', 'tolist', 'tounicode', 'typecode']

This output shows all the names defined in the **array** module, including functions like **append()** and **pop()**, as well as attributes like **itemsize** and **typecode**.

**28.Write a Python class named Circle constructed from a radius and two methods that will compute the area and the perimeter of a circle.**

Here's an example Python class named **Circle** that is constructed from a radius and includes two methods, **area()** and **perimeter()**, which compute the area and perimeter of a circle:

class Circle:

def \_\_init\_\_(self, radius):

self.radius = radius

def area(self):

return 3.14 \* (self.radius \*\* 2)

def perimeter(self):

return 2 \* 3.14 \* self.radius

In this class, the **\_\_init\_\_()** method is the constructor, which is called when a new **Circle** object is created. It takes a single argument, **radius**, and sets an instance variable **self.radius** to the value of the argument.

The **area()** method computes the area of the circle using the formula πr² (where r is the radius), and returns the result.

The **perimeter()** method computes the perimeter (i.e., circumference) of the circle using the formula 2πr, and returns the result.

Here's an example usage of the **Circle** class:

c = Circle(5)

print(c.area())

print(c.perimeter())

In this example, we create a new **Circle** object with radius 5 and store it in the variable **c**. We then call the **area()** and **perimeter()** methods on the **c** object, which compute the area (78.5) and perimeter (31.4) of the circle, respectively.

**29. Are class and structure the same? If not, what's the difference between a class and a structure?**

A class and a structure are not the same thing, although they share some similarities.

In Python, a class is a blueprint for creating objects that have properties (i.e., attributes) and methods. Classes are used to define data types and encapsulate behavior, and are an essential component of object-oriented programming.

In other programming languages like C, C++, and C#, a structure is a composite data type that groups together variables of different data types. Structures allow you to define your own data types that can store multiple pieces of data together. Like classes, structures can have methods (in C++ and C#) that operate on their data members.

The main difference between classes and structures is that classes are used to define objects with behavior, while structures are used to group together variables of different data types. In Python, you can use a class to define a data type that has both attributes and methods, while in languages like C++, you would typically use a class for that purpose and a structure for defining composite data types with no behavior.

It's worth noting that in Python, there is no difference between defining a class and a structure. You can use the **class** keyword to define a class that has attributes and methods, and you can also use the **class** keyword to define a "structure" that only has attributes (i.e., instance variables). However, this is not a recommended practice in Python, as it can be confusing for other developers who expect classes to have behavior (i.e., methods).

**30.Create a Temperature class. Make two methods: a) convert Fahrenheit -- It will take Celsius and will print it into Fahrenheit. b) convert Celsius -- It will take Fahrenheit and will convert it into Celsius.**

Here's an example implementation of a **Temperature** class in Python, with two methods **convert\_fahrenheit()** and **convert\_celsius()**, as described:

class Temperature:

def convert\_fahrenheit(self, celsius):

fahrenheit = (celsius \* 1.8) + 32

print(f"{celsius}°C = {fahrenheit}°F")

def convert\_celsius(self, fahrenheit):

celsius = (fahrenheit - 32) / 1.8

print(f"{fahrenheit}°F = {celsius}°C")

In this implementation, the **convert\_fahrenheit()** method takes a temperature in Celsius as an argument, computes its equivalent in Fahrenheit using the formula **(Celsius \* 1.8) + 32**, and then prints the result.

The **convert\_celsius()** method takes a temperature in Fahrenheit as an argument, computes its equivalent in Celsius using the formula **(Fahrenheit - 32) / 1.8**, and then prints the result.

Here's an example usage of the **Temperature** class:

temp = Temperature()

temp.convert\_fahrenheit(20)

temp.convert\_celsius(68)

In this example, we create a new **Temperature** object called **temp**, and then call its **convert\_fahrenheit()** and **convert\_celsius()** methods to convert temperatures between Celsius and Fahrenheit. The output shows the results of the conversions.

**31. Write a Python program that imports the abs() function using the built-ins module, displays the documentation of the abs() function and finds the absolute value of -155.**

Here's a Python program that imports the abs() function, displays its documentation, and finds the absolute value of 155:

import builtins

print(help(builtins.abs))

num = 155

abs\_num = abs(num)

print("The absolute value of", num, "is", abs\_num)

Output:

Help on built-in function abs in module builtins:

abs(x, /)

Return the absolute value of the argument.

None

The absolute value of 155 is 155

Note that the **help()** function is used to display the documentation of the **abs()** function. The **abs()** function takes a single argument and returns its absolute value. In the above program, we pass the value **155** as an argument to the **abs()** function and store the result in the variable **abs\_num**. We then print the value of **abs\_num** using the **print()** function.

**32. Write a Python class to implement pow(x, n).**

class Power:

def pow(self, x, n):

if n == 0:

return 1

elif n == 1:

return x

elif n < 0:

return 1 / self.pow(x, -n)

elif n % 2 == 0:

return self.pow(x\*x, n//2)

else:

return x \* self.pow(x\*x, (n-1)//2)

The **Power** class has a single method called **pow** that takes two arguments **x** and **n**. It returns the result of raising **x** to the power of **n**. This method implements the algorithm for exponentiation by squaring which is an efficient algorithm to compute the power of a number.

Here's how to use the **Power** class:

power = Power()

result = power.pow(2, 10)

print(result)

result = power.pow(3, -2)

print(result)

In the above example, we create an instance of the **Power** class and call the **pow** method to compute the power of a number. The first example computes the value of 2 raised to the power of 10, which is 1024. The second example computes the value of 3 raised to the power of -2, which is 1/3^2 = 0.1111111111111111.

**33. Write a Python class to reverse a string word by word.**

class StringReverser:

def reverse\_words(self, s):

words = s.split()

words.reverse()

reversed\_s = " ".join(words)

return reversed\_s

The **StringReverser** class has a single method called **reverse\_words** that takes a string **s** as an argument. It returns the reversed string, with the words in reversed order.

Here's how to use the **StringReverser** class:

reverser = StringReverser()

result = reverser.reverse\_words("Hello World")

print(result)

result = reverser.reverse\_words("The quick brown fox jumps over the lazy dog")

print(result)

In the above example, we create an instance of the `StringRevers

34. Write a Python class that has two methods: get\_String and print\_String , get\_String accept a string from the user and print\_String prints the string in upper case.

class StringProcessor:

def \_\_init\_\_(self):

self.user\_string = ''

def get\_String(self):

self.user\_string = input("Enter a string: ")

def print\_String(self):

print("String in upper case:", self.user\_string.upper())

This class is called **StringProcessor** and has two methods:

* **get\_String()**: This method accepts a string from the user and sets it as the **user\_string** attribute of the class instance.
* **print\_String()**: This method prints the **user\_string** attribute in upper case.

Here's an example usage of the class:

sp = StringProcessor()

sp.get\_String()

sp.print\_String()

When you run this program and provide a string as input, it will output the string in upper case.

**35. Illustrate the difference between: a) read( ) and readlines ( ) b) write( ) and writelines( ) c) r+ file mode and rb+ mode. d) w‟ and „a‟ modes**

a) **read()** and **readlines()**:

* **read()** reads the entire content of the file as a single string. It takes an optional argument that specifies the number of bytes to read.
* **readlines()** reads the entire content of the file and returns a list of lines, where each line is a string.

b) **write()** and **writelines()**:

* **write()** writes a string to the file. If the file does not exist, it creates a new file. If the file exists, it overwrites the existing content of the file.
* **writelines()** writes a list of strings to the file. Each string is written as a separate line. If the file does not exist, it creates a new file. If the file exists, it overwrites the existing content of the file.

c) **r+** file mode and **rb+** mode:

* **r+** mode opens a file for reading and writing. If the file does not exist, it raises an error. In this mode, the file pointer is initially at the beginning of the file.
* **rb+** mode opens a binary file for reading and writing. If the file does not exist, it raises an error. In this mode, the file pointer is initially at the beginning of the file.

d) **w** and **a** modes:

* **w** mode opens a file for writing. If the file does not exist, it creates a new file. If the file exists, it overwrites the existing content of the file.
* **a** mode opens a file for appending. If the file does not exist, it creates a new file. If the file exists, it appends the new content to the end of the existing content.

**36. Apply the below instructions when writing the program. (a) Import Module\_Imp2 as mi (b) Take two integers a and b as inputs from the user (c) Call the function mi.arithoperation by passing a, b.**

import Module\_Imp2 as mi

a = int(input("Enter a: "))

b = int(input("Enter b: "))

mi.arithoperation(a, b)

In the above program, we first import the **Module\_Imp2** module using the **import** statement and alias it as **mi** using the **as** keyword.

Next, we use the **input()** function to take two integers **a** and **b** as inputs from the user.

Finally, we call the **arithoperation()** function from the **mi** module by passing the values of **a** and **b** as arguments.

**37. Demonstrate the use of Class while writing a code following the given instructions to add the details of the two students by taking the inputs from the user. a) Create a class Student. b) Create an instance Stud\_1 of class Student. c) Create another instance Stud\_2 of class Student. d) Take name, age, and degree of the student as inputs from the user. e) Print the details of the student.**

class Student:

def \_\_init\_\_(self, name, age, degree):

self.name = name

self.age = age

self.degree = degree

def print\_details(self):

print("Name: ", self.name)

print("Age: ", self.age)

print("Degree: ", self.degree)

Stud\_1 = Student(input("Enter name of student 1: "),

int(input("Enter age of student 1: ")),

input("Enter degree of student 1: "))

Stud\_2 = Student(input("Enter name of student 2: "),

int(input("Enter age of student 2: ")),

input("Enter degree of student 2: "))

print("Details of student 1:")

Stud\_1.print\_details()

print("\nDetails of student 2:")

Stud\_2.print\_details()

In the above program, we first define a **Student** class that has a constructor method **\_\_init\_\_()** to initialize the name, age, and degree attributes of a student object. It also has a method **print\_details()** that prints the details of the student object.

**38. f= open(“data.txt”) Make use of the code given above and write the answers of the following: a) Identify name of the file. b) What is „f‟ in above code? c) What is the mode of operation in the above file? d) Discuss the different types of close ().**

a) The name of the file is "data.txt".

b) In the given code, **f** is a file object that represents the file "data.txt". This object can be used to read data from or write data to the file.

c) The mode of operation in the above code is the default mode, which is "r" (read mode). This means that the file is opened for reading, and any attempt to write to the file will result in an error.

d) There are three types of **close()** methods that can be used to close a file:

* **close()** method: This is the most common method to close a file. It flushes any unwritten data to the file, closes the file handle, and releases any system resources associated with the file handle.
* **flush()** method: This method is used to flush any unwritten data to the file, but it does not close the file handle. This is useful when you want to ensure that all data written to the file is saved, but you still want to write more data to the file.
* **\_\_exit\_\_()** method: This method is used to close a file automatically when it is used with a context manager (i.e., with the **with** statement). When the block of code inside the **with** statement completes, the **\_\_exit\_\_()** method is called automatically, which closes the file handle and releases any system resources associated with the file handle.

**39. Build a class Employee, which contains the details of an employee like name and salary. Take name and salary as inputs from the console, print the result.**

class Employee:

def \_\_init\_\_(self, name, salary):

self.name = name

self.salary = salary

def print\_details(self):

print("Name: ", self.name)

print("Salary: ", self.salary)

name = input("Enter name of employee: ")

salary = float(input("Enter salary of employee: "))

emp = Employee(name, salary)

emp.print\_details()

In the above program, we first define an **Employee** class that has a constructor method **\_\_init\_\_()** to initialize the name and salary attributes of an employee object. It also has a method **print\_details()** that prints the details of the employee object.

We then use the **input()** function to take the name and salary of an employee as inputs from the console.

Next, we create an instance of the **Employee** class using the input values for name and salary.

Finally, we call the **print\_details()** method of the **emp** object to print the details of the employee.

**40. Assume a filename and write a Python program to copy one file to another file in file handling. Explain the tell() and seek() in python file handling.**

source\_file = input("Enter source file name: ")

destination\_file = input("Enter destination file name: ")

# open source file for reading

with open(source\_file, "rb") as source:

with open(destination\_file, "wb") as destination:

destination.write(data)

print("File copied successfully.")

In the above program, we first take the input for the source and destination file names.

We then open the source file in binary read mode (**"rb"**) using a **with** statement. This ensures that the file is closed automatically after we're done reading from it.

We also open the destination file in binary write mode (**"wb"**) using another **with** statement. This ensures that the file is closed automatically after we're done writing to it.

We then read the data from the source file using the **read()** method and write it to the destination file using the **write()** method.

Finally, we print a message to indicate that the file has been copied successfully.

Now, let's discuss **tell()** and **seek()** in Python file handling:

* **tell()**: This method returns the current position of the file pointer. In other words, it returns the byte offset from the beginning of the file to the current position of the file pointer. For example, if you have a file object **f**, you can use **f.tell()** to get the current position of the file pointer.
* **seek()**: This method is used to change the current position of the file pointer. It takes two arguments: the first argument is the byte offset from the beginning of the file, and the second argument (optional) specifies the reference point from where the offset should be calculated. The reference point can be one of three constants: **0** (the beginning of the file), **1** (the current position of the file pointer), or **2** (the end of the file). For example, if you have a file object **f**, you can use **f.seek(10)** to move the file pointer to the 10th byte from the beginning of the file. You can also use **f.seek(10, 0)** to move the file pointer to the 10th byte from the beginning of the file, or **f.seek(-10, 2)** to move the file pointer to the 10th byte from the end of the file.

**41. Define a function check Negative Number which has an argument num. Write a program to check the given num is a positive or negative. Take the input number from the user, and print the result as shown in the examples.**

Sample Input and Output 1: a: 20 positive

Sample Input and Output 2: a: -90 negative

Here's the Python code to define the function **checkNegativeNumber()** that takes a number **num** as an argument and checks whether it's positive or negative. It then prints the result accordingly.

def checkNegativeNumber(num):

if num >= 0:

print(num, "is positive")

else:

print(num, "is negative")

Now, to take input from the user and use this function to check if the input is positive or negative, you can use the following code:

num = int(input("Enter a number: "))

checkNegativeNumber(num)

When you run the program and enter a number, it will call the **checkNegativeNumber()** function with the user input as the argument. The function will then check whether the number is positive or negative and print the result accordingly.

Sample Input and Output 1:

Enter a number: 20

20 is positive

Sample Input and Output 2:

Enter a number: -90

-90 is negative

**42. The below program Module\_Imp2 is already written**

. • This file is same as Module1 written earlier, which takes two parameters and does arithmetic operations on these two, and print the result.

• Import it in your Module4.py.

• Take two integers a and b as input from the user.

• Call the function arithoperations() of Module\_Imp2 module by passing the arguments a and b. Sample Input and Output: a: 12 b: 10 addition: 22 subtraction: 2 multiplication: 120 division: 1.2

To complete this task, you can follow these steps:

1. Save the below code as **Module\_Imp2.py** in the same directory where you will save **Module4.py**.

def arithoperations(a,b):

add = a + b

sub = a - b

mul = a \* b

div = a / b

print("addition:", add)

print("subtraction:", sub)

print("multiplication:", mul)

print("division:", div)

1. Now, create a new file **Module4.py** in the same directory and add the following code:

import Module\_Imp2

a = int(input("Enter first number: "))

b = int(input("Enter second number: "))

Module\_Imp2.arithoperations(a,b)

1. When you run this program, it will ask the user to enter two numbers. After taking the input, it will import the **Module\_Imp2** module and call its **arithoperations()** function, passing the user input as arguments. The **arithoperations()** function will perform arithmetic operations on these two numbers and print the results.

Sample Input and Output:

Enter first number: 12

Enter second number: 10

addition: 22

subtraction: 2

multiplication: 120

division: 1.2

**43. Apply the below instructions when writing the program. (a) Import Module\_Imp2 as mi (b) Take two integers a and b as inputs from the user. (c) Call the function mi.arithoperation by passing a, b.**

To complete this task, you can follow these steps:

1. Save the below code as **Module\_Imp2.py** in the same directory where you will save your main program file.

def arithoperations(a,b):

add = a + b

sub = a - b

mul = a \* b

div = a / b

print("addition:", add)

print("subtraction:", sub)

print("multiplication:", mul)

print("division:", div)

1. Now, create a new file and add the following code:

import Module\_Imp2 as mi

a = int(input("Enter first number: "))

b = int(input("Enter second number: "))

mi.arithoperations(a,b)

1. When you run this program, it will ask the user to enter two numbers. After taking the input, it will import the **Module\_Imp2** module as **mi** and call its **arithoperations()** function, passing the user input as arguments. The **arithoperations()** function will perform arithmetic operations on these two numbers and print the results.

Sample Input and Output:

Enter first number: 12

Enter second number: 10

addition: 22

subtraction: 2

multiplication: 120

division: 1.2

**44. Inspect whether class and structure are the same? If not, what's the difference between a class and a structure?**

In programming, a class and a structure are similar in that they both represent a blueprint for creating objects with similar properties and behaviors. However, there are some differences between the two.

A class is a reference type, which means that when you create an instance of a class, you are actually creating a reference to that instance. Classes can have properties, methods, events, and other members, and can be inherited from and extended. Classes are typically used in object-oriented programming (OOP) languages like Java, C#, and Python.

A structure, on the other hand, is a value type. When you create an instance of a structure, you are creating a copy of that instance. Structures can also have properties, methods, and other members, but cannot be inherited from. Structures are typically used in procedural programming languages like C and C++.

One key difference between classes and structures is how they are stored in memory. Classes are reference types and are stored on the heap, while structures are value types and are stored on the stack. This means that creating and copying structures is typically faster than creating and copying classes.

Another difference is how they are passed as arguments to methods. Classes are passed by reference, which means that changes made to the object inside the method are reflected outside the method as well. Structures, on the other hand, are passed by value, which means that a copy of the structure is passed to the method and changes made inside the method do not affect the original structure.

In summary, while classes and structures are similar in that they both represent a blueprint for creating objects with similar properties and behaviors, they differ in how they are stored in memory, how they are passed as arguments to methods, and how they can be inherited from.

**45. Inspect whether a parent class be called without first creating an instance of it? Explain.**

Yes, a parent class can be called without first creating an instance of it. This is because when a child class is created by inheriting from a parent class, the parent class is automatically loaded into memory and becomes available for use by the child class.

For example, consider the following code:

class Parent:

def \_\_init\_\_(self):

self.parent\_var = "I am a parent."

class Child(Parent):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.child\_var = "I am a child."

In this code, the **Child** class is inheriting from the **Parent** class. When an instance of the **Child** class is created, the **\_\_init\_\_()** method of the **Parent** class is automatically called using the **super()** function. This means that the **Parent** class is loaded into memory and can be accessed without explicitly creating an instance of it.

For example, you can access a method or attribute of the **Parent** class directly from the **Child** class, like this:

print(Parent().parent\_var)

This creates a new instance of the **Parent** class and accesses the **parent\_var** attribute.

**46. Build a two 2 D array. Plot it using matplotlib.**

import numpy as np

import matplotlib.pyplot as plt

arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

plt.imshow(arr, cmap='viridis')

plt.colorbar()

plt.show()

In this example, we first import the NumPy and Matplotlib libraries. Then, we create a 2D array using the **np.array()** function. This array has three rows and three columns, with the values 1 through 9. Finally, we use Matplotlib's **imshow()** function to plot the array as an image. We also add a colorbar to show the range of values in the array, and use the **show()** function to display the plot.

You can adjust the values in the array to create different patterns and images. Additionally, you can change the color scheme of the plot by adjusting the **cmap** parameter of the **imshow()** function.

**47. Analyse the steps to create a 1D array and 2D array.**

**Creating a 1D Array**

1. Import the NumPy library using the following code:

import numpy as np

2. Create a list of values that you want to include in the array, like this:

my\_list = [1, 2, 3, 4, 5]

3. Use the **np.array()** function to create a NumPy array from the list:

my\_array = np.array(my\_list)

4. You now have a 1D array with the values **[1, 2, 3, 4, 5]**. You can print it using the following code:

print(my\_array)v

Output:

**[1 2 3 4 5]**

**Creating a 2D Array**

1. Import the NumPy library using the following code:

import numpy as np

2. Create a list of lists, where each inner list represents a row in the array. Each inner list should have the same number of elements, like this

my\_list = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

3. Use the **np.array()** function to create a NumPy array from the list:

my\_array = np.array(my\_list)

4. You now have a 2D array with the values:

[[1 2 3]

[4 5 6]

[7 8 9]]

You can print it using the following code:

print(my\_array)

Output:

[1 2 3]

[4 5 6]

[7 8 9]]

**48. Write a program that uses \* in import.**

**• from Module\_Imp3 import \***

**• Take integer as input from user and store it in the variable side.**

**• Call the function calculatearea(side, side)**

**• Call the function pivalue()**

**• print the third element in shapes.**

**• Note: The Module\_Imp3.py already is written.**

from Module\_Imp3 import \*

side = int(input("Enter the side length of the square: "))

area = calculatearea(side, side)

print("Area of square:", area)

pi = pivalue()

print("Value of pi:", pi)

print("Third element of shapes:", shapes[2])

In this example, we import all the functions and variables from Module\_Imp3 using the **\*** operator. We then take an integer input from the user and store it in the **side** variable. We call the **calculatearea()** function with the **side** variable as arguments to calculate the area of a square. We then call the **pivalue()** function to get the value of pi. Finally, we print the third element of the **shapes** list.

**49. Follow the given instructions while writing the program**

**• Use the Module\_Imp3 which contains functions that can be imported.**

**• Use from Module\_Imp3 import \***

**• Take an integer as input from user and store it in the variable side.**

**• Call the function calculatearea(side,side)**

**• Call the function calculatediameter(side)**

**• Call the function pivalue()**

**• print shapes[1:2]**

from Module\_Imp3 import \*

# take input from user

side = int(input("Enter the side length of the square: "))

# calculate area and diameter

area = calculatearea(side, side)

diameter = calculatediameter(side)

# get pi value

pi = pivalue()

# print results

print("Area of square:", area)

print("Diameter of square:", diameter)

print("Value of pi:", pi)

print("Shapes slice [1:2]:", shapes[1:2])

In this program, we first import all functions and variables from **Module\_Imp3** using the **\*** operator. We then take an integer input from the user and store it in the variable **side**.

We call the **calculatearea()** function with **side** as arguments to calculate the area of the square. We then call the **calculatediameter()** function with **side** as an argument to calculate the diameter of the square.

Next, we call the **pivalue()** function to get the value of pi.

Finally, we print the results of the area, diameter, pi, and a slice of the **shapes** list (in this case, the element at index 1). Note that the exact output of this program will depend on the implementation of the functions and variables in **Module\_Imp3**

**50. Write a simple program followed by the instructions given below:**

**• A base class Person and a derived class Student with Person as its base class.**

**• Add two methods setname() (which takes the parameter self and name)and getname() which prints the name in the base class.**

**• Add two methods in the derived class: setage() (which takes the parameters self and age) which sets the age and getage() which prints the age.**

**• Create an instance of Student and name it as s1.**

**• Take name and age as inputs from the console.**

**• Call the setname() and setage() on this instance by passing the name and age parameters.**

**• Call the getname() and getage() on this class, which prints the passed parameters**

class Person:

def setname(self, name):

self.name = name

def getname(self):

print("Name:", self.name)

class Student(Person):

def setage(self, age):

self.age = age

def getage(self):

print("Age:", self.age)

s1 = Student()

name = input("Enter name: ")

age = input("Enter age: ")

s1.setname(name)

s1.setage(age)

s1.getname()

s1.getage()

In this program, we define a base class **Person** with methods **setname()** and **getname()**. We then define a derived class **Student** with a method **setage()** and **getage()**. **Student** inherits from **Person**, so it has access to the **setname()** and **getname()** methods as well.

We create an instance of **Student** called **s1**. We then take input from the console for the **name** and **age** variables. We call **setname()** and **setage()** on **s1** to set the **name** and **age** attributes.

Finally, we call **getname()** and **getage()** on **s1** to print out the **name** and **age** attributes. Note that the exact output of this program will depend on the input provided by the user.