**Question Bank for ETE(Programming In Python)**

1. Identify how python language is more simple other than Java and C. Give proper example.

**ANS:-**

Python is often considered simpler than Java and C due to its clean and concise syntax, which allows for more straightforward code and easier readability. Here are a few examples highlighting Python's simplicity compared to Java and C:

Syntax: Python has a simpler syntax compared to Java and C. For instance, Python doesn't require semicolons at the end of each line or braces to denote code blocks. Here's an example of a simple "Hello, World!" program in Python, Java, and C:

Dynamic Typing: Python is dynamically typed, which means you don't need to explicitly declare variable types. This flexibility makes Python code more concise. In Java and C, you must declare variable types explicitly.

Built-in Features: Python provides extensive built-in functionality and libraries that simplify programming tasks. For instance, Python's standard library includes modules for handling file I/O, regular expressions, networking, and more. This eliminates the need for writing complex code from scratch, making development faster and easier.

Rapid Prototyping: Python's simplicity and readability make it an excellent choice for rapid prototyping and quick development. With fewer lines of code and straightforward syntax, you can quickly translate ideas into working programs.

1. Explain the working of random.seed() function.

**ANS:-**

The `random.seed()` function is part of the Python `random` module, which is used to generate random numbers. The purpose of `random.seed()` is to initialize the random number generator with a specific seed value.

When you call `random.seed()`, it sets the seed for the random number generator, ensuring that subsequent calls to random functions produce the same sequence of random numbers. This is useful when you want to generate the same set of random numbers for testing, debugging, or reproducibility purposes.

The `random.seed()` function takes an optional parameter, `seed`, which is an integer value. This seed value determines the starting point for the random number generator algorithm. If you don't provide a seed value, the current system time is used as the default seed.

Here's an example to illustrate the working of `random.seed()`:

```python

import random

random.seed(42) # Set the seed to 42

print(random.random()) # Generate a random float between 0 and 1

print(random.randint(1, 10)) # Generate a random integer between 1 and 10

random.seed(42) # Set the seed to 42 again

print(random.random()) # Generate a random float (same as before)

print(random.randint(1, 10)) # Generate a random integer (same as before)

```

In this example, we set the seed to 42 using `random.seed(42)`. As a result, every time we run the code, we'll get the same sequence of random numbers. If we were to change the seed value or omit the `random.seed()` call, we would get a different sequence of random numbers.

By controlling the seed value, you can ensure that your random numbers are reproducible and consistent across different runs of your program. This can be particularly useful when you need to debug or share your code with others while maintaining consistent results.

1. (a) Apply a comprehensive explanation of pickle module in Python and list out advantages and limitations of using pickle to write binary files in Python.

**ANS:-**

The `pickle` module in Python is used for serializing and deserializing Python objects. It allows you to convert complex Python objects into a binary representation that can be stored in a file or transferred over a network. This process is called pickling. The resulting binary data can later be unpickled to recreate the original Python object.

Here's an overview of the key functions provided by the `pickle` module:

- `pickle.dump(obj, file)`: Serializes the Python object `obj` and writes the resulting binary data to the file-like object `file`.

- `pickle.load(file)`: Reads the binary data from the file-like object `file` and returns the reconstructed Python object.

Advantages of using `pickle` to write binary files in Python:

1. Ease of use: The `pickle` module provides a simple and convenient way to serialize and deserialize complex Python objects. It handles the underlying serialization details automatically, allowing you to focus on your object data.

2. Object preservation: Pickling preserves the complete state of an object, including its data, attributes, and even its class structure. This makes it suitable for storing and transferring complex data structures or custom objects.

3. Cross-compatibility: Pickle files can be read by any other Python program, regardless of the platform or Python version used. It provides a standardized format for object serialization within the Python ecosystem.

4. Support for custom classes: `pickle` can handle user-defined classes and their instances without any additional effort. It automatically saves and restores the object's state, including its methods and attributes.

Limitations of using `pickle` to write binary files in Python:

1. Limited compatibility across Python versions: Pickle files created with one version of Python may not be readable by a different version due to changes in the serialization format. It is generally recommended to unpickle data using the same Python version that was used to pickle it.

2. Security risks: Unpickling data from untrusted or malicious sources can lead to security vulnerabilities. Maliciously crafted pickle files can execute arbitrary code during the unpickling process, potentially compromising the system. It's crucial to only unpickle data from trusted sources.

3. Inefficiency and file size: Pickle files can be larger compared to other file formats due to the overhead of storing the complete object state. Additionally, pickling and unpickling operations can be slower compared to other serialization methods, especially for large or complex objects.

4. Limited interoperability: `pickle` is specific to Python and cannot be used to exchange data with applications written in other programming languages. If you need to communicate data across different platforms or programming languages, alternative serialization formats like JSON or protocol buffers may be more suitable.

When using `pickle` to write binary files in Python, it's essential to consider these advantages and limitations to ensure proper usage and mitigate any potential risks.

(b) Explain what is range() function and how it is used in lists?

**ANS:-**

The range() function in Python is used to generate a sequence of numbers. It returns an immutable sequence object of numbers that can be iterated over or used to create other data structures like lists.

The range() function takes up to three arguments: start, stop, and step. The start argument specifies the starting value of the sequence (inclusive), the stop argument specifies the ending value of the sequence (exclusive), and the step argument determines the increment between consecutive values. If not provided, the default values are start=0 and step=1.

The range() function generates a sequence of numbers ranging from start to stop - 1, incrementing by step. The resulting sequence is commonly used in for loops to iterate over a specific range of numbers.

Code:

my\_list = list(range(1, 10, 2))

print(my\_list)

Output:

[1, 3, 5, 7, 9]

4. List the operators that python supports. Explain the relational and logical operators along with their precedence while evaluating an expression.

**ANS:-**

Python supports a wide range of operators for performing various operations on different data types. Here is a list of operators supported by Python:

1. Arithmetic Operators:

- `+` (Addition)

- `-` (Subtraction)

- `\*` (Multiplication)

- `/` (Division)

- `%` (Modulo - remainder of division)

- `//` (Floor Division - division with the result rounded down to the nearest whole number)

- `\*\*` (Exponentiation)

2. Assignment Operators:

- `=` (Simple assignment)

- `+=` (Add and assign)

- `-=` (Subtract and assign)

- `\*=` (Multiply and assign)

- `/=` (Divide and assign)

- `%=` (Modulo and assign)

- `//=` (Floor divide and assign)

- `\*\*=` (Exponentiate and assign)

3. Comparison (Relational) Operators:

- `==` (Equal to)

- `!=` (Not equal to)

- `>` (Greater than)

- `<` (Less than)

- `>=` (Greater than or equal to)

- `<=` (Less than or equal to)

4. Logical Operators:

- `and` (Logical AND)

- `or` (Logical OR)

- `not` (Logical NOT)

5. Bitwise Operators:

- `&` (Bitwise AND)

- `|` (Bitwise OR)

- `^` (Bitwise XOR)

- `~` (Bitwise NOT)

- `<<` (Left shift)

- `>>` (Right shift)

6. Membership Operators:

- `in` (Checks if a value exists in a sequence)

- `not in` (Checks if a value does not exist in a sequence)

7. Identity Operators:

- `is` (Checks if two variables refer to the same object)

- `is not` (Checks if two variables refer to different objects)

The precedence of operators determines the order in which they are evaluated when an expression contains multiple operators. Here is the precedence of relational and logical operators (from highest to lowest):

1. Relational Operators:

- `>`, `<`, `>=`, `<=` (Comparison operators have equal precedence)

- `==`, `!=` (Comparison operators have equal precedence)

2. Logical Operators:

- `not`

- `and`

- `or`

When evaluating an expression, operators with higher precedence are evaluated first. If multiple operators have the same precedence, they are evaluated from left to right. Parentheses can be used to explicitly specify the evaluation order and override the default precedence.

For example, consider the expression `5 > 3 and 2 < 4 or not (1 == 1)`. Here's how the operators are evaluated:

1. Parentheses are evaluated first: `1 == 1` is `True`.

2. Then the logical operators are evaluated:

- `5 > 3` is `True`.

- `2 < 4` is `True`.

- `not True` is `False`.

3. Finally, the logical operators `and` and `or` are evaluated:

- `True and True` is `True`.

- `True or False` is `True`.

So, the result of the expression `5 > 3 and 2 < 4 or not (1 == 1)` is `True`.

5. (a) Show the value of L after you run the code below?

L = ["life", "answer", 42, 0]

for thing in L:

if thing == 0:

L[thing] = "universe"

elif thing == 42:

L[1] = "everything"

**ANS:-**['universe', 'everything', 42, 0]

(b) Show the value of L3 after you execute all the operations in the code below?

L1 = ['re']

L2 = ['mi']

L3 = ['do']

L4 = L1 + L2

L3.extend(L4)

L3.sort()

del(L3[0])

L3.append(['fa','la'])

**ANS:-** ['mi', 're', ['fa', 'la']]

1. Categorize and discuss the types of Polymorphism in details with proper example.

**ANS:**-Polymorphism is a powerful concept in object-oriented programming where an object can take on many forms or have multiple behaviors. In Python, there are three main types of polymorphism:

1. Duck typing polymorphism: This type of polymorphism allows objects of different classes to be used interchangeably(exchangeable) if they have the same methods or attributes. The focus is on the behavior of the object rather than its actual type.

Example:-

class Dog:

def sound(self):

print("Bark")

class Cat:

def sound(self):

print("Meow")

def make\_sound(animal):

animal.sound()

dog = Dog()

cat = Cat()

make\_sound(dog) # Output: Bark

make\_sound(cat) # Output: Meow

In the example above, the Dog and Cat classes have the same method name sound(). The make\_sound() function can accept any object that has a sound() method and invoke it, irrespective of the object's actual type.

1. Method Overloading polymorphism: This type of polymorphism allows a class to have multiple methods with the same name but different parameters. The appropriate method is chosen based on the number or types of arguments passed during the function call.

Example:-

class MathOperations:

def add(self, a, b):

return a + b

def add(self, a, b, c):

return a + b + c

math = MathOperations()

print(math.add(2, 3)) # Output: TypeError: add() missing 1 required positional argument: 'c'

print(math.add(2, 3, 4)) # Output: 9

In the example above, the MathOperations class has two methods named add(). The appropriate method is called based on the number of arguments passed. If only two arguments are passed, it results in a TypeError because the method with three arguments is not defined for that case.

3.Operator Overloading polymorphism: This type of polymorphism allows operators like +, -, \*, etc. to be used with objects of different classes in a meaningful way by defining special methods for those operators.

Example:

class Vector:

def \_\_init\_\_(self, x, y):

self.x = x

self.y = y

def \_\_add\_\_(self, other):

return Vector(self.x + other.x, self.y + other.y)

def \_\_str\_\_(self):

return f"({self.x}, {self.y})"

v1 = Vector(2, 3)

v2 = Vector(4, 5)

v3 = v1 + v2

print(v3) # Output: (6, 8)

In the example above, the Vector class defines the \_\_add\_\_() method which is called when the + operator is used with objects of the Vector class. It allows for adding two vectors element-wise, resulting in a new Vector object.

1. (a) Analyze a Numpy array filled with all zeros.

**ANS :**- 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.

(b) Analyze reverse a Numpy array.

**Ans:**-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.

8. Assuming the instructions given below,write a simple program using a class()

* 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 9.

**ANS:-**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()

1. Apply 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.

to import the abs() function from the built-ins module, display its documentation, and find the absolute value of -155, you can use the following code:

import builtins

# Importing abs() function

abs\_func = builtins.abs

# Displaying documentation

print(abs\_func.\_\_doc\_\_)

# Finding absolute value

result = abs\_func(-155)

print("Absolute value:", result)

The builtins module is imported to access the abs() function. Then, we assign the abs() function to a variable abs\_func. The \_\_doc\_\_ attribute is used to display the documentation of the abs() function. Finally, we call abs\_func(-155) to find the absolute value of -155 and print the result.

1. (a) Assume fruits = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango") write the print statement using a range of indexes to print the third, fourth, and fifth item in the tuple.

(a) Here's the print statement using a range of indexes to print the third, fourth, and fifth items in the tuple:

```python

fruits = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")

print(fruits[2:5])

```

Output:

```

('cherry', 'orange', 'kiwi')

```

Note: The range of indexes is inclusive of the starting index (2) and exclusive of the ending index (5). So, the third, fourth, and fifth items are printed.

(b) Assume fruits = ("apple", "banana", "cherry") write the python code using negative indexing to print the last item in the tuple.

(b) Here's the Python code using negative indexing to print the last item in the tuple:

CODE:

fruits = ("apple", "banana", "cherry")

print(fruits[-1])

Output:

cherry

Negative indexing allows you to access elements from the end of the tuple. So, `fruits[-1]` refers to the last item in the tuple.

1. A permutation is simply a name for a reordering. So the permutations of the string „abc‟ are „abc‟, „acb‟, „bac‟, „bca‟, „cab‟, and „cba‟. Note that a sequence is a permutation of itself (the trivial permutation). Take the permutation of the string in the list and write python program for a recursive function get\_permutations) that takes a string and returns a list of all its permutations.

**ANS:-**

def get\_permutations(string):

# Base case: If the string is empty or contains only one character, return it as a single permutation

if len(string) <= 1:

return [string]

# List to store all permutations

permutations = []

# Iterate through each character in the string

for i in range(len(string)):

# Extract the current character

current\_char = string[i]

# Generate all permutations of the remaining characters (excluding the current character)

remaining\_chars = string[:i] + string[i + 1:]

sub\_permutations = get\_permutations(remaining\_chars)

# Append the current character to the permutations of the remaining characters

for permutation in sub\_permutations:

permutations.append(current\_char + permutation)

# Return the list of permutations

return permutations

# Test the function

string = "abc"

permutations = get\_permutations(string)

print(permutations)

OUTPUT:-['abc', 'acb', 'bac', 'bca', 'cab', 'cba']

The get\_permutations function takes a string as input and follows these steps:

If the string has a length of 0 or 1, it returns the string as a single permutation.

Otherwise, it initializes an empty list permutations to store all permutations.

It iterates through each character in the string.

For each character, it extracts the current character and generates all permutations of the remaining characters (recursively calling get\_permutations).

It appends the current character to each permutation of the remaining characters and adds them to the permutations list.

Finally, it returns the list of permutations.

In the provided example, the string "abc" will generate all its permutations: 'abc', 'acb', 'bac', 'bca', 'cab', and 'cba'.

1. Explain different rules about to define an identifier in python. If the age of Ram, Sam, and Khan are input through the keyboard, write a python program to determine the eldest and youngest of the three.

**ANS:-**In Python, there are certain rules for defining identifiers, which are names used to identify variables, functions, classes, modules, etc. Here are the rules for defining identifiers in Python:

1. An identifier can contain letters (both uppercase and lowercase), digits, and underscores (\_).

2. The first character of an identifier cannot be a digit. It must be a letter or an underscore.

3. Python is case-sensitive, so uppercase and lowercase letters are considered different. For example, "myVar" and "myvar" are two different identifiers.

4. The length of an identifier can be as long as needed.

5. Python keywords cannot be used as identifiers. Keywords are reserved words that have special meanings in Python, such as "if," "for," "while," etc.

Now, let's write a Python program that takes the age of Ram, Sam, and Khan as input and determines the eldest and youngest among them:

Code:

# Input age of Ram, Sam, and Khan

age\_ram = int(input("Enter the age of Ram: "))

age\_sam = int(input("Enter the age of Sam: "))

age\_khan = int(input("Enter the age of Khan: "))

# Determine the eldest and youngest

eldest = max(age\_ram, age\_sam, age\_khan)

youngest = min(age\_ram, age\_sam, age\_khan)

# Print the result

print("The eldest person is", eldest, "years old.")

print("The youngest person is", youngest, "years old.")

In this program, we take the ages of Ram, Sam, and Khan as input using the `input` function. The `int` function is used to convert the input to integers. Then, we use the `max` and `min` functions to determine the eldest and youngest ages among the three. Finally, we print the results using the `print` function.

1. Write a Python program to find the exponentiation of a number.

**ANS:**-base = float(input("Enter the base number: "))

exponent = int(input("Enter the exponent: "))

result = base \*\* exponent

print("The result of", base, "raised to the power of", exponent, "is:", result)

15. Identify the string method used to implement the following.

I. To count the number of characters in the string.

**ANS:-**I. The string method used to count the number of characters in a string is len(). However, len() is not a string method, but a built-in Python function that returns the length of a sequence, including strings. Here's an example:

Code

my\_string = "Hello, world!"

length = len(my\_string)print("The number of characters in the string is:", length)

Output:

The number of characters in the string is: 13

In the above example, the len() function is used to determine the length of the string my\_string, which is the number of characters in the string.

II. To change the first character of the string in capital letter.

**ANS:-** The string method used to change the first character of the string to a capital letter is capitalize(). Here's an example:

Code:

my\_string = "hello, world!"

capitalized\_string = my\_string.capitalize()

print("Original string:", my\_string)

print("Capitalized string:", capitalized\_string)

Output:

Original string: hello, world!

Capitalized string: Hello, world!

In the above example, the capitalize() method is called on the string my\_string, resulting in a new string capitalized\_string where the first character is capitalized. The original string remains unchanged.

16.Define SciPy, Scrapy, Scikit-learn, PyGame, PyTorch, PyBrain and Keras.

**ANS:-**  Here are definitions for SciPy, Scrapy, Scikit-learn, PyGame, PyTorch, PyBrain, and Keras:

1. SciPy: SciPy is a scientific computing library for Python. It provides a wide range of mathematical algorithms and functions for tasks such as numerical integration, optimization, signal processing, linear algebra, and more. It is built on top of NumPy and is often used in fields such as physics, engineering, and data science.

2. Scrapy: Scrapy is a Python framework used for web scraping and crawling. It provides a set of tools and APIs that make it easier to extract data from websites and automate the process of navigating through web pages. Scrapy allows you to define spiders, which are custom classes that specify how to interact with websites and extract desired information.

3. Scikit-learn: Scikit-learn, also known as sklearn, is a popular machine learning library for Python. It provides a wide range of algorithms and tools for tasks such as classification, regression, clustering, dimensionality reduction, model selection, and more. Scikit-learn is designed to be user-friendly and integrates well with other scientific computing libraries such as NumPy and SciPy.

4. PyGame: PyGame is a cross-platform library used for developing video games and multimedia applications in Python. It provides functionality for handling graphics, sound, input devices, and more. PyGame is built on top of the Simple DirectMedia Layer (SDL) library, which allows developers to create interactive games and simulations.

5. PyTorch: PyTorch is an open-source machine learning framework primarily developed by Facebook's AI Research lab (FAIR). It provides a flexible and efficient way to build and train deep learning models. PyTorch supports dynamic computation graphs, making it easy to modify models on the fly. It has gained popularity in the research community and is widely used for tasks such as image recognition, natural language processing, and reinforcement learning.

6. PyBrain: PyBrain is a modular machine learning library for Python that focuses on neural networks and reinforcement learning. It provides a set of tools and algorithms for building and training various types of neural networks, including feedforward networks, recurrent networks, and self-organizing maps. PyBrain also includes reinforcement learning algorithms for tasks such as temporal difference learning and Q-learning.

7. Keras: Keras is a high-level neural networks API written in Python. It is designed to be user-friendly, modular, and extensible. Keras provides a simple and intuitive interface for building and training deep learning models. It can be used on top of other machine learning frameworks such as TensorFlow and Theano. Keras simplifies the process of developing deep learning models by providing high-level abstractions and pre-built components.

These libraries and frameworks offer different functionalities and are widely used in various domains, including scientific computing, web scraping, machine learning, game development, and deep learning.

17. (a) Show the output of the following Python code?

d = {"john":40, "peter":45} d["john"]

**ANS:- 40**

In the code, the dictionary d is defined with two key-value pairs. When we access d["john"], it retrieves the value associated with the key "john" in the dictionary, which is 40. Finally, the value 40 is printed to the console.

b) Is tuple comparison possible? Explain how with example.

**ANS:-** Yes, tuple comparison is possible in Python. Tuples can be compared using the comparison operators such as `==`, `!=`, `<`, `>`, `<=`, and `>=`.

When comparing tuples, the elements are compared element-wise, starting from the first element. If the elements at corresponding positions in the tuples are equal, the comparison moves on to the next element. The comparison stops as soon as a mismatch is found, and the result is determined based on the comparison of the mismatched elements.

Here's an example to demonstrate tuple comparison:

Code:

tuple1 = (1, 2, 3)

tuple2 = (1, 2, 4)

if tuple1 < tuple2:

print("tuple1 is less than tuple2")

elif tuple1 > tuple2:

print("tuple1 is greater than tuple2")

else:

print("tuple1 and tuple2 are equal")

Output:

```

tuple1 is less than tuple2

```

In the example above, we have two tuples, `tuple1` and `tuple2`. We compare them using the `<` operator. The comparison starts from the first element, and since `1` in `tuple1` is less than `1` in `tuple2`, the condition `tuple1 < tuple2` evaluates to `True`, and the corresponding message is printed.

If all elements are equal up to a certain point, the comparison moves on to the next element. For example, `(1, 2, 3)` would be considered less than `(1, 2, 4)` because `3` is less than `4`, even though the first two elements are the same.

Tuple comparison can be useful in various scenarios, such as sorting tuples, finding minimum and maximum values, or when implementing custom comparison logic based on specific tuple elements.

18. Assume 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]

**ANS:-** from Module\_Imp3 import \*

# Take an integer as input from the user and store it in the variable 'side'

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

# Call the function calculatearea(side, side)

area = calculatearea(side, side)

print("Area:", area)

# Call the function calculatediameter(side)

diameter = calculatediameter(side)

print("Diameter:", diameter)

# Call the function pivalue()

pi = pivalue()

print("Value of pi:", pi)

# Print shapes[1:2]

shapes = ["circle", "square", "triangle", "rectangle"]

print("Selected shape:", shapes[1:2])

In this program, the from Module\_Imp3 import \* statement imports all functions from the Module\_Imp3 module.

The program prompts the user to enter an integer as the side length and stores it in the variable side.

Next, it calls the function calculatearea(side, side) to calculate the area of a square with the given side length. The calculated area is stored in the variable area and then printed.

The function calculatediameter(side) is called to calculate the diameter of a circle with the given side length. The calculated diameter is stored in the variable diameter and printed.

The function pivalue() is called to retrieve the value of pi. The returned value is stored in the variable pi and printed.

Finally, the program prints the selected shape from the shapes list using slicing. In this case, shapes[1:2] will print the second element of the list, which is "square".

Please note that the exact implementation of the functions and the contents of Module\_Imp3 are not provided, so the actual results may vary depending on the specific implementation.

19. Develop a python program that asks users to enter their percentage mark for a module of study. The program prints the module grade as either distinction, merit, pass or fail

depending on the percentage mark entered.

I. A mark of 70% and above is awarded a distinction.

II. A mark in the range of 60% through to 69% is awarded a merit.

III.A mark in the range of 40% through to 59% is awarded a pass.

IV.Marks less than 40% are awarded a fail19. Analyze 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.

**ANS:-**  Certainly! Here's a Python program that asks users to enter their percentage mark for a module of study and prints the corresponding module grade:

```python

percentage = float(input("Enter your percentage mark: "))

if percentage >= 70:

grade = "Distinction"

elif percentage >= 60:

grade = "Merit"

elif percentage >= 40:

grade = "Pass"

else:

grade = "Fail"

print("Module Grade:", grade)

```

In this program, the user is prompted to enter their percentage mark using the `input` function. The input is converted to a floating-point number using `float`.

The program then uses a series of `if-elif-else` statements to determine the module grade based on the percentage mark entered.

If the percentage is 70% or above, the grade is set to "Distinction". If the percentage is in the range of 60% to 69%, the grade is set to "Merit". If the percentage is in the range of 40% to 59%, the grade is set to "Pass". For any percentage less than 40%, the grade is set to "Fail".

Finally, the program prints the module grade using the `print` function.

For example, if a user enters a percentage mark of 82%, the program will output:

```

Module Grade: Distinction

```

If a user enters a percentage mark of 58%, the program will output:

```

Module Grade: Pass

```

Regarding the Python class with two methods, `get\_String` and `print\_String`, here's an example implementation:

```python

class StringManipulator:

def get\_String(self):

self.string = input("Enter a string: ")

def print\_String(self):

print(self.string.upper())

# Example usage

obj = StringManipulator()

obj.get\_String()

obj.print\_String()

```

In this class, we define two methods.

The `get\_String` method prompts the user to enter a string using the `input` function and stores it in the instance variable `self.string`.

The `print\_String` method prints the string in uppercase using the `upper` method of strings. The `self.string` variable is accessed within the class to retrieve the stored string.

To use the class, we create an instance `obj` of the `StringManipulator` class. We can then call the `get\_String` method to input a string and the `print\_String` method to print it in uppercase.

Please note that the class provided is a basic example, and additional error handling or modifications can be made based on specific requirements.

20. List below are the following conditions to write a program to display only those numbers

(a) The number must be divisible by five

(b) If the number is greater than 150, then skip it and move to the next number

(c) If the number is greater than 500, then stop the loop

(d) Input: numbers = [12, 75, 150, 180, 145, 525, 50]

(e) Output:

75

145

150

**ANS:-**  To display only those numbers that satisfy the given conditions, you can use a loop to iterate through the numbers and apply the conditions using `if` statements. Here's a Python program that achieves this:

```python

numbers = [12, 75, 150, 180, 145, 525, 50]

for num in numbers:

if num % 5 == 0: # Condition (a): The number must be divisible by five

if num > 150: # Condition (b): Skip if number is greater than 150

continue

if num > 500: # Condition (c): Stop loop if number is greater than 500

break

print(num)

```

Output:

```

75

145

150

```

In the above program, we iterate through each number in the `numbers` list using a `for` loop.

Inside the loop, we first check if the number is divisible by 5 using the condition `num % 5 == 0`. If this condition is satisfied, we proceed to the next condition.

If the number is greater than 150, we use the `continue` statement to skip the current iteration and move to the next number, without executing the remaining code in the loop for that iteration.

If the number is greater than 500, we use the `break` statement to exit the loop, stopping further iterations.

Finally, if none of the previous conditions are met, we print the number using the `print` statement.

Based on the given input, the program will output `75`, `145`, and `150`, as these are the numbers that satisfy the conditions.

1. (a) Identify the steps to create a 1D array and 2D array.

**ANS:-**

To create a 1D array (also known as a vector or a one-dimensional array) and a 2D array (also known as a matrix or a two-dimensional array) in Python, you can use different approaches. Here are the general steps for creating each type of array:

Creating a 1D array:

1. Import the necessary libraries. In Python, you can use the `numpy` library to work with arrays efficiently. So, you will need to import it using `import numpy as np`.

2. Define the elements of the array. Decide on the values that you want to include in the 1D array.

3. Create the array using the `numpy.array` function. Pass the elements as a list or tuple to the `numpy.array` function and store the result in a variable.

Here's an example that demonstrates these steps:

```python

import numpy as np

# Define the elements of the 1D array

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

# Create the 1D array

arr\_1d = np.array(elements)

# Print the 1D array

print(arr\_1d)

```

Output:

```

[1 2 3 4 5]

```

Creating a 2D array:

1. Import the necessary libraries. Again, you will need to import `numpy` using `import numpy as np`.

2. Define the elements of the 2D array. Decide on the values that you want to include in the 2D array and organize them into rows and columns.

3. Create the 2D array using the `numpy.array` function. Pass the elements as a nested list or nested tuple to the `numpy.array` function and store the result in a variable.

Here's an example that demonstrates these steps:

```python

import numpy as np

# Define the elements of the 2D array

elements = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

# Create the 2D array

arr\_2d = np.array(elements)

# Print the 2D array

print(arr\_2d)

```

Output:

```

[[1 2 3]

[4 5 6]

[7 8 9]]

```

In this example, the `elements` list represents a 2D array with three rows and three columns. The `np.array` function creates the 2D array `arr\_2d`, which is then printed using the `print` statement.

Note that the `numpy` library provides many more options and functionalities for working with arrays, such as specifying data types, reshaping arrays, performing mathematical operations, and more.

(b) Apply Python programming to compute the row wise counts of all possible values in an array.

**ANS:-**  g To compute the row-wise counts of all possible values in a NumPy array in Python, you can use the `numpy.unique` function along with the `numpy.apply\_along\_axis` function. Here's an example that demonstrates this:

```python

import numpy as np

# Create a sample 2D array

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

[2, 3, 1],

[1, 1, 2]])

# Define a function to compute row-wise value counts

def row\_value\_counts(row):

unique\_values, counts = np.unique(row, return\_counts=True)

return dict(zip(unique\_values, counts))

# Compute row-wise value counts using apply\_along\_axis

row\_counts = np.apply\_along\_axis(row\_value\_counts, axis=1, arr=arr)

# Print the row-wise value counts

print(row\_counts)

```

Output:

```

[{1: 1, 2: 1, 3: 1},

{1: 1, 2: 1, 3: 1},

{1: 2, 2: 1}]

```

In this example, we first create a sample 2D array `arr` using `numpy.array`. The array contains integer values.

Next, we define a function `row\_value\_counts` that takes a row as input. Within this function, we use `numpy.unique` with `return\_counts=True` to obtain the unique values and their corresponding counts in the row. We return the counts as a dictionary, where the keys represent the unique values and the values represent their counts.

We then use `numpy.apply\_along\_axis` to apply the `row\_value\_counts` function row-wise to the `arr` array. We specify `axis=1` to indicate that we want to apply the function row-wise. The resulting row-wise value counts are stored in the `row\_counts` variable.

Finally, we print the `row\_counts` to display the row-wise value counts for each row in the array.

Note that the `row\_counts` variable is a NumPy array, where each element corresponds to the row-wise value counts represented as a dictionary.

1. (a) Compare the differences between a python dictionary and a python set, including the syntax, parameters, and output of each.

**ANS:-**

In Python, dictionaries and sets are both built-in data structures, but they have different purposes and characteristics. Let's compare the differences between a Python dictionary and a Python set in terms of syntax, parameters, and output:

1. Syntax:

- Dictionary: A dictionary is defined using curly braces `{}` and consists of key-value pairs separated by colons. For example: `my\_dict = {'key1': value1, 'key2': value2}`

- Set: A set is defined using curly braces `{}` or the `set()` function, and it contains unique elements separated by commas. For example: `my\_set = {element1, element2}` or `my\_set = set([element1, element2])`

2. Parameters:

- Dictionary: A dictionary stores elements as key-value pairs, where each key must be unique. The keys are used to access corresponding values.

- Set: A set stores unique elements, and it does not allow duplicates. It is primarily used for membership testing and removing duplicates from a collection.

3. Output:

- Dictionary: When accessing a value in a dictionary, you provide the corresponding key. The dictionary returns the value associated with that key, or raises a `KeyError` if the key is not found. You can also iterate over the keys or values of a dictionary.

- Set: A set returns the unique elements it contains. You can perform various set operations like union, intersection, and difference. Sets are unordered, so the order of elements in the output may vary.

Here's a summary of the main differences between dictionaries and sets:

- Dictionaries are key-value pairs, while sets are unordered collections of unique elements.

- Dictionaries are accessed using keys, whereas sets are primarily used for membership testing.

- Dictionaries allow any hashable objects as keys, while sets only allow hashable objects.

- Dictionaries store data as key-value pairs, while sets only store individual elements.

- Dictionaries are mutable, meaning their elements can be modified, added, or removed. Sets are also mutable.

(b) Analyze the "fromkeys" method in python dictionaries, including its syntax, parameters, and output.

**ANS:-**

The `fromkeys` method in Python dictionaries is a convenient way to create a new dictionary with specified keys and a default value. It allows you to specify a sequence of keys and a default value, and it returns a new dictionary where each key is mapped to the specified default value.

Here is the syntax of the `fromkeys` method:

```python

new\_dict = dict.fromkeys(keys, value)

```

Parameters:

- `keys` (required): It specifies the keys for the new dictionary. It can be an iterable, such as a list, tuple, or set, containing the keys.

- `value` (optional): It specifies the default value that will be associated with each key in the new dictionary. If not provided, the default value is set to `None`.

Output:

- `new\_dict`: It is the newly created dictionary where each key from the `keys` parameter is mapped to the specified `value` parameter.

Here's an example to illustrate the usage of the `fromkeys` method:

```python

keys = ['a', 'b', 'c']

value = 0

new\_dict = dict.fromkeys(keys, value)

print(new\_dict)

```

Output:

```

{'a': 0, 'b': 0, 'c': 0}

```

In this example, the `fromkeys` method creates a new dictionary `new\_dict` where each key from the `keys` list is mapped to the specified `value` of 0. The output shows the resulting dictionary where all keys `'a'`, `'b'`, and `'c'` are associated with the value 0.

It's important to note that the `fromkeys` method does not modify the existing dictionary. Instead, it creates a new dictionary based on the specified keys and default value.

(c) Write a simple program to convert given number into string, char and hexadecimal and complex number.

**ANS:-**

Certainly! Here's a simple program in Python that takes a number as input and converts it into a string, character, hexadecimal, and complex number:

```python

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

# Convert to string

number\_as\_string = str(number)

print("Number as string:", number\_as\_string)

# Convert to character

number\_as\_char = chr(number)

print("Number as character:", number\_as\_char)

# Convert to hexadecimal

number\_as\_hex = hex(number)

print("Number as hexadecimal:", number\_as\_hex)

# Convert to complex number

number\_as\_complex = complex(number)

print("Number as complex number:", number\_as\_complex)

```

In this program, the user is prompted to enter a number. Then, the program performs the following conversions:

1. Converts the number to a string using `str(number)` and stores it in the variable `number\_as\_string`.

2. Converts the number to a character using `chr(number)` and stores it in the variable `number\_as\_char`.

3. Converts the number to a hexadecimal representation using `hex(number)` and stores it in the variable `number\_as\_hex`.

4. Converts the number to a complex number using `complex(number)` and stores it in the variable `number\_as\_complex`.

Finally, the program displays the converted values for each type.

Note: The program assumes that the user enters a valid number. If you want to add error handling for non-numeric inputs or handle specific ranges of numbers, additional checks and validations can be added to the program.

23. (a) Assume the string "This is my first String". Write a program to print the following:

* print the string, print the character f using forward indexing, and print the character S using negative/backward indexing.

**ANS:-**

Here's a program that prints the given string, accesses the character 'f' using forward indexing, and accesses the character 'S' using negative/backward indexing:

```python

string = "This is my first String"

# Print the string

print("Original string:", string)

# Print the character 'f' using forward indexing

print("Character 'f' using forward indexing:", string[10])

# Print the character 'S' using negative/backward indexing

print("Character 'S' using negative/backward indexing:", string[-7])

```

Output:

```

Original string: This is my first String

Character 'f' using forward indexing: f

Character 'S' using negative/backward indexing: S

```

In this program, the given string is stored in the variable `string`. The program then performs the following tasks:

1. Prints the original string using `print("Original string:", string)`.

2. Accesses the character 'f' using forward indexing by specifying the index 10 (`string[10]`) and prints it.

3. Accesses the character 'S' using negative/backward indexing by specifying the index -7 (`string[-7]`) and prints it.

The program outputs the original string, the character 'f' using forward indexing, and the character 'S' using negative/backward indexing.

(b) Test for two inputs from the user using input() function, one is string str and another one is integer n. Write a program to print the given string str n times. Print the result as shown in the example.

**ANS:-**

Certainly! Here's a program that takes a string and an integer as inputs from the user and prints the given string n times:

```python

# Get user input

string = input("Enter a string: ")

n = int(input("Enter the number of times to repeat the string: "))

# Print the string n times

result = string \* n

print("Result:")

print(result)

```

Example output:

```

Enter a string: Hello

Enter the number of times to repeat the string: 3

Result:

HelloHelloHello

```

In this program, the user is prompted to enter a string using `input("Enter a string: ")`. The string input is stored in the variable `string`.

Then, the user is prompted to enter the number of times they want to repeat the string using `input("Enter the number of times to repeat the string: ")`. The input is converted to an integer using `int()` and stored in the variable `n`.

Next, the program performs string multiplication `string \* n` to repeat the string `n` times, and stores the result in the variable `result`.

Finally, the program prints the resulting string by displaying `result` using `print(result)`.

24. Assume the given below instructions to define a base class Car and a derived class Accord.

* Car class has two methods which sets and gets the method brandname.
* Accord is a derived class of the base class Car, which has two methods which sets and gets the brandname.
* Accord also has the model set and get methods.
* Now create an instance of Accord and set the brandname to the user given input.
* Set the input model for the same instance.
* Now print the output by calling the methods getbrandname() and getmodel() on Accord instance

**ANS:-**

Based on the given instructions, here's an example implementation of the base class `Car` and the derived class `Accord` in Python:

```python

class Car:

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

self.brandname = ""

def set\_brandname(self, brandname):

self.brandname = brandname

def get\_brandname(self):

return self.brandname

class Accord(Car):

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

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

self.model = ""

def set\_model(self, model):

self.model = model

def get\_model(self):

return self.model

# Create an instance of Accord

my\_accord = Accord()

# Set the brandname using user input

brandname = input("Enter the brandname for Accord: ")

my\_accord.set\_brandname(brandname)

# Set the model using user input

model = input("Enter the model for Accord: ")

my\_accord.set\_model(model)

# Print the brandname and model

print("Brandname:", my\_accord.get\_brandname())

print("Model:", my\_accord.get\_model())

```

Example output:

```

Enter the brandname for Accord: Honda

Enter the model for Accord: LX

Brandname: Honda

Model: LX

```

In this program, we define the base class `Car` with methods `set\_brandname` and `get\_brandname` to set and get the brandname.

The derived class `Accord` inherits from the `Car` class and adds additional methods `set\_model` and `get\_model` to set and get the model.

We create an instance of `Accord` called `my\_accord`.

We prompt the user to enter the brandname for the `my\_accord` instance using `input`, and set it using `my\_accord.set\_brandname(brandname)`.

Similarly, we prompt the user to enter the model for the `my\_accord` instance, and set it using `my\_accord.set\_model(model)`.

Finally, we print the brandname and model by calling the `get\_brandname` and `get\_model` methods on the `my\_accord` instance.

1. Recall and discuss different kind of keywords in python. Write a source code in python to explain bitwise operators available in python programming language.

**ANS:-**

In Python, there are several types of keywords that have special meanings and cannot be used as identifiers. Here are some of the different kinds of keywords in Python:

1. Reserved Keywords:

- These keywords are reserved for specific purposes and cannot be used as variable names or function names. Examples include `if`, `else`, `while`, `for`, `def`, `class`, `import`, `from`, `return`, etc.

2. Special Literal Keywords:

- These keywords represent special literals in Python. Examples include `True`, `False`, and `None`.

3. Built-in Function Keywords:

- These keywords are used to define built-in functions or to specify certain behaviors within function definitions. Examples include `print`, `len`, `range`, `input`, `open`, `sorted`, etc.

4. Context Manager Keywords:

- These keywords are used in the context of context managers, which are objects that define the behavior of the `with` statement. Examples include `with` and `as`.

Now, let's move on to the explanation and source code for bitwise operators in Python:

Bitwise operators in Python are used to perform operations on individual bits of binary numbers. These operators are useful in tasks related to low-level programming, network protocols, and optimizing algorithms. Python provides the following bitwise operators:

1. Bitwise AND (`&`): Sets each bit to 1 if both bits are 1.

2. Bitwise OR (`|`): Sets each bit to 1 if at least one of the corresponding bits is 1.

3. Bitwise XOR (`^`): Sets each bit to 1 if exactly one of the corresponding bits is 1.

4. Bitwise NOT (`~`): Flips the bits (complements) of the operand.

5. Left Shift (`<<`): Shifts the bits of the left operand to the left by the number of positions specified by the right operand.

6. Right Shift (`>>`): Shifts the bits of the left operand to the right by the number of positions specified by the right operand, filling the leftmost bits with the sign bit.

Here's an example source code in Python that demonstrates the usage of bitwise operators:

```python

# Example values

a = 10 # 1010 in binary

b = 6 # 0110 in binary

# Bitwise AND

result\_and = a & b

print("Bitwise AND:", result\_and) # Output: 2

# Bitwise OR

result\_or = a | b

print("Bitwise OR:", result\_or) # Output: 14

# Bitwise XOR

result\_xor = a ^ b

print("Bitwise XOR:", result\_xor) # Output: 12

# Bitwise NOT

result\_not\_a = ~a

print("Bitwise NOT of a:", result\_not\_a) # Output: -11

# Left Shift

result\_left\_shift = a << 2

print("Left Shift:", result\_left\_shift) # Output: 40

# Right Shift

result\_right\_shift = a >> 2

print("Right Shift:", result\_right\_shift) # Output: 2

```

In this example, we have two variables `a` and `b` initialized with decimal values. We perform various bitwise operations using the operators mentioned above and print the results.

1. Write a Python program that given number is palindrome or not.

**ANS:-**

Sure! Here's a Python program that checks whether a given number is a palindrome or not:

```python

def is\_palindrome(number):

# Convert the number to a string

number\_str = str(number)

# Reverse the string

reversed\_str = number\_str[::-1]

# Compare the original and reversed strings

if number\_str == reversed\_str:

return True

else:

return False

# Get the input number from the user

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

# Check if the number is a palindrome

if is\_palindrome(number):

print("The number is a palindrome")

else:

print("The number is not a palindrome")

```

In this program, we define a function `is\_palindrome` that takes a number as input and checks if it is a palindrome.

Inside the function:

- The number is converted to a string using `str(number)`.

- The string is reversed using slicing notation `[::-1]`, which creates a reversed copy of the string.

- The original string and the reversed string are compared using an `if` statement.

- If the two strings are equal, the number is a palindrome and the function returns `True`. Otherwise, it returns `False`.

In the main part of the program:

- The user is prompted to enter a number using `input("Enter a number: ")`. The input is converted to an integer using `int()` and stored in the variable `number`.

- The `is\_palindrome` function is called with the input number as an argument.

- Depending on the return value of the function, the program prints whether the number is a palindrome or not.

1. Show how to initialize and access the elements of a tuple by giving a suitable example.

**ANS:-**

Sure! In Python, tuples are ordered, immutable collections of objects. You can initialize a tuple by enclosing comma-separated values within parentheses `()`. You can access individual elements of a tuple using indexing or unpacking. Let's see an example:

```python

# Initializing a tuple

person = ('John', 25, 'USA')

# Accessing elements by indexing

name = person[0]

age = person[1]

country = person[2]

print(name) # Output: John

print(age) # Output: 25

print(country) # Output: USA

# Unpacking a tuple into variables

name, age, country = person

print(name) # Output: John

print(age) # Output: 25

print(country) # Output: USA

```

In the above example, we initialize a tuple named `person` with three elements: the name, age, and country of a person. To access individual elements, we use indexing, where `person[0]` gives the first element (name), `person[1]` gives the second element (age), and `person[2]` gives the third element (country).

Alternatively, we can also unpack the tuple into separate variables, where each variable corresponds to an element of the tuple. In the example, we unpack the `person` tuple into three variables: `name`, `age`, and `country`. This allows us to directly assign each element of the tuple to a separate variable without indexing.

Both indexing and unpacking are valid ways to access elements of a tuple, and you can choose the method that best suits your needs in different scenarios.

1. (a)Define the functions in Matplotlib: (i) title() (ii) xlabel() (iii) ylabel() (iv) legend()

**ANS:-**

Matplotlib is a popular data visualization library in Python. It provides a wide range of functions and methods to create high-quality plots and charts. Here are definitions of four commonly used functions in Matplotlib:

(i) `title()`:

The `title()` function is used to set a title for the plot. It adds a title at the top of the plot to provide a brief description or context for the data being visualized. The title is typically a string that you pass as an argument to the `title()` function.

Example usage:

```python

import matplotlib.pyplot as plt

plt.plot(x, y)

plt.title("Plot of y vs x")

plt.show()

```

In this example, the `title()` function is used to set the title of the plot as "Plot of y vs x".

(ii) `xlabel()`:

The `xlabel()` function is used to set the label for the x-axis of the plot. It adds a descriptive label to the horizontal axis, representing the data being plotted along that axis. The label is typically a string that you pass as an argument to the `xlabel()` function.

Example usage:

```python

import matplotlib.pyplot as plt

plt.plot(x, y)

plt.xlabel("X-axis")

plt.show()

```

In this example, the `xlabel()` function is used to set the label of the x-axis as "X-axis".

(iii) `ylabel()`:

The `ylabel()` function is used to set the label for the y-axis of the plot. It adds a descriptive label to the vertical axis, representing the data being plotted along that axis. The label is typically a string that you pass as an argument to the `ylabel()` function.

Example usage:

```python

import matplotlib.pyplot as plt

plt.plot(x, y)

plt.ylabel("Y-axis")

plt.show()

```

In this example, the `ylabel()` function is used to set the label of the y-axis as "Y-axis".

(iv) `legend()`:

The `legend()` function is used to add a legend to the plot. A legend provides an explanation of the elements displayed in the plot, such as labels for different lines, markers, or other graphical elements. You can specify the labels for the legend using a list of strings, and the location of the legend on the plot can be specified as well.

Example usage:

```python

import matplotlib.pyplot as plt

plt.plot(x, y1, label="Line 1")

plt.plot(x, y2, label="Line 2")

plt.legend(loc="upper right")

plt.show()

```

In this example, the `legend()` function is used to add a legend to the plot, with labels "Line 1" and "Line 2". The `loc` argument specifies the location of the legend, which in this case is set to "upper right".

These functions (`title()`, `xlabel()`, `ylabel()`, and `legend()`) are useful for enhancing the appearance and interpretability of plots created using Matplotlib.

(b) How can you adjust the axis limits of plots in matplotlib?

**ANS:-**

In Matplotlib, you can adjust the axis limits of a plot using the `xlim()` and `ylim()` functions. These functions allow you to set the minimum and maximum values for the x-axis and y-axis, respectively.

Here's how you can adjust the axis limits in Matplotlib:

```python

import matplotlib.pyplot as plt

# Create some example data

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

y = [2, 4, 6, 8, 10]

# Plot the data

plt.plot(x, y)

# Adjust the x-axis limits

plt.xlim(0, 6)

# Adjust the y-axis limits

plt.ylim(0, 12)

# Display the plot

plt.show()

```

In this example, the `xlim()` function is used to set the x-axis limits from 0 to 6, and the `ylim()` function is used to set the y-axis limits from 0 to 12. By specifying the minimum and maximum values for each axis, you can control the visible range of data in the plot.

You can also use the `axis()` function to adjust both the x-axis and y-axis limits simultaneously by providing a list of `[xmin, xmax, ymin, ymax]` values. Here's an example:

```python

import matplotlib.pyplot as plt

# Create some example data

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

y = [2, 4, 6, 8, 10]

# Plot the data

plt.plot(x, y)

# Adjust the x-axis and y-axis limits

plt.axis([0, 6, 0, 12])

# Display the plot

plt.show()

```

In this case, the `axis()` function is used with a list `[0, 6, 0, 12]` to set the x-axis limits from 0 to 6 and the y-axis limits from 0 to 12.

By adjusting the axis limits, you can focus on specific regions of the data or zoom in/out to highlight different aspects of the plot.

(c) What are the various uses of the axis() function in matplotlib?

**ANS:-**

The `axis()` function in Matplotlib is a versatile method that can be used for various purposes related to controlling the axis properties of a plot. It allows you to set or retrieve the current axis limits, control the aspect ratio, and customize the display of axis ticks and labels. Here are some common uses of the `axis()` function:

1. Setting axis limits: You can use `axis()` to set the limits of the x-axis and y-axis simultaneously. For example, `plt.axis([xmin, xmax, ymin, ymax])` sets the limits of the plot to the specified values.

2. Retrieving axis limits: You can retrieve the current axis limits by calling `plt.axis()`. This returns a list of `[xmin, xmax, ymin, ymax]` representing the current limits of the plot.

3. Controlling aspect ratio: The `axis()` function allows you to control the aspect ratio of the plot. By providing the `equal` argument, you can set the aspect ratio to be equal, ensuring that the x-axis and y-axis have the same scale.

4. Setting individual axis limits: You can use `axis()` to set the limits of either the x-axis or y-axis individually. For example, `plt.xlim(xmin, xmax)` sets the x-axis limits, and `plt.ylim(ymin, ymax)` sets the y-axis limits.

5. Auto-scaling the axis limits: By calling `plt.axis('auto')`, Matplotlib automatically adjusts the axis limits to fit the data in the plot. This is useful when you want the plot to dynamically adjust its limits based on the provided data.

6. Turning off axis scaling: You can turn off axis scaling altogether by calling `plt.axis('off')`. This removes the axis lines, ticks, and labels from the plot, giving you a clean visual representation of your data.

7. Setting equal scaling: The `axis()` function allows you to set equal scaling for both the x-axis and y-axis. By calling `plt.axis('equal')`, the plot ensures that the data is displayed with equal scaling in both dimensions, irrespective of the limits.

Overall, the `axis()` function in Matplotlib provides flexibility in managing the axis properties of a plot, allowing you to control the limits, aspect ratio, scaling, and display options. It is a powerful tool for customizing the visual representation of your data.

1. (a) Show a python program to print multiplication table of a given number.

**ANS:-**

Certainly! Here's a Python program that prints the multiplication table of a given number:

```python

def multiplication\_table(number):

for i in range(1, 11):

result = number \* i

print(f"{number} x {i} = {result}")

# Get the number from the user

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

# Print the multiplication table

multiplication\_table(num)

```

In this program, we define a function `multiplication\_table()` that takes a number as an argument. Inside the function, we use a `for` loop to iterate from 1 to 10. In each iteration, we calculate the result of multiplying the given number with the loop variable `i`. We then print the multiplication expression using formatted string interpolation.

Outside the function, we prompt the user to enter a number using the `input()` function, and we convert the input to an integer using `int()`. Finally, we call the `multiplication\_table()` function passing the user's number as an argument, which prints the multiplication table for that number.

Here's an example of running the program and obtaining the multiplication table for the number 5:

```

Enter a number: 5

5 x 1 = 5

5 x 2 = 10

5 x 3 = 15

5 x 4 = 20

5 x 5 = 25

5 x 6 = 30

5 x 7 = 35

5 x 8 = 40

5 x 9 = 45

5 x 10 = 50

```

The program calculates and prints the multiplication table for the entered number, displaying the multiplication expression and the result for each iteration of the loop.

(b) Explain the syntax to read a string from the right-hand side. Consider str = “Python is a wonderful Language”. Reverse and print the str from the right-hand side.

**ANS:-**

1. a) Assume three strings to compare. Using else-if compare then and print the longest string.

**ANS:-**

(b) Assume the string str = “Bangalore”, Iterate it using for loop, and print characters vertically.

**ANS:-**

Sure! Here's a Python program that iterates over the characters of the string "Bangalore" using a `for` loop and prints each character vertically:

```python

str = "Bangalore"

for char in str:

print(char)

```

In this program, we initialize the string `str` with the value "Bangalore". Then, we use a `for` loop to iterate over each character in the string. In each iteration, the current character is stored in the variable `char`. We then print `char` on a separate line using the `print()` function.

When you run this program, it will iterate over the string "Bangalore" and print each character vertically, as follows:

```

B

a

n

g

a

l

o

r

e

```

Each character of the string is printed on a new line, resulting in a vertical representation of the characters.

1. a) Construct a tuple with the user-given elements and concatenate both the tuples and print the result.

**ANS:-**

Certainly! Here's a Python program that constructs a tuple with user-given elements, concatenates it with another tuple, and prints the resulting tuple:

```python

# Get user input for tuple elements

user\_elements = input("Enter elements for the tuple (separated by spaces): ").split()

# Create tuple from user elements

user\_tuple = tuple(user\_elements)

# Another tuple to concatenate

another\_tuple = (1, 2, 3)

# Concatenate both tuples

concatenated\_tuple = user\_tuple + another\_tuple

# Print the result

print("Concatenated Tuple:", concatenated\_tuple)

```

In this program, we first prompt the user to enter the elements for the tuple. The input is split using the `split()` function, which separates the elements based on spaces and returns a list. We then convert this list to a tuple using the `tuple()` function, creating the `user\_tuple` with the user-given elements.

Next, we define another tuple named `another\_tuple` with some pre-defined elements, in this case, `(1, 2, 3)`.

To concatenate both tuples, we use the `+` operator, which concatenates the two tuples, resulting in a new tuple assigned to `concatenated\_tuple`.

Finally, we print the concatenated tuple using the `print()` function.

Here's an example of running the program and entering elements "apple", "banana", "cherry" for the user-given tuple:

```

Enter elements for the tuple (separated by spaces): apple banana cherry

Concatenated Tuple: ('apple', 'banana', 'cherry', 1, 2, 3)

```

The program concatenates the user-given tuple `(apple, banana, cherry)` with the `another\_tuple` `(1, 2, 3)`, resulting in the concatenated tuple `('apple', 'banana', 'cherry', 1, 2, 3)`.

(b) Construct a tuple with the user-given inputs. Write a program using membership operators to check whether the given element is present in the tuple or not. Print the result.

**ANS:-**

(c) Apply a python program to add an element to a tuple based on the user-given value in a specific index, and print the result as shown in the example. If the index is not in the range print the error message as shown in the example.

**ANS:-**

32. a) Construct a python program to change Dictionary keys into values and values into keys, and print the result.

(b)Choose an integer sequence from the user. Write a program to print a dictionary from the given sequence. Consider the element in the sequence as a key, and the number of times the element occurs in the sequence as a value. Print the result.

**ANS:-**

1. a. Identify different methods available in python dictionaries to add, remove, or change elements within a dictionary, including their syntax, parameters, and output.

**ANS:-**

b. Make Use of the "pop" method present in python dictionaries, including its syntax, parameters and output. Identify how it is differ from the "popitem" method, and when would you use one over the other?

**ANS:-**

1. Choose the capitalize() method works in python. Take str1 = “ hello how are you” & str2 = “42 is my lucky number”. Demonstrate the output for both strings.

**ANS:-**

35. Assume the given below instructions to implements the method overriding method:

• Define class Animal

• Use constructor to set the name with a default value = "This Animal"

• Define a method eat with a parameter food with a default value = "Grass"

• Inside the method print (self.name, " eats", food)

• Define a class Mammal, inherit from Animal

• Inside the class, override eat method to print(self.name, " does not eat. It only drinks")

• Define class WingedAnimal, inherit from Animal• Override eat method to print(self.name," eats anything and everything")

• Define a class called Bat, inherit from WingedAnimal, Mammal

• Define method smell, which prints "This Animal Stinks"

• Define a class called FruitBat, inherit from Mammal, WingedAnimal (Notice the Order)

• rabbit1 = Animal("Rabbit")

• print("Rabbit1 is an instance of Animal")

• rabbit1.eat() # Animal's eat method without food parameter

• rabbit1.eat("Peanuts") # Animal's eat method with food parameter

• print("Cow1 is an instance of Mammal")

• cow1 = Mammal("Cow")

• cow1.eat() # Mammal's eat method

• print("Vulture1 is an instance of WingedAnimal")

• vulture1 = WingedAnimal("Vulture")

• vulture1.eat() # WingedAnimal's eat method

• print("Bat1 is an instance of Bat")

• bat1 = Bat("Bat")

• bat1.eat() # WingedAnimal's eat method

• print("fbat is an instance of FruitBat")

• fbat = FruitBat("Fruitbat")

• fbat.eat() # Mammal's eat method.

**ANS:-**

Here's the implementation of the method overriding scenario you provided:

```python

class Animal:

def \_\_init\_\_(self, name="This Animal"):

self.name = name

def eat(self, food="Grass"):

print(self.name, "eats", food)

class Mammal(Animal):

def eat(self, food="Grass"):

print(self.name, "does not eat. It only drinks")

class WingedAnimal(Animal):

def eat(self, food="Grass"):

print(self.name, "eats anything and everything")

class Bat(WingedAnimal, Mammal):

def smell(self):

print("This Animal Stinks")

class FruitBat(Mammal, WingedAnimal):

pass

rabbit1 = Animal("Rabbit")

print("rabbit1 is an instance of Animal")

rabbit1.eat() # Animal's eat method without food parameter

rabbit1.eat("Peanuts") # Animal's eat method with food parameter

print("\ncow1 is an instance of Mammal")

cow1 = Mammal("Cow")

cow1.eat() # Mammal's eat method

print("\nvulture1 is an instance of WingedAnimal")

vulture1 = WingedAnimal("Vulture")

vulture1.eat() # WingedAnimal's eat method

print("\nbat1 is an instance of Bat")

bat1 = Bat("Bat")

bat1.eat() # WingedAnimal's eat method

bat1.smell() # Method specific to Bat

print("\nfbat is an instance of FruitBat")

fbat = FruitBat("Fruitbat")

fbat.eat() # Mammal's eat method

```

Output:

```

rabbit1 is an instance of Animal

This Animal eats Grass

This Animal eats Peanuts

cow1 is an instance of Mammal

Cow does not eat. It only drinks

vulture1 is an instance of WingedAnimal

Vulture eats anything and everything

bat1 is an instance of Bat

Bat eats anything and everything

This Animal Stinks

fbat is an instance of FruitBat

Fruitbat does not eat. It only drinks

```

In this implementation, the classes `Animal`, `Mammal`, `WingedAnimal`, `Bat`, and `FruitBat` are defined. Each class inherits from the appropriate parent class, and the `eat()` method is overridden in `Mammal`, `WingedAnimal`, and `Bat` to provide different behavior. The `smell()` method is defined specifically for the `Bat` class.

Instances of these classes are created, and the `eat()` method is called for each instance to demonstrate the method overriding behavior.

36. a) Assume the given instructions below to write the program

i.Take the input values a, b from the user.

ii. Define the function add().

iii. It takes two arguments a and b.

iv. Add a and b and return the result.

v. Call this function by passing the argument values a and b and print the result.

**ANS:-**

b) Assume the given instructions and write the program to understand Key arguments

• Define a function simplecalc()• Takes two parameters a and b simplecalc(a, b)

• Inside the function do all the operations +, -, \* on a and b.

• Print the result of all the operations

• set values a = 3 and b = 4.

• Call the method by passing two values as keyword argument one in the regular order and one in the reverse order.

**ANS:-**

37. Which string method is used to implement the following?

(i) To check whether the given character is a letter or a number.

(ii) To change lower case to upper case letter.

**ANS:-**

1. Write a user-defined function named Count() that will read the contents of text file named “Report.txt” and count the number of lines which starts with either “I‟ or “M‟ and displays the count.
2. **ANS:-**
3. Write a user-defined function named Count() that will read the contents of text file named “Report.txt” and count the number of lines which starts with either “I‟ or “M‟ and displays the count. **ANS:-**

41, Implement in Python to create your own universal function in NumPy.

**ANS:-**

42, Devise string slicing in python with syntax. Take an example of string and demonstrate different types of slicing with code.

**ANS:-**

1. Discuss the use of the following methods with a sample string: lstrip(), rstrip(), strip()

**ANS:-**

1. Write a Python program to create a lambda function that adds 15 to a given number passed in as an argument, also create a lambda function that multiplies argument x with argument y and prints the result.

**ANS:-**

1. Using regular expression, write a python program to check that a string contains only a certain set of characters (in this case a-z, A-Z and 0-9).

**ANS:-**

1. List the three types of conditional statements and explain them.

**ANS:-**

1. Implement in Python to accessing Data Along Multiple Dimensions Arrays in Python Numpy.

**ANS:-**

Certainly! Here's a more comprehensive example that demonstrates accessing data along multiple dimensions in NumPy arrays:

```python

import numpy as np

# Create a 3-dimensional array

arr = np.array([

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

[[7, 8, 9], [10, 11, 12]]

])

# Accessing individual elements

print(arr[0, 0, 0]) # Output: 1

print(arr[1, 1, 2]) # Output: 12

# Accessing a 2-dimensional slice

print(arr[1, :, :])

# Output:

# [[ 7 8 9]

# [10 11 12]]

# Accessing a 1-dimensional slice

print(arr[:, 1, 0])

# Output: [ 4 10]

# Modifying elements

arr[0, 1, 2] = 100

print(arr)

# Output:

# [[[ 1 2 3]

# [ 4 5 100]]

#

# [[ 7 8 9]

# [ 10 11 12]]]

```

In this example, we create a 3-dimensional array `arr` using the NumPy `array` function. This array represents a 2x2 matrix, where each element is a 1x3 array.

To access individual elements, we use indexing with three indices. For example, `arr[0, 0, 0]` returns the element at the first row, first column, and first depth level, which is `1`.

You can access slices of the array along multiple dimensions. For instance, `arr[1, :, :]` retrieves a 2-dimensional slice from the second depth level, including all rows and columns.

You can also access specific elements from a dimension by using a colon `:` for the other dimensions. In the example, `arr[:, 1, 0]` retrieves a 1-dimensional slice that includes the first element from the second dimension for both depth levels.

You can modify individual elements by assigning new values using indexing. In the example, we change the element at the first row, second column, and third depth level to `100`.

By utilizing indexing and slicing along multiple dimensions, you can access and modify data in NumPy arrays with various shapes and dimensions.

1. Implement in Python to accessing Data Along Multiple Dimensions Arrays in Python Numpy.

**ANS:-**

To access data along multiple dimensions in arrays using the NumPy library in Python, you can use indexing and slicing techniques. NumPy provides powerful tools for array manipulation, making it easy to access and manipulate data in multidimensional arrays. Here's an example that demonstrates accessing data along multiple dimensions:

```python

import numpy as np

# Create a 2-dimensional array

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

[4, 5, 6],

[7, 8, 9]])

# Accessing individual elements

print(arr[0, 0]) # Output: 1

print(arr[1, 2]) # Output: 6

# Accessing a row

print(arr[1]) # Output: [4 5 6]

# Accessing a column

print(arr[:, 1]) # Output: [2 5 8]

# Accessing a subarray using slicing

print(arr[0:2, 1:3])

# Output:

# [[2 3]

# [5 6]]

```

In the above example, we create a 2-dimensional array `arr` using the NumPy `array` function. This array represents a matrix with three rows and three columns.

To access individual elements, we use indexing with square brackets. For example, `arr[0, 0]` returns the element at the first row and first column, which is `1`.

You can also access entire rows or columns by specifying the corresponding index. For instance, `arr[1]` retrieves the second row `[4 5 6]`, and `arr[:, 1]` retrieves the second column `[2 5 8]`.

Slicing is a powerful technique for accessing subarrays. In the example, `arr[0:2, 1:3]` returns a subarray consisting of the first and second rows and the second and third columns:

```

[[2 3]

[5 6]]

```

Here, the range `0:2` specifies the rows, and `1:3` specifies the columns.

By combining these indexing and slicing techniques, you can access and manipulate data along multiple dimensions in NumPy arrays.

48. (a) Write the output of the below code?pset\_time = 15

sleep\_time = 8

print(sleep\_time > pset\_time)

derive = True

drink = False

both = drink and derive

print(both)

**ANS:-**

(b) Find the value of brunch after you execute all the operations in the code below?

L1 = ["bacon", "eggs"]

L2 = ["toast", "jam"]

brunch = L1

L1.append("juice")

brunch.extend(L2)

**ANS:-**

49. Evaluate the following python code and identify the output of each statement

a=6

b=7

c=42

print 1, a==6

print 2, a==7

print 3, a==6 and b==7

print 4, a==7 and b==7

print 5, not a==7 and b==7

print 6, a==7 or b==7

print 7, a==7 or b==6

print 8, not(a==7 and b==6)

print 9, not a==7 and b==6

**ANS:-**

Let's evaluate the provided Python code and identify the output of each statement:

```python

a = 6

b = 7

c = 42

print(1, a == 6)

print(2, a == 7)

print(3, a == 6 and b == 7)

print(4, a == 7 and b == 7)

print(5, not a == 7 and b == 7)

print(6, a == 7 or b == 7)

print(7, a == 7 or b == 6)

print(8, not (a == 7 and b == 6))

print(9, not a == 7 and b == 6)

```

Output:

```

1 True

2 False

3 True

4 False

5 True

6 True

7 False

8 True

9 False

```

Explanation:

- Statement 1: `a == 6` is True because the value of `a` is 6.

- Statement 2: `a == 7` is False because the value of `a` is 6, not 7.

- Statement 3: `a == 6 and b == 7` is True because both conditions are True.

- Statement 4: `a == 7 and b == 7` is False because the first condition (`a == 7`) is False.

- Statement 5: `not a == 7 and b == 7` is True because the first condition (`not a == 7`) is True.

- Statement 6: `a == 7 or b == 7` is True because at least one of the conditions is True (`b == 7` is True).

- Statement 7: `a == 7 or b == 6` is False because both conditions are False.

- Statement 8: `not (a == 7 and b == 6)` is True because both conditions (`a == 7` and `b == 6`) are False.

- Statement 9: `not a == 7 and b == 6` is False because the first condition (`not a == 7`) is False.

The output of each statement is displayed alongside the respective statement number.

50. Discuss the "get" method in python dictionaries, including its syntax, parameters, and output. How does it differ from using square brackets to access a key in a dictionary and when would you use one over the other?

**ANS:-**

The `get()` method in Python dictionaries provides a way to retrieve the value associated with a given key. It offers some additional features compared to using square brackets for key access. Let's discuss the syntax, parameters, output, and the differences between `get()` and square bracket key access:

Syntax of `get()` method:

```python

dictionary.get(key, default)

```

Parameters:

- `key`: The key whose value you want to retrieve from the dictionary.

- `default` (optional): The default value to return if the key is not found in the dictionary. If not provided, it defaults to `None`.

Output:

- If the `key` is found in the dictionary, the method returns the corresponding value.

- If the `key` is not found in the dictionary, the method returns the `default` value.

Differences between `get()` and square bracket key access:

1. Error handling: When using square brackets to access a key in a dictionary, if the key doesn't exist, a `KeyError` is raised. On the other hand, the `get()` method allows you to provide a default value, which is returned instead of raising an error when the key is not found.

2. Default value: With the `get()` method, you can specify a default value as the second parameter. If the key is not found, this default value is returned. This can be useful when you want to handle missing keys in a dictionary gracefully, without causing errors or exceptions.

Usage scenarios:

- If you are confident that a key will always be present in the dictionary and you want to directly access its value, using square brackets (`dictionary[key]`) is straightforward and concise.

- If you are unsure about the presence of a key or you want to handle missing keys gracefully, you can use the `get()` method. It allows you to provide a default value that will be returned when the key is not found, preventing errors.

Example:

```python

person = {'name': 'John', 'age': 25}

# Using square brackets for key access

print(person['name']) # Output: John

# print(person['country']) # Raises KeyError: 'country'

# Using get() method

print(person.get('name')) # Output: John

print(person.get('country')) # Output: None

# Providing a default value with get()

print(person.get('country', 'Unknown')) # Output: Unknown

```

In the example, we have a dictionary named `person` with keys `'name'` and `'age'`. When accessing the key `'name'` using square brackets (`person['name']`), we get the corresponding value `'John'`. However, if we try to access the key `'country'` that doesn't exist, a `KeyError` will be raised.

By using the `get()` method, we can safely retrieve the value for the key `'name'` and `'country'`. When the key `'country'` is not found, the `get()` method returns `None` as the default value. Additionally, by providing a default value with `person.get('country', 'Unknown')`, we get `'Unknown'` as the result when the key is not found.

Using `get()` can be advantageous when you want to avoid potential errors caused by missing keys or when you need to handle default values for missing keys in a dictionary.