Python Interview Questions

Q.1. What is python ?

Ans: Python is a popular high-level programming language known for its simplicity and readability. Created by Guido van Rossum and first released in 1991, Python has gained widespread popularity due to its versatility and ease of use. It is widely used in various domains such as web development, scientific computing, data analysis, artificial intelligence, machine learning, automation, and more.

Key features of Python include:

Readability: Python's syntax emphasizes readability and allows programmers to express concepts in fewer lines of code compared to languages like C++ or Java.
Dynamically Typed: Python is dynamically typed, meaning you don't need to explicitly declare variable types. The interpreter determines the type at runtime.
Interpreted: Python is an interpreted language, which means you can write and run code directly without the need for compiling.
Cross-platform: Python programs can run on various operating systems without modifications, as long as the required interpreter is available.
Extensive Standard Library: Python comes with a rich standard library that provides modules and functions for a wide range of tasks, from file handling to networking and more.
Large Community and Ecosystem: Python has a vibrant community of developers, which has led to the creation of numerous third-party libraries and frameworks that extend its capabilities.
Object-Oriented: Python supports object-oriented programming (OOP) concepts, allowing developers to create and use classes and objects.
Indentation: Unlike many programming languages that use braces or other symbols to define code blocks, Python uses indentation to indicate the structure of the code. This enforces consistent formatting and improves code readability.
Open Source: Python is open source, meaning its source code is available to the public, and developers can contribute to its development.

Q.2. How to optimize Python code for better performance?

Ans: Optimizing Python code for better performance often involves identifying and addressing bottlenecks in your code.
Here are some strategies and techniques to improve the performance of your Python programs:

1. **Profile Your Code:**
   Before optimizing, it's essential to know where the performance bottlenecks are. Use profiling tools like `cProfile` or third-party profilers to identify which parts of your code consume the most time.

2. **Use Built-in Functions and Libraries:**
   Python's built-in functions and libraries are highly optimized and typically faster than writing custom code. Utilize them wherever possible.
3. **Avoid Global Variables:**
   Global variables are slower to access than local variables. Minimize the use of global variables within functions.
4. **Use List Comprehensions:**
   List comprehensions are often faster and more concise than traditional loops for creating lists.
5. **Avoid Unnecessary Loop:**
   Reduce the number of loops in your code, especially nested loops, as they can significantly impact performance. Look for ways to simplify and optimize loops.
6. **Optimize Data Structures:**
   Choose the appropriate data structures for your specific use case. Lists, sets, dictionaries, and NumPy arrays have different performance characteristics.
7. **Use Generators and Iterators:**
   Use generators and iterators when dealing with large datasets. They allow you to process data lazily, reducing memory consumption.
8. **Avoid Recursion for Deep Calls:**
   Python's recursion depth is limited. Avoid deep recursive calls, and use iteration or other techniques for deep tasks.
9. **Profile Memory Usage:**
   Use memory profiling tools like `memory_profiler` to identify memory-intensive parts of your code. Reducing memory usage can lead to performance improvements.
10. **Use Cython or Numba:**
    For CPU-bound code, consider using Cython or Numba to convert Python code to optimized C code or machine code, respectively. These tools can provide significant speedups.
11. **Concurrency and Parallelism:**
    For CPU-bound tasks, consider using the `multiprocessing` module to utilize multiple CPU cores. For I/O-bound tasks, use asynchronous programming with `asyncio.`
12. **Cache Results:**
    If your code performs costly calculations that produce the same results for the same inputs, consider caching the results to avoid redundant computations.
13. **Avoid Unnecessary File Operations:**
    Minimize reading and writing to files, databases, or network resources inside loops. Instead, read or write data in bulk when possible.
14. **Optimize Algorithm Complexity:**
    Analyze your algorithms for time complexity and choose algorithms that are more efficient for your specific problem.
15. **Profiling on Target Environment:**

Profile and test your code in the target environment or deployment configuration to ensure that optimizations are effective.

16. **Keep Your Code Clean:**
    Maintain clean and readable code. Well-structured code is easier to optimize, debug, and maintain.
17. **Use a JIT Compiler:**
    Consider using Just-In-Time (JIT) compilers like PyPy, which can provide substantial performance improvements for certain workloads.

Q.3. What are mutable and immutable data types available in python?
Ans: In Python, data types can be classified as either mutable or immutable. Here's a breakdown of mutable and immutable data types:

Immutable Data Types:

Integers (`int`): Once an integer value is assigned, it cannot be changed. Any mathematical operation on an integer creates a new integer object.
Floating-Point Numbers (`float`): Like integers, floating-point numbers are also immutable. Operations involving floating-point numbers create new objects.
Strings (`str`): Strings are immutable, which means you cannot modify individual characters of a string after it's created. However, you can create new strings through concatenation and other string manipulation methods.
Tuples (`tuple`): Tuples are immutable sequences. Once a tuple is created, you cannot modify its elements. However, you can create new tuples by concatenating or slicing existing tuples.
Frozen Sets (`frozenset`): Similar to sets, but immutable. Once a frozen set is created, you cannot add or remove elements from it.
Bytes (`bytes`): Bytes are immutable sequences of bytes. Once created, you cannot modify the individual bytes.

Mutable Data Types:

Lists (`list`): Lists are mutable, meaning you can change their contents after they're created. You can add, remove, or modify elements within a list.
Dictionaries (`dict`): Dictionaries are mutable collections of key-value pairs. You can add, update, or remove key-value pairs from a dictionary.
Sets (`set`): Sets are mutable collections of unique elements. You can add or remove elements from a set after it's created.
Byte Arrays (`bytearray`): Byte arrays are mutable sequences of bytes. Unlike the `bytes` type, you can modify the individual bytes within a byte array.

User-Defined Classes (objects): If you create your own classes, whether they are mutable or immutable depends on how you implement them. You can design classes with mutable or immutable properties based on your needs.

Q.4. What is PEP8 and how do you implement it when you are writing coding in python?
Ans:
PEP 8 is the official style guide for writing Python code. It stands for "Python Enhancement Proposal 8," and it outlines conventions and guidelines for writing clean, readable, and consistent Python code. Following PEP 8 helps improve code maintainability and collaboration by ensuring a uniform style across different projects**.**

Here are some key principles and recommendations from PEP 8:

Indentation: Use 4 spaces per indentation level. Avoid tabs or mixing spaces and tabs.
Maximum Line Length: Limit lines to 79 characters for code and 72 characters for docstrings and comments. If a line needs to be longer, it's recommended to break it into multiple lines**.**
Imports: Import individual modules instead of using wildcard imports **(**`from module import *`**).** Place imports at the top of the file and group them in the following order: standard library modules, third-party library modules, and local project modules.
Whitespace in and Statements: Avoid extraneous whitespace, but use whitespace for improved readability. For example, use a single space around binary operators and after commas in function arguments.
Comments: Use comments to explain non-obvious parts of your code. Keep comments concise and to the point. Use docstrings to document functions, classes, and modules.
Naming Conventions: Follow consistent naming conventions. Use `lowercase_with_underscores` **for variable and function names,** `CAPITALIZED_WITH_UNDERSCORES` **for constants, and** `CamelCase` **for class names.**
Whitespace in Function and Class Definitions: Use two blank lines to separate top-level functions and classes. Use one blank line to separate method definitions within a class.
Parentheses: Use parentheses around conditional expressions for improved clarity, even if they are not strictly necessary.
Avoid Extraneous Spaces: Don't put spaces immediately inside parentheses, brackets, or braces.
Imports Ordering: Group imports from the same module and put them in alphabetical order.

To implement PEP 8 when writing Python code:

Use Linters: Use tools like `pylint` or `flake8` to automatically check your code against PEP 8 guidelines. These tools can identify style violations and provide suggestions for corrections.

Editor Integration: Many code editors and integrated development environments (IDEs) offer plugins or built-in features that can automatically highlight PEP 8 violations as you type.
Be Consistent: Consistency is key. Choose a consistent style and stick to it throughout your codebase.
Review and Refactor: Regularly review your code for PEP 8 compliance and refactor if needed. This is particularly important when collaborating with others.

Q.5. What is append(), extend() and insert() in list python?
Ans: In Python, lists are a common data structure used to store collections of items. The methods `append()`, `extend()`, and `insert()` are used to modify lists, but they have different purposes:

1. **append()** Method:
   The `append()` method is used to add a single element to the end of a list.
   Syntax: `list.append(item)`
   Example:
   ```
   my_list = [1, 2, 3]
   my_list.append(4)
   print(my_list)  # Output: [1, 2, 3, 4]
   ```

2. **extend()** Method:
   The `extend()` method is used to append multiple elements from an iterable (usually another list) to the end of the current list.
   Syntax: `list.extend(iterable)`
   Example:
   ```
   list1 = [1, 2, 3]
   list2 = [4, 5, 6]
   list1.extend(list2)
   print(list1)  # Output: [1, 2, 3, 4, 5, 6]
   ```

3. **insert()** Method:
   The `insert()` method is used to insert an element at a specific index within the list.
   Syntax: `list.insert(index, item)`
   Example:
   ```
   my_list = [1, 2, 3]
   my_list.insert(1, 4)
   print(my_list) # Output: [1, 4, 2, 3]
   ```

Q.6. What are pop(), remove(),clear(), del in list python ?
Ans: In Python, the pop(),remove() and clear() methods are used to modify lists by removing elements.

1. **pop()** Method:

The `pop()` method is used to remove an element from a specific index in the list and return the removed element.
If no index is provided, it removes and returns the last element by default.
Syntax: `list.pop(index)`
Example:

```
my_list = [1, 2, 3, 4, 5]
removed_element = my_list.pop(2)  # Removes element at index 2 (value 3)
print(my_list)  # Output: [1, 2, 4, 5]
print(removed_element)  # Output: 3
```

2. **remove()** Method**:**
   The `remove()` method is used to remove the first occurrence of a specific value from the list.
   Syntax: `list.remove(value)`
   Example:
   ```
    my_list = [1, 2, 3, 4, 5]
   my_list.remove(3)  # Removes the first occurrence of the value 3
   print(my_list)  # Output: [1, 2, 4, 5]
   ```

3. **clear()** Method:
   The `clear()` method is used to remove all elements from the list, effectively emptying the list.
   Syntax: `list.clear()`
   Example:
   ```
   my_list = [1, 2, 3, 4, 5]
   my_list.clear()  # Removes all elements from the list
   print(my_list)  # Output: []
   ```

4. **del** keyword:
   The del keyword is used to remove elements from list based index.
   E.g l1=[10,20,30,40]
   ```
       del l1[1]  # [10,30,40]
       del l1[1:3] # [10,40]
   ```

Q.7. What is module,package and library in python?
Ans:
Module:

- A module is a single file containing Python code. It can include functions, classes, variables, and other definitions. Modules are used to organize related code into separate files for better organization and maintainability.

- Modules can be imported and used in other Python scripts or modules using the `import` statement.
- **Example:** If you have a file named `my_module.py`, it can be imported in another script using `import my_module`
- **Example:** import os,random ,datetime,csv.

Package:

- A package is a collection of related modules grouped together in a directory structure. The directory that contains the modules is considered the package, and it must also include a special file called `__init__.py` (which can be empty) to indicate that the directory should be treated as a package.
- Packages help organize and manage larger projects by breaking code into smaller, logical units.
- Example: If you have a directory named `my_package` containing multiple `.py` files and an `__init__.py` file, it's considered a package. You can then import modules from this package using `import my_package.my_module`.
- Example: Numpy, Pandas

Library:

- A library (sometimes called a "library package") is a collection of related packages and modules that provide a set of functionalities to solve specific problems. A library can include a variety of modules and packages designed to be used together to accomplish specific tasks.
- Libraries can be third-party or built-in. Built-in libraries are included with Python, while third-party libraries are created by other developers and can be installed using tools like **`pip`**.
- **Example:** request,math,matplotlib,seaborn etc.

Q.8. What is the Floor and Ceil method in Python?
Ans:
floor() Function:
The `floor()` function is used to round a number down to the nearest integer that is less than or equal to the original number. In other words, it returns the largest integer that is less than or equal to the given number.
Example: import math

x = 4.8
result = math.floor(x)
print(result) # Output: 4

ceil() Function:
The ceil() function, short for "ceiling," is used to round a number up to the nearest integer that is greater than or equal to the original number. In other words, it returns the smallest integer that is greater than or equal to the given number.

```
Example:
import math
x = 4.2
result = math.ceil(x)
print(result)  # Output: 5
```

Q.9. What are global, protected and private attributes in Python?
Ans:

- **Global attribute** are public variables that are defined in the global scope. To use the variable in the global scope inside a function, we use the global keyword.
- **Protected attribute** are attributes defined with an underscore prefixed to their identifier eg. _sara. They can still be accessed and modified from outside the class they are defined in but a responsible developer should refrain from doing so.
- **Private attribute** are attributes with double underscore prefixed to their identifier eg. __ansh. They cannot be accessed or modified from the outside directly and will result in an AttributeError if such an attempt is made.

Q.10. What is break, continue and pass keyword in python?

Ans: In Python, `break`, `continue`, and `pass` are control flow keywords used within loops and conditional statements to control the flow of execution in your code.

1. **`break`:** The `break` keyword is used inside loops (like `for` and `while` loops) to prematurely exit the loop. When a **break** is encountered, the loop terminates, and the program continues to execute the code after the loop.

   Example:

   ```
   for i in range(5):
      if i == 3:
         break
      print(i)   # output:  0 1 2
   ```

2. **continue:** The `continue` keyword is used inside loops to skip the rest of the current iteration and move on to the next iteration. It's often used when you want to skip some specific iteration based on a certain condition.

   Example:

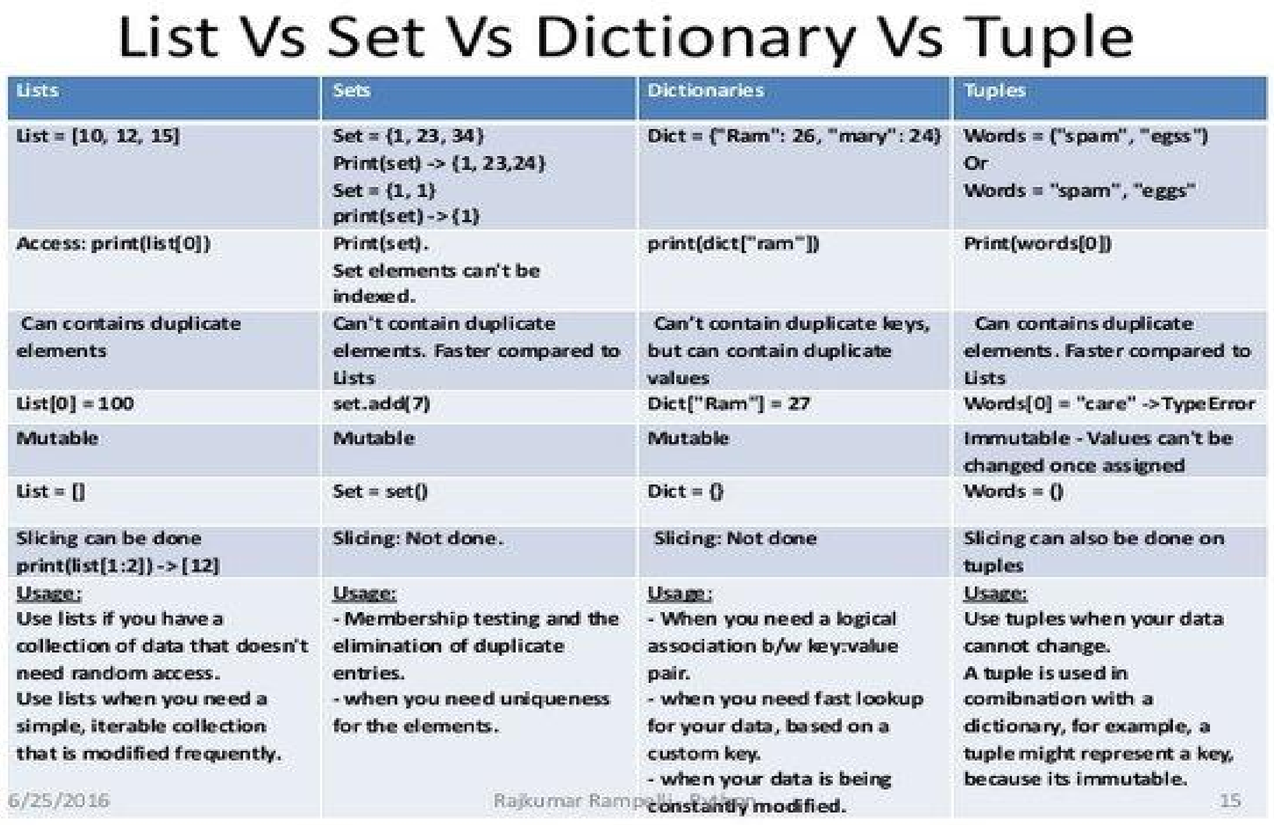
   ```
   for i in range(5):
      if i == 2:
   ```

```
            continue
        print(i)    #output: 0 1 3 4
```

3. **`pass`**: The `pass` keyword is a placeholder statement that does nothing. It's used when you need to have a syntactically correct block of code but you don't want to put any actual statements there. This can be useful when you're designing code and you want to leave certain parts unfinished for now.
   Example:
   ```
   for i in range(5):
       if i == 2:
           pass
       else:
           print(i)  #output: 0 1 3 4
   ```

Q.11. List vs Tuple vs Set vs Dictionary in Python ?

Ans:



List Vs Set Vs Dictionary Vs Tuple

Lists	Sets	Dictionaries	Tuples

List = [10, 12, 15]	Set = {1, 23, 34}
	Print(set) -> {1, 23,24}
	Set = {1, 1}
	print(set) -> {1}
		Dict = {"Ram": 26, "mary": 24}	Words = ("spam", "egss")
			Or
			Words = "spam", "eggs"

Access: print(list[0])	Print(set). Set elements can't be indexed.	print(dict["ram"])	Print(words[0])
Can contains duplicate elements	Can't contain duplicate elements. Faster compared to Lists	Can't contain duplicate keys, but can contain duplicate values	Can contains duplicate elements. Faster compared to Lists
List[0] = 100	set.add(7)	Dict["Ram"] = 27	Words[0] = "care" ->TypeError
Mutable	Mutable	Mutable	Immutable - Values can't be changed once assigned
List = []	Set = set()	Dict = {}	Words = ()
Slicing can be done print(list[1:2]) -> [12]	Slicing: Not done.	Slicing: Not done	Slicing can also be done on tuples
Usage: Use lists if you have a collection of data that doesn't need random access. Use lists when you need a simple, iterable collection that is modified frequently.	Usage: - Membership testing and the elimination of duplicate entries. - when you need uniqueness for the elements.	Usage: - When you need a logical association b/w key:value pair. - when you need fast lookup for your data, based on a custom key. - when your data is being constantly modified.	Usage: Use tuples when your data cannot change. A tuple is used in comibnation with a dictionary, for example, a tuple might represent a key, because its immutable.

6/25/2016 Rajkumar Rampelli - Python 15

Q.12. What is the lambda function in python?

Ans: A lambda function in Python is a small, anonymous (unnamed) function that can have any number of arguments, but can only have one expression.

Lambda functions are often used for short, simple operations that can be defined in a single line of code. They are also known as "lambda expressions."

lambda arguments: expression

`lambda`: This keyword is used to define a lambda function.

`arguments`: These are the input parameters or arguments that the lambda function takes. You can have multiple arguments separated by commas.

`expression`: This is the single expression that the lambda function will evaluate and return.

Lambda functions are commonly used with functions like **`map()`, `filter()`, `sorted()`** when you need to pass a simple function as an argument.

Example 1 - Using **`lambda`** with **`map()`:**

```
numbers = [1, 2, 3, 4, 5]
squared = map(lambda x: x**2, numbers)
print(list(squared))  # Output: [1, 4, 9, 16, 25]
```

Example 2 - Using **`lambda`** with **`sorted()`** to sort a list of tuples by the second element:

```
points = [(2, 3), (1, 5), (4, 1)]
sorted_points = sorted(points, key=lambda point: point[1])
print(sorted_points)  # Output: [(4, 1), (2, 3), (1, 5)]
```

Example 3: Using reduce() The **reduce(fun,seq)** function is used to **apply a particular function passed in its argument to all of the list elements**

```
 from functools import reduce
 n=[1,2,3,4,5]
res= reduce(lambda a,b:a+b,n)
print(res) #  17
```

Q.13. .py vs .pyc

Ans:

`.py` Files: A `.py` file is a source code file written in the Python programming language. It contains the human-readable Python code that defines functions, classes, variables, and other program logic. These files are plain text files that developers create and edit using text editors or integrated development environments (IDEs).

`.pyc` Files: A `.pyc` file is a compiled Python file. It stands for "Python Compiled file" or "Compiled Python Bytecode." When you run a Python program, the Python interpreter first translates the human-readable source code from the `.py` files into a lower-level representation called bytecode. Bytecode is a set of instructions that the Python interpreter can execute directly.

The Python interpreter compiles the source code into bytecode to improve execution speed.

Q.14. What is List comprehension?

Ans: List comprehension is a concise and readable way to create lists in Python.

It provides a compact syntax for generating lists by applying an expression to each item in an iterable (such as a list, tuple, or range) and optionally applying conditions to filter the items.

The general syntax of a list comprehension is:

new_list = [expression for item in iterable if condition]

Example:

squares = [x**2 for x in range(10)]

print(squares)   #output: `[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]`

Q.15. What is Dictionary Comprehension?

Ans: Dictionary comprehension is a concise and efficient way to create dictionaries in Python. It follows a similar syntax to list comprehension but allows you to generate dictionaries by specifying both keys and values using an expression applied to each item in an iterable.

The general syntax of a dictionary comprehension is:

new_dict = {key_expression: value_expression for item in iterable if condition}

Example:

squares_dict = {x: x**2 for x in range(1, 6)}

print(squares_dict)   #output: {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

However, just like with list comprehensions, it's important to use dictionary comprehensions in a way that maintains code readability and avoids excessive complexity.

Q.16. How to find the size of the variable in python?

Ans: You can find the size of a variable or object in Python using the `sys` module's getsizeof() function. The getsizeof() function returns the memory size of the object in bytes. Here's how you can use it:

```
import sys
variable = ...  # Replace this with the variable you want to measure
size = sys.getsizeof(variable)
print(f"Size of the variable: {size} bytes")
```

Q.17. What is an Iterator in Python?

Ans: In Python, an **iterator** is an object that provides a way to access elements from a collection or sequence one by one, without the need to know the underlying structure of that collection. It allows you to iterate (loop) through the elements of a container like a list, tuple, dictionary, etc., in a controlled and efficient manner.

The key concept behind iterators is that they implement two methods: `__iter__()` and `__next__()`.

The `__iter__()` method returns the iterator object itself and is responsible for setting up the initial state of the iteration.

The `__next__()` method returns the next element from the container and advances the internal state of the iterator.

When there are no more items to be returned, the `__next__()` method raises the `StopIteration` exception to signal the end of iteration.

Example 1: Using iter() and next() Functions**:**

```
my_list = [100, 200, 300]
iter_obj = iter(my_list)
print(next(iter_obj))  # Output: 100
print(next(iter_obj))  # Output: 200
print(next(iter_obj))  # Output: 300
```

Q.18. What is a generator in python?

Ans: In Python, a **generator** is a type of iterable that allows you to iterate over a sequence of values while generating each value on-the-fly, rather than storing them in memory all at once. Generators are implemented using a special type of function called a generator function or using generator expressions.

The key difference between generators and other iterables (like lists or tuples) is that generators do not store all the values in memory at once. Instead, they generate values one at a time as you iterate over them, which can significantly reduce memory consumption and improve efficiency, especially when dealing with large data sets or infinite sequences.

There are two main ways to create generators in Python:

1.Generator Functions: These are defined using the `yield` keyword instead of the `return` keyword. When a generator function is called, it doesn't execute the entire function body immediately; instead, it returns a generator object that can be used to iterate over the values produced by the `yield` statements.

Here's an example of a generator function that generates Fibonacci numbers:

```
def fibonacci_generator():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b
fib_gen = fibonacci_generator()
for _ in range(10):
    print(next(fib_gen))
```

2.Generator Expressions: These are similar to list comprehensions, but they create generator objects instead of lists. They are more memory-efficient for large data sets.

```
even_squares_gen = (x ** 2 for x in range(10) if x % 2 == 0)
for square in even_squares_gen:
    print(square)
```

Q.19. Explain the purpose of access modifiers (private, protected, public) in Python?

Ans:

In Python, access modifiers, such as private, protected, and public, are used to control the visibility and accessibility of class members (attributes and methods).

1. **Public (No Modifier):**
   a. Attributes and methods declared without any access modifier are considered public.
   b. They can be accessed from anywhere, both inside and outside the class.
   c. Public members are part of the class's public interface and are intended to be used freely.

   Example:

```
class MyClass:
    def __init__(self):
            self.public_var = 42
    def public_method(self):
        return "This is a public method"
```

2. **Protected (_ Prefix):**
   a. In Python, there's no strict enforcement of protected members. Instead, it is a convention to prefix attribute and method names with a single underscore (e.g., _protected_var).
   b. Protected members are meant to be considered non-public, and their use outside the class is discouraged. However, they can still be accessed if needed.

   Example:

```
class MyClass:
    def __init__(self):
        self._protected_var = 42
    def _protected_method(self):
        return "This is a protected method"
```

3. **Private (Double Underscore Prefix):**
   a. Similar to protected members, private members are also not strictly enforced in Python. Conventionally, attribute and method names are prefixed with a double underscore (e.g., `__private_var`).
   b. Private members are considered highly non-public, and their use outside the class is strongly discouraged.

Example:

```
class MyClass:
  def __init__(self):
    self.__private_var = 42
  def __private_method(self):
    return "This is a private method"
```

Q.20. What is Pickling and Unpickling in Python?

Ans:

Pickling: Pickling is the process of converting a Python object into a serialized byte stream. This serialized representation can be stored in a file, transmitted over a network, or used for other purposes. The `pickle` module provides functions to perform pickling. You can use the `pickle.dump()` function to serialize an object and write it to a file, or you can use the `pickle.dumps()` function to get the serialized representation as a byte object.

Example:

```
import pickle
with open('data.pickle', 'rb') as file:
  loaded_data = pickle.dump(file)
```

Unpickling: Unpickling is the reverse process of pickling. It involves converting a serialized byte stream back into a Python object. You can use the **`pickle.load()`** function to deserialize an object from a file, or the **`pickle.loads()`** function to deserialize from a byte object.

Example:

```
import pickle
with open('data.pickle', 'rb') as file:
  loaded_data = pickle.load(file)
```

In summary, pickling and unpickling are important techniques for saving and restoring Python object states. They are commonly used for caching, saving program states, and transferring data between processes or systems.

Q.21. What is Decorator in Python?

Ans:

In Python, a decorator is a higher-order function that allows you to modify or enhance the behavior of other functions or methods without changing their source code. Decorators are often used to add functionality like logging, authentication, access control, or caching to functions or methods.

Decorators are applied to functions or methods using the "@" symbol followed by the decorator function's name.

Example:

```
def sub_number_decorator(func):
   def wrapper(a,b):
      return func(a,b)
   return wrapper
def add_number_decorator(func):
   def wrapper(a,b):
      return func(a,b)
   return wrapper
@sub_number_decorator  # 2nd calling
@add_number_decorator   #1st calling
def add(a,b):
   print("Subtraction:",(a-b))
   print("Addition:",(a+b))
add(10,20)
Output: Subtraction: -10
           Addition: 30
```

```
----------------------------------------- Decorator class  Example  --------------------------
class Power(object):
          def __init__(self, arg):
                    self._arg = arg
          def __call__(self, a, b):
                    retval = self._arg(a, b)
                    return retval ** 2
@Power
def multiply_together(a, b):
          return a * b
```

```
print(multiply_together(2, 2))
```

Q.22. *args vs **kwargs in python

Ans:

`*args`: The `*args` syntax allows a function to accept a variable number of positional arguments. It collects these arguments into a tuple within the function. This is useful when you don't know in advance how many arguments will be passed to the function.

Here's an example:

```
def my_function(*args):
    for arg in args:
        print(arg)
my_function(1, 2, 3)   #output: 1 2 3
```

`**kwargs`: The `**kwargs` syntax allows a function to accept a variable number of keyword arguments (named arguments). These arguments are collected into a dictionary within the function. This is useful when you want to pass named parameters to a function without knowing all possible parameter names beforehand.

```
def my_function(**kwargs):
    for key, value in kwargs.items():
        print(key, value)
my_function(a=1, b=2, c=3)
Output:
a 1
b 2
c 3
```

Q.23. What is GIL(Global Interpreter Lock):

Ans:

The Global Interpreter Lock (GIL) is a mutex (a type of locking mechanism) in the CPython interpreter, which is the most widely used implementation of the Python programming language. The GIL is a mechanism used to synchronize the execution of threads in a multithreaded Python program.

Here are some key points about the GIL:

CPython-Specific: The GIL is specific to CPython, which is the standard and most commonly used implementation of Python. Other implementations like Jython (for Java) and IronPython (for .NET) do not have a GIL because they execute Python code differently.

Simplifies Memory Management: Python's memory management is not thread-safe. The GIL ensures that only one thread accesses and modifies Python objects in memory at a time, reducing the risk of memory corruption and simplifying memory management.

Impact on Multithreading: Due to the GIL, multi-threaded Python programs often do not achieve true parallelism when running CPU-bound tasks. This is because even though you might have multiple threads, only one of them can execute Python bytecode at a time, limiting the utilization of multiple CPU cores.

IO-Bound Tasks: The GIL has less impact on programs that perform a lot of I/O-bound tasks (such as reading/writing files, network operations) because during I/O operations, the GIL can be temporarily released by the executing thread.

Multithreading for Concurrency: While the GIL restricts true parallelism for CPU-bound tasks, it can still be beneficial for programs that require concurrency, such as those that need to handle multiple connections or perform asynchronous I/O operations.

Use of Multiprocessing: To take full advantage of multiple CPU cores for CPU-bound tasks, Python programmers often use the **multiprocessing** module, which creates separate processes (each with its own GIL) to achieve true parallelism.

Q.24. What is MultiThreading in python?

Ans:

Thread is a lightweight, independent, and concurrent execution unit within a program. Threads are a way to achieve parallelism, allowing you to perform multiple tasks concurrently.

Multithreading in Python refers to the ability of a program to execute multiple threads concurrently within a single process.

Each thread represents a separate path of execution, and multiple threads within a program can run independently, sharing the same resources such as memory space, file handles, and other process-specific resources.

Multithreading is a way to achieve concurrent execution and can be particularly useful for tasks that are I/O-bound (input/output-bound) or that require parallelism without the need for full utilization of multiple CPU cores.

Python's standard library provides the **threading** module, which allows you to create and manage threads in your programs.

Here's a basic example of using the `threading` module to create and start two threads:

```
from threading import Thread
def print_numbers():

    for i in range(1, 6):
        print("Number:", i)
def print_letters():
    for letter in 'abcde':
        print("Letter:", letter)
# Create two thread objects
thread1 = Thread(target=print_numbers,args=())
thread2 = Thread(target=print_letters,args=())
# Start the threads
thread1.start()
thread2.start()
# Wait for both threads to finish
thread1.join()
thread2.join()
print("Both threads have finished.")
```

Q.25.What is Multiprocessing python?

Ans:

A **Process** is an independent and self-contained execution unit of a program. Each process has its own memory space, resources, and Python interpreter. Processes are separate from each other, meaning they do not share memory by default, and they run independently. This makes processes a way to achieve true parallelism in Python, as opposed to threads, which are limited by the Global Interpreter Lock (GIL) and may not achieve full parallelism in CPU-bound tasks.

Multiprocessing in Python is a module that allows you to create and manage multiple processes in Python. A process is a program in execution. It is a self-contained execution environment that contains its own memory space, code, and variables.

Multiprocessing can be used to speed up your Python code by running multiple tasks in parallel. This is useful for tasks that are CPU-bound, such as image processing or data analysis.
To use multiprocessing, you first need to create a Process object. This object represents a new process. You can then start the process by calling the start() method.
Once the process has started, you can communicate with it using the send() and receive() methods. You can also use the join() method to wait for the process to finish.
Each process runs independently and has its own memory space and has its own Process id(PID).

Example:

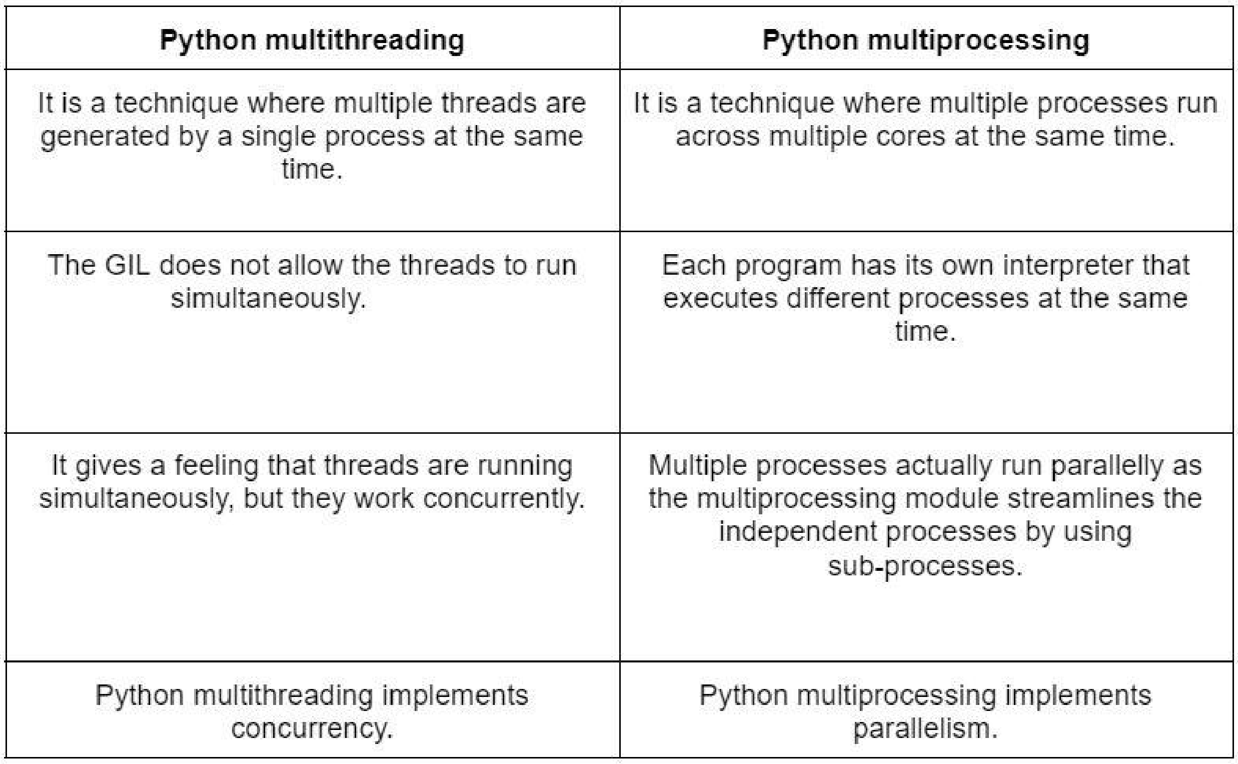
```
from multiprocessing import Process
def print_cube(num):
   print("Cube: {}".format(num * num * num))

def print_square(num):
   print("Square: {}".format(num * num))

if __name__ == "__main__":
   p1 = Process(target=print_square, args=(10, ))
   p2 = Process(target=print_cube, args=(10, ))
   p1.start()
   p2.start()
    p1.join()
   p2.join()
```

Q.26. MultiProcessing vs MultiThreading in Python?

Ans:



Python multithreading	Python multiprocessing
It is a technique where multiple threads are generated by a single process at the same time.	It is a technique where multiple processes run across multiple cores at the same time.
The GIL does not allow the threads to run simultaneously.	Each program has its own interpreter that executes different processes at the same time.
It gives a feeling that threads are running simultaneously, but they work concurrently.	Multiple processes actually run parallelly as the multiprocessing module streamlines the independent processes by using sub-processes.
Python multithreading implements concurrency.	Python multiprocessing implements parallelism.

Q.27. What is asynchronous programming in Python, and how does it differ from multi-threading?

Ans:

Asynchronous programming in Python is a programming paradigm that allows you to write non-blocking, concurrent code to handle multiple tasks efficiently. It enables a single-threaded program to perform tasks concurrently without waiting for one task to complete before starting another. Asynchronous programming is particularly useful for I/O-bound operations, such as network requests or file I/O, where the program can make progress while waiting for external resources.

Example:

```
import asyncio
async def foo():
   await asyncio.sleep(1)
   print("Foo done")
async def bar():
   await asyncio.sleep(2)
 print("Bar done")
```

```
async def main():
    await asyncio.gather(foo(), bar())
 asyncio.run(main())
```

In this example, **`foo()`** and **`bar()`** are asynchronous functions that pause execution with **`await asyncio.sleep()`** to simulate I/O-bound operations. The **`main()`** function gathers both coroutines, allowing them to run concurrently without blocking.

Now, let's compare asynchronous programming to multi-threading:

Asynchronous Programming:

- Uses a single thread.
- Suitable for I/O-bound tasks where tasks spend time waiting for external resources (e.g., network requests, file I/O).
- Achieves concurrency through the event loop, allowing multiple tasks to run concurrently but not in true parallelism.
- Typically has lower overhead than multithreading because it doesn't create many threads.

Multi-Threading:

- Uses multiple threads (multiple threads can run concurrently on multi-core processors).
- Suitable for CPU-bound tasks that can benefit from true parallelism.
- Requires synchronization mechanisms like locks and semaphores to manage shared resources and prevent race conditions.
- Can have higher overhead due to the creation and management of multiple threads
- Additionally, Python's Global Interpreter Lock (GIL) still affects multi-threading, limiting its ability to utilize multiple CPU cores for CPU-bound tasks.

Q.28. Explain the purpose of the async and await keywords in Python?

Ans:

`async` **Keyword:**

1. The `async` keyword is used to define an asynchronous function, also known as a coroutine. When you mark a function as `async`, you're indicating that this function may contain `await` expressions, and it won't block the event loop when called.
2. Async functions can be paused and resumed during their execution, allowing other tasks to run while the current task waits for asynchronous operations, like I/O, to complete.

`await` Keyword:

1. The `await` keyword is used within an async function to indicate that the function should pause at that point and allow the event loop to execute other tasks. It is typically used before an asynchronous operation that may take some time to complete, such as I/O or a network request.
2. When the awaited operation is finished, the async function resumes execution from where it left off.

`await` can be used with any asynchronous object or function, including asyncio's own asynchronous functions or third-party libraries that support async/await.

Together, **`async`** and **`await`** allow you to write asynchronous code that can efficiently handle multiple concurrent tasks. They enable you to perform I/O-bound operations, such as making network requests or reading/writing files, without blocking the entire program.

Example:

```
import asyncio
async def fetch_url(url):
    print(f"Fetching {url}")
    await asyncio.sleep(2)  # Simulate an asynchronous operation
    print(f"Completed {url}")

async def main():
    tasks = [fetch_url("https://example.com"), fetch_url("https://google.com")]
    await asyncio.gather(*tasks)
asyncio.run(main())
```

Q.29. How is memory managed in python?

Ans: Python uses a private heap to manage memory. The heap is a region of memory that is allocated to the Python process and is used to store all Python objects. The Python memory manager is responsible for allocating and freeing memory from the heap.

When a Python object is created, the memory manager allocates space for the object on the heap. The size of the space allocated is determined by the size of the object.

When a Python object is no longer needed, the memory manager frees the space that was allocated to the object.

The Python memory manager uses a garbage collector to free memory from the heap. The garbage collector periodically checks the heap for objects that are no longer referenced by any Python code. If the garbage collector finds an object that is no longer referenced, it frees the space that was allocated to the object.

The Python memory manager also uses a number of other techniques to manage memory efficiently. For example, the memory manager can share memory between objects that are similar. The memory manager can also preallocate memory for objects that are likely to be created.

Overall, the Python memory manager is a very efficient and effective way to manage memory. The memory manager is responsible for allocating and freeing memory from the heap, and it uses a number of techniques to manage memory efficiently.

Q.30. How to Handle Exceptions in Python?

Ans:

In Python, exceptions are used to handle errors or exceptional situations that may occur during the execution of a program. Properly handling exceptions is crucial for writing robust and reliable code. You can handle exceptions using the try, except, else, and finally blocks.

Here's how to handle exceptions in Python:

try: This block contains the code that might raise an exception. If an exception occurs, Python immediately jumps to the corresponding except block.

except ExceptionType as e: You specify the type of exception you want to catch (e.g., ZeroDivisionError, ValueError, FileNotFoundError, or a more general Exception). You can also use multiple except blocks to handle different types of exceptions. When an exception occurs, the code inside the corresponding except block is executed. You can access the exception object (the error message and details) as e.

else (optional): This block contains code that runs if no exception occurs in the try block. It is optional.

finally (optional): This block contains code that always runs, whether there was an exception or not. It's typically used for cleanup operations (e.g., closing files or network connections). It is also optional.

Example:

```
try:
    num1 = int(input("Enter a number: "))
    num2 = int(input("Enter another number: "))
    result = num1 / num2
except ZeroDivisionError as e:
    print(f"Error: {e}. You cannot divide by zero.")
except ValueError as e:
    print(f"Error: {e}. Please enter valid numbers.")
else:
    print(f"Result: {result}")
finally:
    print("Execution completed.")
```

Handling exceptions allows your program to gracefully recover from errors or terminate gracefully without crashing. It's important to choose the appropriate exception-handling strategy based on the specific errors your program may encounter.

Q.31. Explain Function of List, Set, Tuple And Dictionary?

Ans:

Functions Of List

❏ sort(): Sorts the list in ascending order.

❏ append(): Adds a single element to a list.

❏ extend(): Adds multiple elements to a list.

❏ index(): Returns the first appearance of the specified value.

❏ max(list): It returns an item from the list with max value.

❏ min(list): It returns an item from the list with min value.

❏ len(list): It gives the total length of the list.

❏ list(seq): Converts a tuple into a list.

❏ cmp(list1, list2): It compares elements of both lists list1 and list2.

❏ type(list): It returns the class type of an object.

Functions Of Tuple

❏ cmp(tuple1, tuple2) - Compares elements of both tuples.

❏ len(): total length of the tuple.

❏ max(): Returns item from the tuple with max value.

❏ min(): Returns item from the tuple with min value.

❏ tuple(seq): Converts a list into tuple.

❏ sum(): returns the arithmetic sum of all the items in the tuple.

❏ any(): If even one item in the tuple has a Boolean value of True, it returns True. Otherwise, it returns False.

❏ all(): returns True only if all items have a Boolean value of True. Otherwise, it returns False.

❏ sorted(): a sorted version of the tuple.

❏ index(): It takes one argument and returns the index of the first appearance of an item in a tuple

❏ count(): It takes one argument and returns the number of times an item appears in the tuple.

Functions Of Dictionary

❏ clear(): Removes all the elements from the dictionary

❏ copy(): Returns a copy of the dictionary

❏ fromkeys(): Returns a dictionary with the specified keys and value

❏ get(): Returns the value of the specified key

❏ items(): Returns a list containing a tuple for each key value pair

❏ keys(): Returns a list containing the dictionary's keys

❏ pop(): Removes the element with the specified key

❏ popitem(): Removes the last inserted key-value pair

❏ setdefault(): Returns the value of the specified key. If the key does not exist: insert the key, with the specified value

❏ update(): Updates the dictionary with the specified key-value pairs

❏ values(): Returns a list of all the values in the dictionary

❏ cmp(): compare two dictionaries

Functions Of Set

❏ add(): Adds an element to the set

❏ clear(): Removes all the elements from the set

❏ copy(): Returns a copy of the set
❏ difference(): Returns a set containing the difference between two or more sets
❏ difference_update(): Removes the items in this set that are also included in another, specified set
❏ discard(): Remove the specified item
❏ intersection(): Returns a set, that is the intersection of two or more sets
❏ intersection_update(): Removes the items in this set that are not present in other, specified set(s)
❏ isdisjoint(): Returns whether two sets have a intersection or not
❏ issubset(): Returns whether another set contains this set or not
❏ issuperset(): Returns whether this set contains another set or not
❏ pop(): Removes an element from the set
❏ remove(): Removes the specified element
❏ symmetric_difference(): Returns a set with the symmetric differences of two sets
❏ symmetric_difference_update(): inserts the symmetric differences from this set and another
❏ union(): Return a set containing the union of sets
❏ update(): Update the set with another set, or any other iterable

Q.32. What is Shallow copy and Deep copy in Python?

Ans:

Shallow Copy:

- A shallow copy creates a new object, but it does not recursively copy all the objects inside the original object. Instead, it copies references to the objects contained within the original object.
- As a result, changes made to objects inside the copied object will affect the original object, and vice versa.
- Shallow copies are typically created using the `copy.copy()` function or the `copy()` method of objects.
- **Example:**

```
import copy
original_list = [1, [2, 3], 4]
shallow_copied_list = copy.copy(original_list)
shallow_copied_list[1][0] = 99
```

```
print(original_list)        # Output: [1, [99, 3], 4]
print(shallow_copied_list)  # Output: [1, [99, 3], 4]
```

Deep Copy:

- A deep copy creates a completely independent copy of the original object along with all objects contained within it, recursively.
- Changes made to objects inside the copied object will not affect the original object, and vice versa.
- Deep copies are typically created using the **`copy.deepcopy()`** function.
- **Example:**

```
import copy
original_list = [1, [2, 3], 4]
deep_copied_list = copy.deepcopy(original_list)
deep_copied_list[1][0] = 99
print(original_list)        # Output: [1, [2, 3], 4]
print(deep_copied_list)     # Output: [1, [99, 3], 4]
```

Q.33. What is Context Manager in Python?

Ans: In Python, a context manager is an object that is used to manage resources, such as file handles, network connections, or database connections, in a way that ensures they are properly acquired and released. Context managers are often used with the **`with`** statement to ensure that resources are cleaned up automatically when they are no longer needed, even if an exception is raised within the code block.

Context managers are created by defining a class with two special methods: **`__enter__`** and **`__exit__`**.

1. **__enter__** method: This method is called when you enter the `with` block. It is responsible for setting up or acquiring the resource. The result of this method can be bound to a variable using the `as` keyword in the `with` statement.
2. **__exit__** method: This method is called when you exit the `with` block, whether normally or due to an exception. It is responsible for releasing or cleaning up the resource. It takes care of handling exceptions and cleanup operations.

Example:

```
class MyContextManager:
    def __enter__(self):
        print("Entering the context")
        return self  # Return an object that you want to use within the 'with' block
    def __exit__(self, exc_type, exc_value, traceback):
        print("Exiting the context")
        # You can perform cleanup operations here if needed

# Using the custom context manager
with MyContextManager() as cm:
    print("Inside the context")

# Output:
# Entering the context
# Inside the context
# Exiting the context
```

Q.34. Text File vs Binary File in Python?

Ans:

Text Files:

1. Text files are files that contain human-readable text, often in the form of characters, words, and lines of text.
2. They are encoded using character encoding schemes such as UTF-8, ASCII, or others, depending on the file's content.
3. Text files can be opened and manipulated using text-based I/O operations.
4. You can use functions like **`open()`**, **`read()`**, **`readline()`**, **`write()`**, and **`writelines()`** to work with text files.

   Example1(Read):

   ```
   with open('example.txt', 'r') as file:
       contents = file.read()
       print(contents)
   ```

   Example2(Write):

   ```
   with open('output.txt', 'w') as file:
   ```

```
    file.write('This is a text file.\n')
    file.write('You can write text data to it.')
```

Binary Files:

1. Binary files, on the other hand, can contain any type of data, including non-textual data like images, audio, video, or structured binary data formats.
2. Binary files are not meant to be human-readable and are encoded in a specific binary format.
3. You can open and manipulate binary files using binary I/O operations.
4. Functions like `open()` with the `'rb'` mode for reading and `'wb'` mode for writing are typically used for binary files.

   Example1(Read):
   ```
   with open('example.txt', 'rb') as file:
       contents = file.read()
       print(contents)
   ```
   Example2(Write):
   ```
   with open('output.txt', 'wb') as file:
       file.write('This is a text file.\n')
       file.write('You can write text data to it.')
   ```

Q.35. What is the 'self' keyword in python?

Ans: In Python, the self keyword is used as a conventional name for the first parameter of a method in a class. It represents the instance of the class that the method is called on. When you define a method in a class, you typically include self as the first parameter, although you can name it differently if you prefer. By convention, it's named self to make the code more readable and maintainable.

Example:

```
class MyClass:
    def __init__(self, value):
        self.value = value
    def get_value(self):
        return self.value
# Creating an instance of MyClass
```

```
obj = MyClass(42)
# Calling the get_value method on the instance
result = obj.get_value()
print(result)  # This will print: 42
```

Q.36. What is operator overloading in python?

Ans:

Operator overloading in Python refers to the ability to define custom behavior for standard operators, such as `+`, `-`, `*`, `/`, `==`, and others, when applied to objects of user-defined classes.

To implement operator overloading in custom classes, you need to define special methods, also known as magic methods or dunder methods, in your class. These methods have double underscores (__) as prefix and suffix, and they correspond to specific operators.

Here are some common operator overloading methods in Python:

- `__add__(self, other)`: Overloads the `+` operator.
- `__sub__(self, other)`: Overloads the `-` operator.
- `__mul__(self, other)`: Overloads the `*` operator.
- `__truediv__(self, other)`: Overloads the `/` operator.
- `__eq__(self, other)`: Overloads the `==` operator.

Example:

```
class ComplexNumber:
  def __init__(self, real, imag):
    self.real = real
    self.imag = imag
  def __add__(self, other):
    # Overload the + operator for complex number addition
    return ComplexNumber(self.real + other.real, self.imag + other.imag)
  def __sub__(self, other):
    # Overload the - operator for complex number subtraction
    return ComplexNumber(self.real - other.real, self.imag - other.imag)
  def __mul__(self, other):
    # Overload the * operator for complex number multiplication
```

```
        return ComplexNumber(
            self.real * other.real - self.imag * other.imag,
            self.real * other.imag + self.imag * other.real)

    def __truediv__(self, other):
        # Overload the / operator for complex number division
        denominator = other.real ** 2 + other.imag ** 2
        real_part = (self.real * other.real + self.imag * other.imag) / denominator
        imag_part = (self.imag * other.real - self.real * other.imag) / denominator
        return ComplexNumber(real_part, imag_part)

    def __eq__(self, other):
        # Overload the == operator for complex number equality
        return self.real == other.real and self.imag == other.imag

    def __str__(self):
        # String representation of the complex number
        return f"{self.real} + {self.imag}i"


# Usage
c1 = ComplexNumber(1, 2)
c2 = ComplexNumber(3, 4)
# Operator overloading in action
result_add = c1 + c2
result_sub = c1 - c2
result_mul = c1 * c2
result_div = c1 / c2
print(f"Addition: {result_add}")
print(f"Subtraction: {result_sub}")
print(f"Multiplication: {result_mul}")
print(f"Division: {result_div}")
```

# Equality check
print(f"Equality: {c1 == c2}")

Q.37. What is method overloading and how to achieve it in python?

Ans:

Method Overloading:

Method overloading refers to the ability to define multiple methods in a class with the same name but with different parameters or argument lists. In Python, method overloading is not directly supported as it is in some other programming languages like Java or C++. However, Python allows you to define multiple methods with the same name in a class, but only the last one defined will be used, effectively overriding the previous ones.

Example:

```
class Calculator:
    def add(self, a, b):
        return a + b
    def add(self, a, b, c):
        return a + b + c
calc = Calculator()
result =calc.add(2, 3, 4)  #This will call the second definition of add
print(result)  # Output: 9
```

Python allows you to define a single method with default arguments and handle method overloading based on the number and types of arguments passed to the method.   Here's how Python achieves method overloading:

Default Arguments: You can define default values for function parameters in Python. When you call the function without providing a value for a parameter with a default value, the default value will be used. This allows you to have a single method that can be called with different numbers of arguments.

Variable-Length Argument Lists: Python provides two special syntaxes for defining variable-length argument lists in functions: `*args` and `**kwargs`. `*args` allows you to pass a variable number of non-keyword arguments to a function, and `**kwargs` allows you to pass a variable number of keyword arguments (i.e., named arguments).

Example:

```
class Calculator:
    def add(self, a, b=0, c=0):
        return a + b + c
calc = Calculator()

result1 = calc.add(2)
result2 = calc.add(2, 3)
result3 = calc.add(2, 3, 4)
print(result1)  # Output: 2
print(result2)  # Output: 5
print(result3)  # Output: 9
```

Q.38. What is method overriding in python?

Ans: Method overriding in Python is a feature that allows a subclass to provide a specific implementation for a method that is already defined in its superclass.

This allows the subclass to customize the behavior of inherited methods without changing their names or parameters. Method overriding is an essential aspect of object-oriented programming and facilitates the creation of more specialized classes that inherit and extend the functionality of their parent classes.

To perform method overriding in Python, follow these rules: 1. The method in the subclass must have the same name and parameters as the method in the superclass that you want to override.

2. The method in the subclass should use the super() function to call the overridden method in the superclass, and then you can provide the specific implementation or modification.

Example:

```
class Animal:
    def speak(self):
        print("Animal speaks")
class Dog(Animal):
    def speak(self):
        super().speak()  # Call the parent class's speak method
        print("Dog barks")
class Cat(Animal):
    def speak(self):
        super().speak()  # Call the parent class's speak method
        print("Cat meows")

# Creating instances of Dog and Cat
dog = Dog()
cat = Cat()
# Calling the speak method on instance
dog.speak()
cat.speak()
```

Output:

```
Animal speaks
Dog barks
Animal speaks
Cat meows
```

Q.39. What is the class method?

Ans:

A class method is a method that is bound to the class itself, not to an instance of the class. Class methods are defined using the @classmethod decorator or the classmethod() function. When a class method is called, the first argument passed to the method is the class itself, not an instance of the class.

Class methods are often used to perform actions on the class as a whole, such as creating new instances of the class or getting or setting class attributes. Class methods can also be used to override existing methods on the class.

Example:

```
class Employee:
    company="Apple"
    def getcompanyname(self):
        print(self.company)

    @classmethod
    def changeCompanyName(cls,newcompany):
        cls.company=newcompany

E1=Employee()
E1.getcompanyname()  # Apple
E1.changeCompanyName("Volvo")
E1.getcompanyname()  #volvo
```

Q.40. What is a static method in python?

Ans: In Python, a static method is a method that belongs to a class rather than an instance of the class. Unlike regular methods, which are associated with instances and can access instance-specific data and attributes, static methods are associated with the class itself and do not have access to instance-specific data or attributes.They are defined using the @staticmethod decorator.
Static methods are typically used for utility functions that are related to the class but do not depend on instance-specific state. They can be called on the class itself without the need to create an instance of the class. Static methods do not have access to the self parameter, which is commonly used in instance methods to reference the instance itself.

Example:

```
class Math:
    def __init__(self,num):
        self.num=num

    #normal method
    def addnum(self,n):
        self.num=self.num+n
        return self.num
    @staticmethod
    def addnumstatic(a,b):
        return a+b

m1=Math(5)
# a=m1.addnum(5)  # 10
a=m1.addnumstatic(m1.num,5)  #10
print(a)
```

Class Method	Static Method
The class method takes cls (class) as first argument.	The static method does not take any specific parameter.
Class method can access and modify the class state.	Static Method cannot access or modify the class state.
The class method takes the class as parameter to know about the state of that class.	Static methods do not know about class state. These methods are used to do some utility tasks by taking some parameters.
@classmethod decorator is used here.	@staticmethod decorator is used here.

The Static methods are used to do some utility tasks, and class methods are used for factory methods.

Q.39. What is an Abstract class and Abstract method in Python?
Ans: In Python, an abstract class is a class that cannot be instantiated directly and is meant to serve as a blueprint or template for other classes. Abstract classes are typically used to define a common interface or set of methods that must be implemented by any concrete (sub)class that inherits from them. Abstract classes are defined using the abc module (Abstract Base Classes) and the ABC (Abstract Base Class) metaclass.
Here's an overview of abstract classes and abstract methods:
Abstract Class:

1. An abstract class is defined using the ABC metaclass from the abc module.
2. It cannot be instantiated on its own, which means you cannot create objects directly from an abstract class.
3. Abstract classes can contain both abstract methods and concrete methods.
4. Abstract classes serve as a blueprint for other classes and define a common interface that concrete subclasses must adhere to.

Abstract Method:

1. An abstract method is a method declared in an abstract class but without any implementation in the abstract class itself.
2. Abstract methods are defined using the @abstractmethod decorator.
3. Subclasses that inherit from an abstract class must provide concrete implementations of all the abstract methods defined in the parent abstract class.
4. Abstract methods ensure that specific behavior is implemented in concrete subclasses, enforcing a common interface.

Example:

```
from abc import ABC, abstractmethod
class Shape(ABC):  # Abstract class
    @abstractmethod
    def area(self):  # Abstract method
        pass

class Circle(Shape):  # Concrete subclass of Shape
    def __init__(self, radius):
        self.radius = radius
    def area(self):  # Concrete implementation of the abstract method
        return 3.14 * self.radius ** 2

class Square(Shape):  # Another concrete subclass of Shape
    def __init__(self, side_length):
        self.side_length = side_length

    def area(self):  # Concrete implementation of the abstract method
        return self.side_length ** 2

# Attempting to create an instance of the abstract class Shape will result in a TypeError
# shape = Shape()  # This will raise an error

# Creating instances of the concrete subclasses Circle and Square
circle = Circle(5)
```

```
square = Square(4)
print("Circle Area:", circle.area())
print("Square Area:", square.area())
```

Output:

```
Circle Area: 78.5
Square Area: 16
```

Q.40. What is encapsulation in python?

Ans: Encapsulation in Python is one of the fundamental concepts of object-oriented programming (OOP) that allows you to restrict access to certain parts of an object, typically its attributes and methods. It helps in hiding the internal details and state of an object while providing controlled and well-defined ways to interact with it.

In Python, there are three levels of encapsulation:

1. **Public**: By default, all attributes and methods in a class are considered public and can be accessed from outside the class. There are no access control modifiers to specify public members explicitly.
2. **Protected (Single Underscore Prefix):** Conventionally, attributes and methods that are intended to be considered protected are given a single underscore prefix (e.g., `_protected_variable`). This is more of a naming convention rather than strict access control, and it indicates that these members should be treated as non-public but still accessible if needed.
3. **Private (Double Underscore Prefix):** Attributes and methods that are intended to be private are given a double underscore prefix (e.g., `__private_variable`). Again, this is more of a convention than strict access control. Python uses name mangling to make it more difficult to access these members from outside the class, but it's still possible.

Example:

```
class MyClass:
    def __init__(self):
        self.public_var = 42
        self._protected_var = 24
        self.__private_var = 12
    def _protected_method(self):
        return "This is a protected method"
    def __private_method(self):
        return "This is a private method"

# Accessing members from outside the class
obj = MyClass()
print(obj.public_var)          # Accessing a public variable
print(obj._protected_var)      # Accessing a protected variable (discouraged but possible)
# Accessing a private variable (name mangling applied)
print(obj._MyClass__private_var)

# Accessing methods
print(obj._protected_method())   # Accessing a protected method (discouraged but possible)
# Accessing a private method (name mangling applied)
print(obj._MyClass__private_method())
```

Q.41. What is MRO(Method Resolution Order) in Python?

Ans: MRO stands for Method Resolution Order in Python. It is a concept related to the inheritance hierarchy of classes and is primarily used in the context of multiple inheritance. MRO determines the order in which classes are searched when an attribute (usually a method) is accessed on an object.

In Python, when you have a class hierarchy with multiple base classes, the MRO helps the interpreter decide from which class to inherit a method or attribute. This ensures that method and attribute lookup is consistent and predictable.

Example:

```
class A:
    def foo(self):
        print("A's foo")

class B(A):
    def foo(self):
        print("B's foo")

class C(A):
    def foo(self):
        print("C's foo")

class D(B, C):
    pass

d = D()
d.foo()
print(D.mro())
# Output: [<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class
'__main__.A'>, <class 'object'>
```

Q. Inheritance and its types in python?