Q1. What is the purpose of Python's OOP?

- Object-oriented Programming (OOPs) is used to design the program using classes and objects. Object could represent a person with properties like a name, age, and address and behaviours such as walking, talking, breathing, and running. Class is a blueprint for that object.
- Variables are referred to as properties of the object, and functions are referred to as the behaviour of the objects
- It is an approach for modelling concrete, real-world things, like cars, as well as relations between things, like companies and employees, students and teachers, and so on
- It aims to implement real-world entities like inheritance, polymorphisms, encapsulation, Data Abstraction called methodologies. It is used to bind the data and the functions that work on that together as a single unit.
- It is used for creating neat and reusable code

Q2. Where does an inheritance search look for an attribute?

- An inheritance search looks for an attribute first in the instance object, then in the class the instance was created from, then in all higher super classes (parent classes), progressing from left to right (by default). The search stops at the first place the attribute is found.
- Inheritance is a way of representing real-world relationships between classes (a blueprint or template from where objects are created) and real-world objects (instances of a class)
- Instead of writing the same code repeatedly, we can simply inherit the properties of an existing class into the other.

Q3. How do you distinguish between a class object and an instance object?

- A class is a type of blueprint that you can use to make objects. A concrete 'thing' that you constructed using a certain class is an object, which is an instance of a class. So, while the terms 'object' and 'instance' are interchangeable, the term 'instance' refers to an object's relationship to its class
- Classes are a kind of factory for creating multiple instances.
- Object is a generic term , it is physically present but remains undifferentiated . Instance is something that gives them a separate identity.
- **Eg:**

- *Class = Design to create a man*
- *Object = Creating a man*
- *Instance = Differentiating man by giving them different names*

Q4. What makes the first argument in a class's method function special?

- It always receives the instance object that is the implied subject of the method call. It's usually called 'self' by convention.
- The keyword 'self' is used to represent an instance (object) of the given class
- If there was no 'self' argument, the same class couldn't hold the information for multiple objects created
- The parameter self is a reference to the current instance of the class, and is used to access variables that belongs to the class.
- we need the keyword self to bind the object's attributes to the arguments received

Q5. What is the purpose of the 'init' method?

- The __init__ function is called every time an object is created from a class
- The __init__ method lets the class initialize the object's attributes and serves no other purpose. It is only used within classes.
- **Eg:**
  class person:
  def __init__(self, p_name ,p_age):
  self.name = p_name
  self.age = p_age
  employee = person("Nudrat","28")
- **Explanation:**
  Object = employee
  Class= person
  Arguments = ("Nudrat","28") - correspond to the respective parameters of the __init__ method
- The __init__ method uses the keyword self to assign the values passed as arguments to the object attributes 'self.name' and 'self.age'

Q6. What is the process for creating a class instance?

- To create instances of a class, you call the class using class name and pass in whatever arguments its __init__ method accepts.
- To create an instance of a class, you call the class as if it were a function
- Following statement **creates an instance** of the class
  x = ClassName()
- **Syntax** to **create the instance** of the class is given below
  <object-name> = <class-name>(<arguments>)

Q7. What is the process for creating a class?

- A class can be created by using the keyword class, followed by the class name
- **Syntax** to **create class**
  class ClassName:

Q8. How would you define the super classes of a class?

- The class from which a class inherits is called the parent or superclass. A class which inherits from a superclass is called a subclass, also called heir class or child class. Super classes are sometimes called ancestors as well. There exists a hierarchical relationship between classes. It forms a tree hierarchy where parent class is the root and subsequent subclasses are the leaves derived from their parent class.
- **Syntax** for **Inheritance**
  class DerivedClassName(BaseClassName):
- **Eg:**

```
# superclass
class Person():
    def display1(self):
        print("This is superclass")

# subclass
class Employee(Person):
    def display2(self):
        print("This is subclass")

emp = Employee()  # creating object of subclass
```

emp.display1()
emp.display2()

Q9. What is the relationship between classes and modules?

- Classes in python are templates for creating objects
- modules are python programs that can be imported into another python program. Importing a module enables the usage of the module's functions and variables into another program
- Modules in Python are files with a .py extension
- Difference is that Class is used to define a blueprint for a given object, whereas a module is used to reuse a given piece of code inside another program
- A class can have its own instance, but a module cannot be instantiated
- We use the 'class' keyword to define a class, whereas to use modules, we use the 'import' keyword
- We can inherit a particular class and modify it using inheritance but module is simply a code containing variables, functions, and classes

Q10. How do you make instances and classes?

- Classes in python are templates for creating objects
- Instances are objects of a class in Python
- A class can be created by using the keyword class, followed by the class name
- **Syntax** to **create class**
  class ClassName:
- use the "def" keyword to define an instance method
- use the "self" as the first parameter within the instance method
- **Creating Instance:**

```
class person:                          #Class
def __init__(self, p_name ,p_age):
self.name = p_name                     #Instance
self.age = p_age                       #Instance
```

Q11. Where and how should be class attributes created?

- Defined directly inside a class
- A class attribute is shared by all instances of the class. To define a class attribute, you place it outside of the __init__() method
- A class attribute is shared by all instances of the class. To define a class attribute, you place it outside of the __init__() method
- The class attributes don't associate with any specific instance of the class. But they're shared by all instances of the class
- You can access the class attribute via instances of the class or via the class name

Eg:
object_name.class_attribute
class_name.class_attribute

Q12. Where and how instance attributes created?

- Instance attributes belong to a specific instance of a class
- Instance attributes are attributes or properties attached to an instance of a class
- If you change the attributes of an instance, it won't affect other instances.
- Defined inside a constructor using the self parameter

Eg: Accessed using object
object.instance_attribute

Eg:

```
class Test:
  x = 10          #Class Attribute

   def __init__(self):
      self.x = 20     #Instance Attribute

test = Test()
print(test.x)          # Output 20
print(Test.x)          #Output 10
```

Q13. What does the term self in a python class mean ?

- The "self" parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class

- It does not have to be named·self·, you can call it whatever you like, but it has to be the first parameter of any function in the class
- By using the “self” ·we can access the attributes and methods of the class in python. It binds the attributes with the given arguments

Q14. How does a Python class handle operator overloading ?

- Operator Overloading·means giving extended meaning beyond their predefined operational meaning
- For example, “+” operator will perform arithmetic addition on two numbers, merge two lists, or concatenate two strings
- If you want to use the same operator to add two objects of some user defined class then you will have to define that behaviour yourself and inform python about that . We define a method for an operator and that process is called operator overloading. We can overload all existing operators
- This feature in Python that allows the same operator to have different meaning according to the context is called·operator overloading
- To perform operator overloading, Python provides some special function (also called double underscore methods or dunder methods)
- Mathematical Operator,Assignment Operator,Relational Operator have special functions to overload

Eg: Overloading “+” operator

```
class Complex:
# defining init method for class
def __init__(self, r, i):
self.real = r
self.img = i
# overloading the add operator using special function
def __add__(self, sec):
r = self.real + sec.real
i = self.img + sec.img
return complx(r,i)
# string function to print object of Complex class
def __str__(self):
return str(self.real)+' + '+str(self.img)+'i'
c1 = Complex(5,3)
c2 = Complex(2,4)
print("sum = ",c1+c2)       #Output 7 +7i
```

Q.15. When do you consider allowing operator overloading of your classes ?

- - When one or both operands are of a user-defined class or structure type, operator overloading makes it easier to specify user-defined implementation for such operations. This makes user-defined types more similar to the basic primitive data types in terms of behaviour.
- - When an operator is overloaded, its original operational meaning might be expanded upon

Q16. What is the most popular form of operator overloading ?

- Polymorphism means·“one action, many forms”. OOP allows objects to perform a·single·action in·different·ways. One way of implementing Polymorphism is through·operator overloading
- The most frequent instance is the adding up operator ‘+’, where it can be used for the usual addition and also for combining two different strings
- when you are using the·‘+’·operator, the·__add__·magical form can automatically describe the·‘+’·operator operation

  Eg:

  class Point:
  def __init__(self, x=0, y=0):
  self.x = x
  self.y = y
  def __str__(self):
  return "({0},{1})".format(self.x, self.y)
  def __add__(self, other):
  x = self.x + other.x
  y = self.y + other.y
  return Point(x, y)
  p1 = Point(1, 2)
  p2 = Point(2, 3)
  print(p1+p2)
  # Output: (3,5)

- when we use·p1 + p2, Python calls·p1.__add__(p2)·which in turn is·Point.__add__(p1,p2)

Q.17. What are the two most important concepts to grasp in order to comprehend Python OOP code ?

- Inheritance and Polymorphism are key ingredients for designing robust, flexible, and easy-to-maintain software

Eg:

```
class Employee():
  def __init__(self, emp_id, salary):
    self.emp_id = emp_id
    self.salary = salary
 def give_raise(self):
    self.salary = self.salary * 1.05
create an instance of the Employee class
emp1 = Employee(1001, 56000)
print(emp1.salary)
56000
emp1.give_raise()
print(emp1.salary)
58800.0
create the Manager class based on the Employee class
class Manager(Employee):
  def give_raise(self):
    self.salary = self.salary * 1.10
```

- Manager is said to be a child class of the Employee class. The child class copies the attributes from the parent class. This concept is called·inheritance
- We can partially inherit from a parent . Python also allows for adding new attributes as well as modifying the existing ones

create a manager object

```
mgr1 = Manager(101, 75000)
print(mgr1.salary)
75000
mgr1.give_raise()
print(mgr1.salary)
82500
create another child class of the Employee class
```

class Director(Employee):
def give_raise(self):
*self.salary = self.salary * 1.20*

- We have three different class and they all have a give_raise method. name of the method is the same, it behaves differently for different type of objects. This is an example of polymorphism .

Q.18. Describe three applications for exception processing ?

- An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions

Applications:

- **Error Handling:** Exception processing is commonly used for handling errors and exceptions that may occur during the execution of a program. By using try-except blocks, you can catch and handle specific types of exceptions, such as ValueError, TypeError, or FileNotFoundError. This allows you to gracefully handle errors and prevent your program from crashing or displaying cryptic error messages to users.
- **Input Validation:** When accepting user input, it's important to validate and handle potential errors or invalid data. For example, if you're expecting a user to enter an integer, you can use exception processing to catch and handle the ValueError that may occur if the user enters a non-numeric value. By validating and handling input exceptions, you can provide better feedback to users and prevent your program from encountering unexpected input.
- **Resource Management:** Exception processing is also useful for managing resources, such as files, database connections, or network connections. When working with resources, it's important to handle exceptions that may occur during operations like opening, reading, or writing. By using try-except-finally blocks, you can ensure that resources are properly closed or released, even if an exception occurs. This helps prevent resource leaks and ensures the efficient and reliable operation of your program

Q.19. What happens if you don't do something extra to treat an exception ?

- If you don't handle or treat an exception in Python, it will propagate up the call stack until it either reaches a higher-level exception handler or causes the program to terminate. Here's what typically happens if an exception is not treated:

- **Exception Traceback:** When an exception occurs and is not handled, Python prints an exception traceback. This traceback includes information about the type of exception, the line of code where the exception occurred, and the call stack leading up to the exception. The traceback helps identify the source of the exception and provides information for debugging.
- **Program Termination:** If an unhandled exception reaches the top-level of the program without being caught, the program will terminate abruptly. This means that any remaining code in the program will not be executed, and the program exits with an error code. The exact behavior may vary depending on the Python environment you're using.
- By treating an exception, you can handle errors and exceptions in a controlled manner, allowing your program to continue executing or providing appropriate feedback to users. This typically involves using try-except blocks to catch specific types of exceptions and providing code to handle the exception, such as logging an error message, displaying a user-friendly error message, or taking alternative actions to recover from the error.
- Handling exceptions allows you to gracefully recover from errors, prevent program crashes, and provide better feedback to users or log errors for debugging purposes. It's good practice to handle exceptions appropriately to ensure the stability and reliability of your Python programs.

Q20. What are your options for recovering from an exception in your script?

- In Python, you have several options for recovering from an exception and controlling the flow of your script. Here are some common approaches:

1. **Exception Handling with try-except:** You can use a try-except block to catch and handle specific types of exceptions. Inside the try block, you write the code that might raise an exception, and inside the except block, you handle the exception. This allows you to gracefully recover from the exception and continue the execution of your script.

Eg: try:

```
    # Code that might raise an exception
    ...
except ExceptionType:
    # Code to handle the exception and recover
    ...
```

2. **Multiple except Clauses:** You can have multiple except clauses to handle different types of exceptions separately. This allows you to provide specific error handling logic for each type of exception that may occur.
   try:
   # Code that might raise an exception
   ...
   except ExceptionType1:
   # Code to handle ExceptionType1
   ...
   except ExceptionType2:
   # Code to handle ExceptionType2
   ...
3. Finally Block: You can use a finally block to specify code that should always be executed, regardless of whether an exception occurs or not. This is useful for releasing resources or cleaning up operations that need to be performed regardless of exception handling.
   try:
   # Code that might raise an exception
   ...
   except ExceptionType:
   # Code to handle the exception
   ...
   finally:
   # Code that always gets executed
   ...
4. Raising Exceptions: In some cases, you might want to explicitly raise an exception to indicate an error or exceptional condition. You can use the raise statement to raise a specific exception at any point in your code. This allows you to control the flow and behavior based on specific conditions.
   if condition:
   raise ExceptionType("Error message")

- By using these techniques, you can handle exceptions, recover from errors, and control the flow of your script based on different conditions. It's important to choose the appropriate approach based on your specific requirements and the types of exceptions you expect to handle.

Q. 21. Describe two methods for triggering exceptions in your script.

- In Python methods available to trigger exceptions in your script are:

1. Using the raise Statement:

The raise statement allows you to explicitly raise an exception of a specified type. It can be used to trigger built-in exceptions or custom exceptions defined by the user. The general syntax for raising an exception is as follows:

raise ExceptionType("Error message")

2. Invoking Built-in Exceptions:

Python provides a wide range of built-in exceptions that can be raised to handle specific error conditions. You can directly invoke these exceptions to trigger the corresponding errors. For instance, some commonly used built-in exceptions include ValueError, TypeError, FileNotFoundError, and IndexError, among others.

To raise a built-in exception, you can simply write the keyword followed by parentheses

raise ValueError

Q. 22. Identify two methods for specifying actions to be executed at termination time, regardless of whether or not an exception exists.

1. **Using the try-except-finally Statement:**

The try-except-finally statement allows you to specify actions to be executed at termination time, regardless of whether an exception is raised or not. The finally block is executed after the try block, regardless of whether an exception is raised or caught in the except block.

The general syntax of the try-except-finally statement is as follows:

```
try:
    # Code that may raise an exception
except ExceptionType:
    # Code to handle specific exceptions
finally:
    # Code to be executed at termination time
```

The finally block is guaranteed to execute, even if an exception occurs or is caught. It is commonly used for cleanup tasks such as releasing resources or closing files.

2. Using a try-finally block with a loop or iterator:

By using a try-finally block along with a loop or iterator, you can ensure that termination actions are executed even if an exception is raised during iteration.

```
try:
    for item in iterable:
        # Code to process each item
finally:
    # Code to be executed at termination time
```

Q.23. What is the purpose of the try statement?

- The purpose of the try statement in Python is to define a block of code in which exceptions can potentially occur. It allows you to handle potential errors and exceptions in a controlled manner. By enclosing a section of code within a try block, you can specify how to handle exceptions that may be raised during the execution of that code.

- The try statement works in conjunction with except, else, and finally clauses to handle different aspects of exception handling:

- The except clause allows you to define specific exception types that you want to catch and handle. If an exception of the specified type occurs within the try block, the corresponding except block is executed.

- The else clause is optional and is executed if no exception occurs within the try block. It is typically used for code that should run only when no exceptions are raised.

- The finally clause is also optional and is executed regardless of whether an exception is raised or not. It is often used for cleanup operations that need to be performed, such as closing files or releasing resources, regardless of the outcome of the try block

Eg:

```
try:
    # Code that may raise an exception
    # ...
    # ...
except ExceptionType:
    # Code to handle the specific exception type
    # ...
else:
    # Code to be executed when no exception occurs (optional)
    # ...
finally:
    # Code to be executed regardless of exceptions (optional)
```

```
    # ...
```

Q.24. What are the two most popular try statement variations?

- The two most popular variations of the try statement in Python are:

1. try-except:

The try-except statement is the most commonly used variation of the try statement. It allows you to catch and handle specific exceptions that may occur within the try block. By specifying one or more except clauses, you can define the type(s) of exceptions to catch and provide the corresponding code to handle each exception.

Here's an example of the try-except statement:

```
try:
    # Code that may raise an exception
except ExceptionType1:
    # Code to handle ExceptionType1
except ExceptionType2:
    # Code to handle ExceptionType2
...
except ExceptionTypeN:
    # Code to handle ExceptionTypeN
```

In this variation, if an exception of ExceptionType1 is raised within the try block, the corresponding except block for ExceptionType1 is executed. Similarly, if an exception of ExceptionType2 is raised, the corresponding except block for ExceptionType2 is executed, and so on. You can have multiple except clauses to handle different exception types or scenarios.

2. try-finally:

The try-finally statement is another popular variation of the try statement. It combines the try block with a finally block that is executed regardless of whether an exception occurs or not. The finally block is useful for specifying cleanup operations that should be performed, such as closing files or releasing resources, regardless of the outcome of the try block.

Here's an example of the try-finally statement:

```
try:
    # Code that may raise an exception
finally:
    # Code to be executed at termination time
```

In this variation, the finally block is executed regardless of whether an exception is raised within the try block or not. It ensures that the specified code in the finally block is executed, even if an exception occurs. This is particularly useful for ensuring proper cleanup and resource release.

Q.25. What is the purpose of the raise statement?

- The purpose of the raise statement in Python is to explicitly raise an exception within your code. It allows you to generate and trigger exceptions at specific points in your program, indicating that an error or exceptional condition has occurred.

- The raise statement is typically used in the following scenarios:

- Raising Built-in Exceptions:

Python provides a set of built-in exceptions, such as ValueError, TypeError, FileNotFoundError, and many others. You can use the raise statement to raise these exceptions, providing valuable information about the error condition.

Here's an **example** of raising a ValueError exception:

raise ValueError("Invalid value detected.")

In this case, the raise statement triggers a ValueError exception with a custom error message. This allows you to communicate and handle specific error situations in your code.

- **Raising Custom Exceptions:**

In addition to built-in exceptions, you can define your own custom exception classes by creating subclasses of the Exception class or its derived classes. The raise statement can be used to raise instances of these custom exception classes.

Here's an **example** of raising a custom exception:

class MyCustomException(Exception):

pass

raise MyCustomException("Something went wrong.")

In this case, the raise statement triggers the MyCustomException exception, which can be caught and handled elsewhere in your code.

Q.26. What does the assert statement do, and what other statement is it like?

- The assert statement in Python is used to assert or verify a condition in your code. It allows you to specify a condition that must be true for the program to continue executing. If the condition evaluates to False, an AssertionError exception is raised.
- **The assert statement has the following syntax:**

  assert condition, [error_message]

- Here, condition is the expression that is tested, and error_message (optional) is an error message that can be displayed when the assertion fails. If the condition is False, the assert statement raises an AssertionError exception, indicating that the condition was not met.
- The purpose of the assert statement is to catch programming errors or unexpected conditions during development and testing. It is often used to validate assumptions and ensure that certain conditions hold true. If an assert statement fails, it indicates a bug or a problem in the code that needs to be addressed.
- The assert statement is similar to the if statement, but with a fundamental difference. While an if statement allows you to conditionally execute code based on a condition, the assert statement is used specifically for sanity checks or assertions that must hold true at certain points in your code.
- Here's an **example** to illustrate the **usage of the assert statement:**

  def divide(a, b):

  assert b != 0, "Division by zero is not allowed."

  return a / b

  result = divide(10, 0) # Raises an AssertionError

- In this example, the assert statement checks if the divisor b is not equal to zero before performing the division operation. If the condition b != 0 is False, the assert statement raises an AssertionError with the specified error message.

Q.27. What is the purpose of the with/as argument, and what other statement is it like?

- The with/as statement in Python is used for resource management and ensures that resources are properly acquired and released. It provides a convenient way to work with objects that support the context management protocol. The with statement guarantees that the necessary setup and teardown actions are performed, even if an exception occurs within the block.

- The general **syntax of the with/as statement** is as follows:

- *with expression as variable:*

 # Code block using the variable

- Here, expression evaluates to an object that supports the context management protocol. The object's __enter__() method is called at the beginning of the block, and its __exit__() method is called at the end of the block.
- The purpose of the with/as statement is to simplify resource management, such as opening and closing files, acquiring and releasing locks, establishing and closing network connections, and more. It eliminates the need for explicit setup and teardown code, ensuring that resources are properly handled.
- An alternative to the with/as statement is using the try/finally construct. However, the with/as statement provides a more concise and readable way of handling resources, especially when multiple resources need to be managed simultaneously.
- Here's an example that demonstrates the usage of the with/as statement with a file object with open("example.txt", "r") as file:

 # Code block using the file object

 # ...

 # Code outside the 'with' block

 # The file is automatically closed at this point:

- In this example, the with statement is used to open a file in read mode. The file object is assigned to the variable file using the as keyword. The indented code block within the with statement can freely use the file object. Once the block is exited, the __exit__() method of the file object is called automatically, ensuring that the file is closed, even if an exception occurs.

Q.28. What are *args, **kwargs?

- In Python, *args and **kwargs are special syntax used in function definitions to allow for variable-length arguments. They provide flexibility when working with functions that can accept a varying number of arguments.

Here's an explanation of each:

***args (Positional Arguments):**

- The *args syntax allows a function to accept an arbitrary number of positional arguments. The term "args" is just a convention; you can use any valid variable name preceded by an asterisk. When the function is called, the arguments passed to it are collected into a tuple.

For example:

```
def my_function(*args):
    for arg in args:
        print(arg)
my_function('Hello', 'World', 123)
```

- In this case, the function my_function accepts any number of arguments. The arguments are collected into the args tuple within the function, and they can be accessed and processed accordingly.

****kwargs (Keyword Arguments):**

- The **kwargs syntax allows a function to accept an arbitrary number of keyword arguments or named arguments. The term "kwargs" is just a convention; you can use any valid variable name preceded by two asterisks. When the function is called, the keyword arguments passed to it are collected into a dictionary, where the keys are the parameter names and the values are the corresponding argument values.

For example:

```
def my_function(**kwargs):
    for key, value in kwargs.items():
        print(key, value)
my_function(name='John', age=30, city='New York')
```

In this case, the function my_function accepts any number of keyword arguments. The arguments are collected into the kwargs dictionary within the function, and they can be accessed and processed as key-value pairs.

Q.29. How can I pass optional or keyword parameters from one function to another?

- To pass optional or keyword parameters from one function to another in Python, you can use the *args and **kwargs unpacking operators when calling the second function. These operators allow you to pass the arguments received by the first function as arguments to the second function.

- Here's an example to illustrate how to pass optional or keyword parameters from one function to another:

```
def function1(*args, **kwargs):
    # Process arguments if needed
    # ...
    # Call function2 and pass the arguments
    function2(*args, **kwargs)
def function2(*args, **kwargs):
    # Process the passed arguments
    # ...
```

- In this example, function1 receives any number of positional arguments as *args and keyword arguments as **kwargs. It then calls function2 and passes the received arguments using the *args and **kwargs unpacking operators.

- When function2 is called from function1 with function2(*args, **kwargs), the arguments received by function1 are passed as arguments to function2 in the same order. The positional arguments are unpacked using *args, and the keyword arguments are unpacked using **kwargs.

Q.30. What are Lambda Functions?

- Lambda functions, also known as anonymous functions, are small, inline functions in Python that are defined using the lambda keyword. They are called "anonymous" because they don't require a named function definition like regular functions defined with the def keyword.

- Lambda functions are typically used for simple and concise operations that can be expressed in a single expression. They are commonly used in functional programming paradigms and situations where a small function is needed as an argument to another function.

The **syntax of a lambda function** is as follows:

lambda arguments: expression

- Here, arguments represents the input parameters of the function, and expression is the result of the function. The lambda function evaluates the expression and returns the result.
- Lambda functions can have any number of arguments, including zero arguments. The expression can be any valid Python expression, and it can use the arguments provided.
- Here's an example of a lambda function that calculates the square of a number:

```
square = lambda x: x**2

result = square(5)

print(result)  # Output: 25
```

- In this example, the lambda function lambda x: x**2 takes a single argument x and returns the square of x. The function is assigned to the variable square, and then it is called with the argument 5, resulting in 25.
- Lambda functions are often used in conjunction with higher-order functions like map(), filter(), and reduce(), where a simple function is required as an argument.
- - Lambda functions have some limitations compared to regular functions. They can only contain a single expression, and they cannot include statements or complex logic. For more complex operations, it is recommended to use regular functions defined with def.

Q.31. Explain Inheritance in Python with an example?

- Inheritance is a fundamental concept in object-oriented programming that allows a class to inherit properties and behaviors from another class. In Python, a class can inherit attributes and methods from a parent class, also known as a base class or superclass, to create a new child class or subclass.

- The child class inherits the characteristics of the parent class, and it can add its own attributes and methods or override the inherited ones. This promotes code reuse and facilitates the creation of hierarchical relationships between classes.

Here's an **example to demonstrate inheritance** in Python:

```
class Animal:
```

```
    def __init__(self, name):
        self.name = name
    def sound(self):
        pass
class Dog(Animal):
    def sound(self):
        return "Woof!"
class Cat(Animal):
    def sound(self):
        return "Meow!"
dog = Dog("Buddy")
cat = Cat("Whiskers")
print(dog.name)       # Output: Buddy
print(dog.sound())    # Output: Woof!
print(cat.name)       # Output: Whiskers
print(cat.sound())    # Output: Meow!
```

- In this example, we have a parent class Animal with an __init__ method that initializes the name attribute and a sound method defined with the pass statement, indicating that it is meant to be overridden by the child classes.
- The child classes Dog and Cat inherit from the Animal class using the syntax class ChildClass(ParentClass). They define their own sound method, which overrides the inherited method.
- We create instances of the Dog and Cat classes, passing the name argument to the parent class __init__ method. We then access the name attribute and call the sound method on each instance, which produces the expected outputs.
- In this example, Dog and Cat are specialized classes that inherit the common behavior and attributes from the Animal class. They can add their own specific behaviors or attributes while utilizing the shared functionality from the parent class.

Q.32. Suppose class C inherits from classes A and B as class C(A,B).Classes A and B both have their own versions of method func(). If we call func() from an object of class C, which version gets invoked?

- When class C inherits from classes A and B as class C(A, B), and both A and B have their own versions of the method func(), the version of func() that gets invoked depends on the method resolution order (MRO) of the classes.

- In Python, the MRO determines the order in which the base classes are searched for a method or attribute. The MRO is determined by the C3 linearization algorithm, which follows a specific order called the Method Resolution Order.

- The MRO can be accessed using the __mro__ attribute or the mro() method of a class. It returns a tuple representing the order in which the classes are searched for methods or attributes.

- To determine which version of func() gets invoked, Python will search for the method in the following order:

The class C itself.

Class A (the first base class in the inheritance list).

Class B (the second base class in the inheritance list).

Any other base classes if there are more in the inheritance hierarchy.

- The method resolution stops as soon as a matching method is found. Therefore, if a matching method func() is found in class C, it will be invoked. If not found in C, Python will search for it in class A, and if still not found, it will finally search in class B.

Here's an **example to illustrate the behavior**:

```
class A:
    def func(self):
        print("Method func() in class A")
class B:
    def func(self):
        print("Method func() in class B")
class C(A, B):
    pass
```

c = C()

c.func()

- In this example, class C inherits from classes A and B, but it does not have its own implementation of the method func(). When c.func() is called, the output will be:

Method func() in class A

- Since A is the first base class in the inheritance list, its version of func() is invoked, even though B also has its own version of func().
- If the order of inheritance was reversed (class C(B, A)), the output would be:

Method func() in class B

- In this case, as B is the first base class in the inheritance list, its version of func() would be invoked.

- The method resolution order and which version of the method gets invoked can be controlled by changing the order of base classes in the inheritance list when defining class C.

Q.33. Which methods/functions do we use to determine the type of instance and inheritance?

- In Python, you can use the following methods/functions to determine the type of an instance and to check inheritance relationships:

type(): The type() function returns the type of an object. It is commonly used to determine the type of an instance. **For example**:

class MyClass:

pass

obj = MyClass()

print(type(obj)) # Output: <class '__main__.MyClass'>

- In this example, type(obj) returns <class '__main__.MyClass'>, indicating that obj is an instance of the MyClass class.

isinstance(): The isinstance() function is used to check if an object is an instance of a specific class or its subclasses. It considers inheritance relationships. It

returns True if the object is an instance of the specified class or any of its subclasses, and False otherwise**. For example**:

```
class Animal:
    pass
class Dog(Animal):
    pass
dog = Dog()
print(isinstance(dog, Dog))       # Output: True
print(isinstance(dog, Animal))    # Output: True
```

In this example, isinstance(dog, Dog) returns True because dog is an instance of the Dog class. Similarly, isinstance(dog, Animal) returns True because Dog is a subclass of Animal.

issubclass(): The issubclass() function is used to check if a class is a subclass of another class. It determines if one class is derived from another class. It returns True if the first class is a subclass of the second class, and False otherwise. For **example**:

```
class Animal:
    pass
class Dog(Animal):
    pass
print(issubclass(Dog, Animal))    # Output: True
print(issubclass(Animal, Dog))    # Output: False
```

- In this example, issubclass(Dog, Animal) returns True because Dog is a subclass of Animal. However, issubclass(Animal, Dog) returns False because Animal is not a subclass of Dog.

Q.34. Explain the use of the 'nonlocal' keyword in Python.

- The nonlocal keyword in Python is used to declare that a variable inside a nested function is a nonlocal variable. It allows the nested function to access and modify a variable that is defined in its nearest enclosing scope, but not in the global scope.

Here's an **example to illustrate the use of the nonlocal keyword**:

```
def outer_function():
```

```
    x = 10
    def inner_function():
        nonlocal x
        x += 5
        print("Inner function:", x)
    inner_function()
    print("Outer function:", x)
outer_function()
```

- In this example, we have an outer_function() that defines a variable x with an initial value of 10. Inside outer_function(), there is a nested inner_function(). Normally, the nested function would have its own local scope, and any modifications to x would create a new local variable with the same name, without affecting the x in the outer function.
- However, by using the nonlocal keyword before the variable x inside the inner_function(), we indicate that x is not a local variable but a nonlocal variable that is defined in the nearest enclosing scope (the outer_function() scope).
- As a result, when inner_function() is called, it can access and modify the nonlocal variable x from the outer function's scope. In this case, x is incremented by 5, and both the inner and outer functions print the updated value of x.

The output of the above example will be:

Inner function: 15

Outer function: 15

- Without the nonlocal keyword, modifying x inside the inner_function() would create a new local variable, and the outer function's x would remain unaffected.
- The nonlocal keyword is particularly useful when dealing with nested functions or closures, where you want to modify variables from the enclosing scope without resorting to global variables. It provides a way to bridge the gap between the local and global scopes, allowing access to variables in intermediate scopes.

Q.35. What is the global keyword?

- The global keyword in Python is used to indicate that a variable defined inside a function should be treated as a global variable, meaning it can be accessed and modified both within and outside the function.

- By default, when you define a variable inside a function, it is considered a local variable, limited in scope to that particular function. The global keyword allows you to override this behavior and explicitly state that the variable is a global variable.

Here's an **example to illustrate the use of the global keyword**:

```
x = 10

def my_function():
    global x
    x += 5
    print(x)
my_function()
print(x)
```

- In this example, we have a global variable x with an initial value of 10. Inside the my_function() function, we use the global keyword before the variable x. This declaration tells Python that x refers to the global variable x, rather than creating a new local variable.
- As a result, when my_function() is called, the value of the global variable x is incremented by 5, and both the function and the subsequent print statement outside the function display the updated value.

The **output** of the above example will be:

```
15
15
```

- Without the global keyword, modifying x inside the function would create a new local variable, and the global variable x would remain unchanged.
- It is important to use the global keyword with caution, as it can lead to code that is harder to understand and maintain. Global variables can introduce unintended side effects and make the code less modular. It is generally recommended to use local variables whenever possible and pass values between functions using parameters and return values instead. The use of global variables should be limited to cases where they are truly necessary.